Advanced Environmental Data Management

Amar Kolapkar

Contents

Preface

This book template is what I learned from *Advanced Data Management*. Sean Kross has a much better template on GitHub.

A few things you may want to do before building the book:

1. Create a designated folder for all files needed for the book –
    - All .Rmd files
    - .yml file(s)
    - R script file if needed
2. Create an r-project
3. Install the package "bookdown" – needed to build the book
4. Read the help file of the function `bookdown::render_book`

Once you have everything in one place, do the following:

```
## for PDF:
bookdown::render_book("index.Rmd", "bookdown::pdf_book")

## for HTML:
bookdown::render_book("index.Rmd", "bookdown::gitbook")
```

We have now completed a book using a basic template. For more options, consult the book by Yihui Xie.

1 Week 2: Intoduction to R Programming

1.1 Importing Data into R

1.1.1 Typing Vectors and Matrices

We used function `c` for entering small data into R. If we have a lot to type in, the function `scan` is more approrpiate when data are in the same mode. We specify the mode by using argument `what=`. The default mode for `scan` is numeric.

```
x <- scan()
1 3 1 3.2
```

We can change the default to read in character values:

```
name <- scan(what="")
joe fred bob john
sam sue mary ann
```

To enter data with different modes

```
pets <- scan(what=list(a=0, b="", c=0))
1 dog 3
2 cat 4
3 duck 8
```

We can enter a matrix:

```
x <- matrix(scan(), ncol=3)
19 30 22
2 5 1
9 3 0
11 34 56
```

1.1.2 Reading Data Frame

The most useful function for importing data to R is `read.table`. It returns a data frame, suitable for reading data files with mixed modes. When we have a single mode data file, `scan` is more efficient. `read.table` expects each field in the input source to be separated by one or more separators (default: any of spaces, tabs, newlines or cariage returns). The `sep=` argument can be used to sepcify alternative separators. When there is no separator in the input data, but each variable occupies the same columns for every observation, we use `read.fwf` function.

If the first line in your input data has variable names separated by the same separator as the data, the `heder=T` argument should be passed to `read.table` to use these names to identify columns. Alternatively, the `col.names=` argument can specify a character vector containing the variable names.

The only required argument to `read.table` is a file name, URL, or connection object. In Windows, make sure that **double backslashes** are used in pathnames. Alternatively, we can use the Unix convention, single forwardslash, for pathnames. You can also mix double backslashes and single forwardslash:

```
my.data <- read.table("c:/myclass\\subject1\\notes/data\\mydata.txt")
```

Because it offers increased efficiency in storage, `read.table` automatically converts character variables to factors (check if it is true: `default.stringAsFactor()`). This may cause some problem when trying to use the variable. We can either change the global option `options(stringAsFactors=FALSE)`, or convert factors back to characters when needed.

Any text after a `#` sign is treated as comment (not read into R). If the source use a different character for comments (e.g., `*`), we can use the `comment.char=` argument. If there are no comments, setting `comment.char=""` may speed up reading.

The arguments `skip` and `nrows` allow us to skip a number of lines at the beginning of the file and read into R only a specific number of lines.

`read.table` expects the same number of fields on each line and will return an error if it detects otherwise. When `read.table` reports unequal number of fields, we can use the `count.fields` function to help determine where the problem is.

Most of data files use comma or tab as separator. R functions `read.csv`, `read.csv2`, and `read.delim` are wrappers for `read.table` with appropriate arguments set for comma-, semicolon-, or tab-delimited data, respectively.

1.1.3 Fixed-with Input Files

Many government agencies store data using fixed-width file – input data is stored with no delimiters between values but with each variable occupying the same columns on each line of input. The function `read.fwf` can be used for such files. For an example of such files, we can go to USGS Water Quality site. Stream flow and water quality data are stored under the "Water Quality and Streamflow Data'' tab. 33 As an example, we download nutrient data from Maumee River (Waterville station) and the format file for water quality data. The format file from the USGS site describes the format of the file. All files starts with basic information on sites and data collection:

For data file with nutrient concentrations,

To use `read.fwf`, we need to specify `widths=` with a vector of the widths of the fields to be read, using negative numbers to indicate columns to be skipped. Because variable names are also specified in the format file, we will use `header=FALSE` and `col.names=` to give variable names. The downloaded file is large. To avoid typing errors, I will copy and past the format file to extract the needed values – deleting text before the format and comment out text after the format number:

Likewise, we need to name each column:

The above code should be kept, in case we have typed something wrong. With `colW` and `colN`, we can now read the file into R:

```
Maumee.data <- read.fwf(file="https://pubs.usgs.gov/dds/wqn96cd/wqn/wq/region04/04193500.nut",
                        widths=colW, col.names=colN, header=F)
# head(Maumee.data)
save(Maumee.data, file = "maumee.RData")
```

1.1.4 Generating Data

From time to time, we need to create data in R. R has a number of ways to generate vectors of data.

1.1.4.1 Sequences To generate a sequence of integers between two values, the colon operator (`:`) can be used:

```
1:12
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

To create a vector of value from 10 to 100, each element separated by 5, we use:

```
seq(from = 10, to = 100, by = 5)
```

```
##  [1]  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90  95 100
```

Alternatively, we can specify the length of the generated sequence:

```
seq(from = 10, by = 5, length = 10)
```

```
##  [1] 10 15 20 25 30 35 40 45 50 55
```

We can also general a sequence of values evenly spaced between two values

```
seq(from=1, to=10, length=20)
```

```
##  [1]  1.000000  1.473684  1.947368  2.421053  2.894737  3.368421  3.842105
##  [8]  4.315789  4.789474  5.263158  5.736842  6.210526  6.684211  7.157895
## [15]  7.631579  8.105263  8.578947  9.052632  9.526316 10.000000
```

Two functions frequently used in generating levels of a designed experiment. One is `gl` (short for generate levels):

```
gl(4, 3, 24) ## 4 levels, each with 3 replicates, and length 24
```

```
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4 1 1 1 2 2 2 3 3 3 4 4 4
## Levels: 1 2 3 4
```

```
thelevels <- data.frame(group=gl(3, 10, length=30),
                        subgroup=gl(5, 2, length=30),
                        obs = gl(2, 1, length=30))
```

The other function is `expand.grid`, which takes a number of sequences and returns a data frame with one row for each unique combinations of input values:

```
oe <- expand.grid(odd=seq(1,5, by=2), even=seq(2,5, by=2))
```

This function is especially useful for generating data for plotting 3D plots (contour and perspective plot).

1.1.5 Random Number Generators

R provides random number generators for about 20 probability distributions. Names of functions for these generators start with letter `r` follwed by the abrreviation of the distribution. For example `rnorm` is a generator for the normal distribution. These functions take `n=` as the first argument (number of random numbers to be generated) and other arguments are based on different distributions.

```
rnorm(10)
```

```
##  [1] -0.66914457 -0.09596419 -1.40630727  0.78522236 -0.14307866  0.22548904
##  [7]  0.84226928  0.35750621 -0.52201658 -0.32810762
```

1.1.6 Random Permutation

The function `sample` is a flexible function for random permutation of a vector (when the first argument is a vector) or of the index starting from 1 (when the first argument is a single integer).

```
sample(10, 3)
```

```
## [1] 6 8 5
```

```
sample(10)
```

```
##  [1]  2  8  3  1  7  9  5  6 10  4
```

1.1.7 Spreadsheet

The package `gdata` has a function `read.xls` that can be used to read Excel spreadsheet files. The function requires `perl`, which is installed in OS X, Unix, and Linux, but not necessarily on Windows. `read.xls` translate a specified spreadsheet to a comma-separated file, then calls `read.csv` to read the file into R.

1.2 Saving R Data

Each time when an R session ends, R will prompt you to save the `Workspace` (all data imported and processed) into a file named `.RData` in the working directory. The saved data will be loaded automatically when R starts the next time.

Most of the time, we don't need to save the entire work space, but only a few selected files. For example, we can save file `x, y, z` using function `save`:

```
save(x,y,z, file="myxyz.RData")
```

Once the data is saved, it can be loaded with function `load`:

```
load("myxyz.RData")
```

1.2.1 Writing R Objects to Files in ASCII Format

In many cases, we want to save data into text files. In R, we use `write`. It accepts an R object and the name of a file, and writes an ASCII representation of the object into the appropriate destination. In most cases, we save numeric objects such as matrices. When writing data into a text file, the file is filled by row. The number of values to be written in a line is specified by the argument `ncolumns=`. Because R stores a matrix by column, R will export the matrix in that order. That is, the first column of a matrix will be saved to fill the first row in the output file. If we want to save a matrix by row, we need to first transpose the matrix and specify `ncolumns=` accordingly.

```
write(t(state.x77), file="state.txt", ncolmns=ncol(state.x77))
```

For mixed mode data, like data frams, we use `write.table` to produce ASCII files. The first argument of the function is the data object and the second argument is `file=`, the destination. By default, character strings are surrounded by quotes. this feature can be suppressed by using argument `quote=FALSE`. We can also suppress row and column names by using `row.names=FALSE` and `col.names=FALSE`. The `sep=` argument is used to specify a separator (default is a blank space).

1.2.2 Reading Data from Other Programs

We often need to access data created by a program other than R or to create a data file that can be easily accepted. The package `foreign` provides functions to read and write in formats supported by a number of different programs.

1.3 Functions in `foreign` package

Function	Purpose
`data.restore`	read `data.dump` output
`read.S`	or saved objects from S version 3
\	may work with older Splus objects
`read.dbf`	saved object from DBF
`read.dta`	read saved objects from Stata
`write.dta`	create a Stata saved object
`read.epinfo`	read saved object from `epinfo`
`read.spss`	read saved object from SPSS
`read.mtp`	read Minitab Portable Worksheet file
`read.octave`	GNU Octave
`read.sport`	SAS export format
`read.systat`	saved object from Systat rectangular data

1.4 Date Objects

A commonly used method for processing dates and time in computer programming is the POSIX standard. It measures dates and times in seconds since beginning of 1970 in UTC time zone. In R, `{POSIXct` is the R date class for this standard. The `POSIXlt` class breaks down the date object into year, month, day of the month, hour, minute, and second. The `POSIXlt` class also calculates day of the week and day of the year (julian day, or ordinal day). The `Date` class are similar but with dates only (without time).

Typically, dates are entered as characters. For example, dates are typically entered in the U.S. using numeric values in a format of mm/dd/yyyy (e.g., 5/27/2000) or with month name plus numeric day and year (e.g., December 31 2013). When read into R, the date column becomes a factor variable. We can use the function `as.Date` to convert the factor variable into dates:

```
> first.date <- as.Date("5/27/2000", format="%m/%d/%Y")
> second.date <- as.Date("December 31 2003", format="%B %d %Y")
> second.date - first.date
```

The first two lines convert two character strings to date class objects. As date objects are numeric (days since the beginning of 1970), we can use them to calculate days eclipsed between two dates. A more general function for converting date-time object is `strptime`, which converts a date-time character string to a POSIXlt class object, measuring time in seconds since the beginning of 1970.

```
first.d <- strptime("5/27/2000 22:15:00",
                    format="%m/%d/%Y %H:%M:%S")
second.d <- strptime("December 31, 2003, 4:25:00",
                     format="%B %d, %Y, %H:%M:%S")
second.d - first.d
```

The format of a date object is defined by the POSIX standard, consists of a % followed by a single letter:

Once a date object is created, we can extract relevant information associated with dates using function `format`. We now create a data frame with a date column:

```
> mytime <- data.frame(x = rnorm(100),
+                      date=as.Date(round(runif(100)*5000),
+                          origin="1970-01-01"))
```

We can add a column of month and a column of week days to the data frame:

```
> mytime$Month <- format(mytime$date, "%b")
> mytime$weekday <- format(mytime$date, "%a")
```

We can also store date object as a POSIXlt class object, which is a list of nine elements: (1) seconds, (2) minutes, (3) hours, (4) day of month (1-31), (4) month of year (0-11), (5) month of the year (0-11), (6) years since 1900, (7) day of the week (0, Sunday, through 6), (8) day of the year (0-365), and (9) daylight savings indicator. If we want to extract day of the year, month, and year as numeric vectors, we can simply assign the eighth (Julian day), fifth (month), and sixth (year) elements:

```
mydata$Julian <- as.POSIXlt(mydata$Date)[[8]]+1
mydata$Month <- as.POSIXlt(mydata$Date)[[5]]+1
mydata$Year <- as.POSIXlt(mydata$Date)[[6]]+1900
```

In the Maumee River data imported from the USGS site using `read.fwf`, sampling dates were entered in three columns: `byear`, `bmonth`, and `bday`. To create a column of `dates`, we need to paste these three columns into a date object:

```
Maumee.data$Dates <- as.Date(paste(Maumee.data$bmonth,
                                   Maumee.data$bday,
                                   Maumee.data$byear,
                                   sep="/"),
                             format="%m/%d/%Y")
range(Maumee.data$Dates)
```

```
## [1] "1973-01-05" "1995-08-30"
```

```
#plot(c(0,1), c(0,1))
```

2 Week 3:Summary Statistics

Scientists almost always calculate average and standard deviation of the data of interest.
But we don't always learn why. The answer lies in the history of the normal distribution.

In 1809, Carl Friedrich Gauss published a monograph commonly known as *Theoria Motus* (see for example: Theory of Motion of the Heavenly Bodies Moving About the Sun in Conic Sections: A Translation of Theoria Motus, Dover Phoenix Editions, ISBN 0486439062). In it, Gauss derived the probability law of measurement error as a justification for using the least squares method for estimating a mean. This probability law is later known as the normal or Gaussian distribution. Pierre-Simon Laplace published the central limit theorem in 1812, which states that the distribution of sample averages of independent random variables can be approximated by the normal distribution, regardless of the original distribution from which these random

variables were drawn. The closer the original distribution is to normal, the better the approximation will be, particularly with small sample sizes.

In environmental studies, the normal distribution is particularly important because many environmental variables (concentration variables in particular) can be approximated by the log-normal distribution (Ott, 1995). Thus, a rule of thumb in environmental statistics is that we should log-transform concentration variables before statistical analysis (von Belle, 2002), so that properties of normal distributions can be used advantageously. An important result of normal distribution theory is that the "best'' estimator of the normal distribution mean is the sample average. It is the best because it is unbiased, and least variable, and it is also a maximum likelihood estimator (MLE). Consequently, sample averages and standard deviations are commonly reported statistics in scientific studies. In environmental standard assessment, these normal distribution properties helped justify to move to a hypothesis testing approach away from a raw score assessment approach.

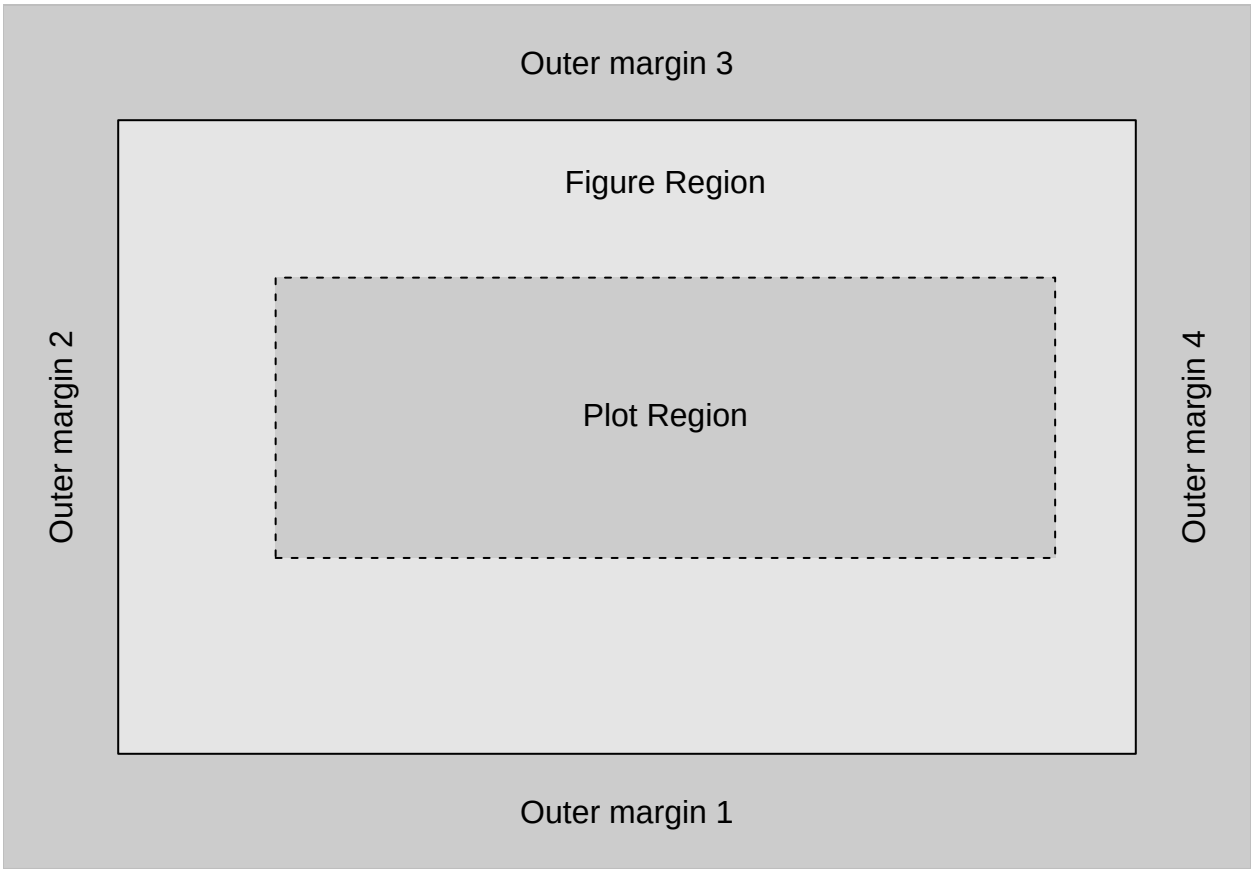
In other words, when we calculate sample average and sample standard deviation, we imply that the data can be approximated by te normal distribution. Whether we know the underlying assumption or not, this practice is common. But if we assume that a concentration variable can be approximated by the log-normal distribution, we should calculate the log mean and log standard deviation.

Which summary statistics to use should be based on the assumption we imposed on the data. To learn about the distribution of the data, we need exploratory data analysis, mostly through graphical display.

2.1 Anatomy of a Plot

A plot consists of a plot region surrounded by margins. Margins include figure margin and outer margin (see K&V for details). The next R code chunk defines the layout of a plot:

```
par(oma=rep(3, 4), bg="grey80")
plot(c(0, 1), c(0, 1), type="n", ann=FALSE, axes=FALSE)
box("outer", col="grey")
# set clipping to figure region
par(xpd=TRUE)
# deliberately draw a stupidly large rectangle
rect(-1, -1, 2, 2, col="grey90")
box("figure")
# set clipping back to plot region
par(xpd=FALSE)
# deliberately draw a stupidly large rectangle
rect(-1, -1, 2, 2, col="grey80")
box("plot", lty="dashed")
text(.5, .5, "Plot Region")
mtext("Figure Region", side=3, line=2)
for (i in 1:4)
    mtext(paste("Outer margin", i), side=i, line=1, outer=TRUE)
```



2.2 Customizing a Plot

We use R function `par` to customize a plot. In the above code chunk, the first line `par(oma=rep(3,4))` sets the outermargin to 3 lines of text on each side. The option `oma` is specified by a vector of 4 numeric values, indicating the outer margin on bottom, left, top, and right. `oma=c(4, 3, 1, 1)` defines outer margins of 4 lines of text at the bottom, 3 lines to the left, 1 line at the top, and 1 line to the right. The outer margin defines the "figure region.''

Inside the figure region, we have a plot region surrounded by margins. Margins of a figure is set by `mar` (margins in lines of texts) or `mai` (in inches). The default margin is `mar=c(5, 4, 4, 2)+0.1`. Again, the margins are specified in the order of bottom, right, top, and left.

By default, R will use the range of the data to set coordinate extremes. But we can also set the coordinate ranges using `usr=c(x.lo,x.hi, y.lo,y.hi)`.

We can also put multiple figures together (e.g., `mfrow=c(3,2)`), specify axes and tick marks. For example, my default setting is:

```
par(mar=c(3,3,1,1), mgp=c(1.25, 0.125, 0), las=1, tck=0.01)
```

where, `mar` specifies margin in lines of texts, `mgp` gives margin lines for the axis title, axis labels, and axis line, `las` defines the style of axis labels (0 – always parallel to the axis, 1 – always horizontal, 2 – always perpendicular to the axis, and 3 – always verticle).

2.3 Graphical Functions

When plotting, we use various graphical functions. These functions use some common options:

- `xlim, ylim` – range of variable plotted on x and y axis
- `pch, col, lty` – plotting character, color, and line type
- `xlab, ylab` – labels of x and y axis
- `main, sub`– main and sub titles

2.4 Example: Lake Erie Harmful Algal Bloom Monitoring

Environmental factors affecting the production of cyanobacterial toxin are explored using two large data sets to support developing strategies for con- trolling and mitigating harmful blooms. Although nutrients, particularly phosphorus, oversupply is the consensus root cause of harmful blooms, factors affecting the production of cyanotoxins are less well studied. Using two large data sets, we analyze the potential factors associated with the variation of microcystin concentrations with an aim of developing a predictive model

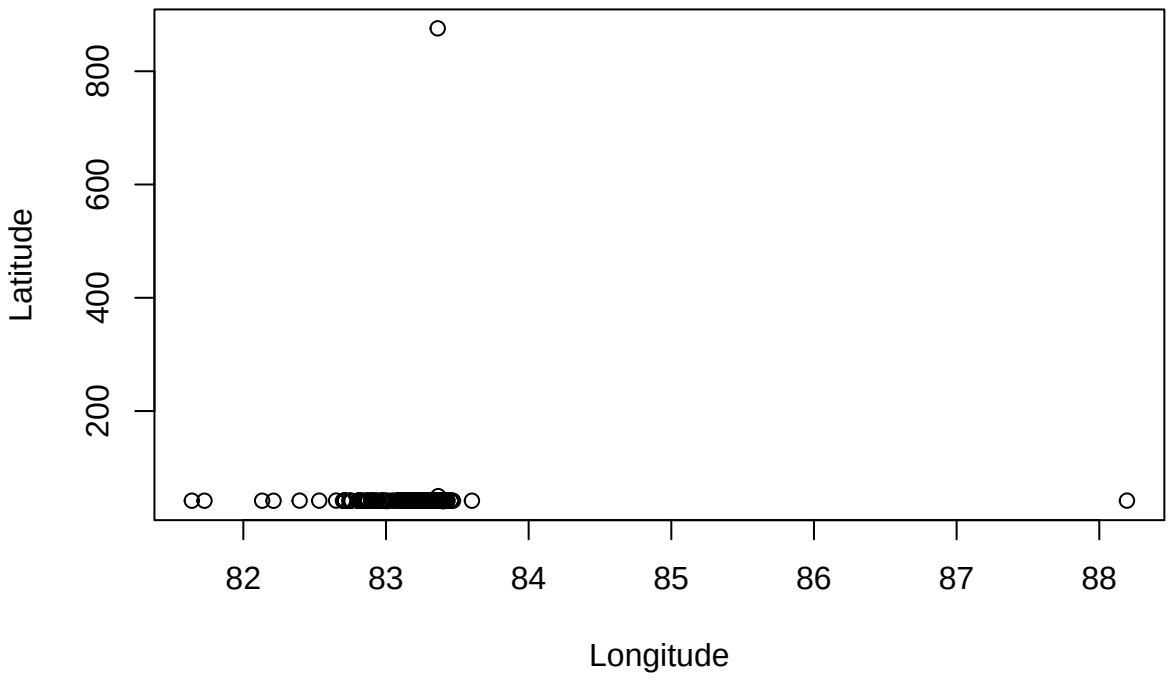
2.5 Data

Reading data and plot sampling sites using R map

```
  ## non-detect (nd, bd, below detect) are replaced with 0
  ## secchi >x is replaced with x (x = 7 or 5)

eriedata <- read.csv(paste(dataDIR, "ErieData.csv", sep="/"), header=T)
#head(eriedata)
```

Data points are labeled by sites (a separate variable). However, because these sampling sites are in the middle of a big lake, each visit to a site is based on a GPS reading. A first step is to see if there are any errors in the locations of sampling sites. First, I will check the latitude and longitude of all sites:

```
  plot(Latitude~Longitude, data=eriedata)
```




```
  ## Five data entry errors:
  ##  1. Latitude for WE8 (10/21/13) was 875.7333 --
  ##     replaced mean latitude of other WE8 with Latitude < 48
  ##  2. Latitude for WE8 (8/13/12) was 49.8369 --
  ##     replaced mean latitude of other WE8 with Latitude < 48
  eriedata$Latitude[eriedata$Latitude>48] <-
      mean(eriedata$Latitude[eriedata$Station=="WE8" &
                                 eriedata$Latitude < 48])
  ##  3. Longitude for WE4 (10/15/14) was 88.1940 -- should be 83.1940?
  eriedata$Longitude[eriedata$Longitude>87] <-
      eriedata$Longitude[eriedata$Longitude>87]-5
  ##  4. Latitude for WE7 (7/6/10) was 40.7649 -- should be 41.6749?
  eriedata$Latitude[eriedata$Latitude<41.1 &
                        eriedata$Station == "WE7"] <- 41.6749
  ##  5. Latitude for WE 2 (5/15/12) was 41.0127 -- 41.7627?
  eriedata$Latitude[eriedata$Latitude<41.1 &
                        eriedata$Station == "WE2"] <- 41.7622
```

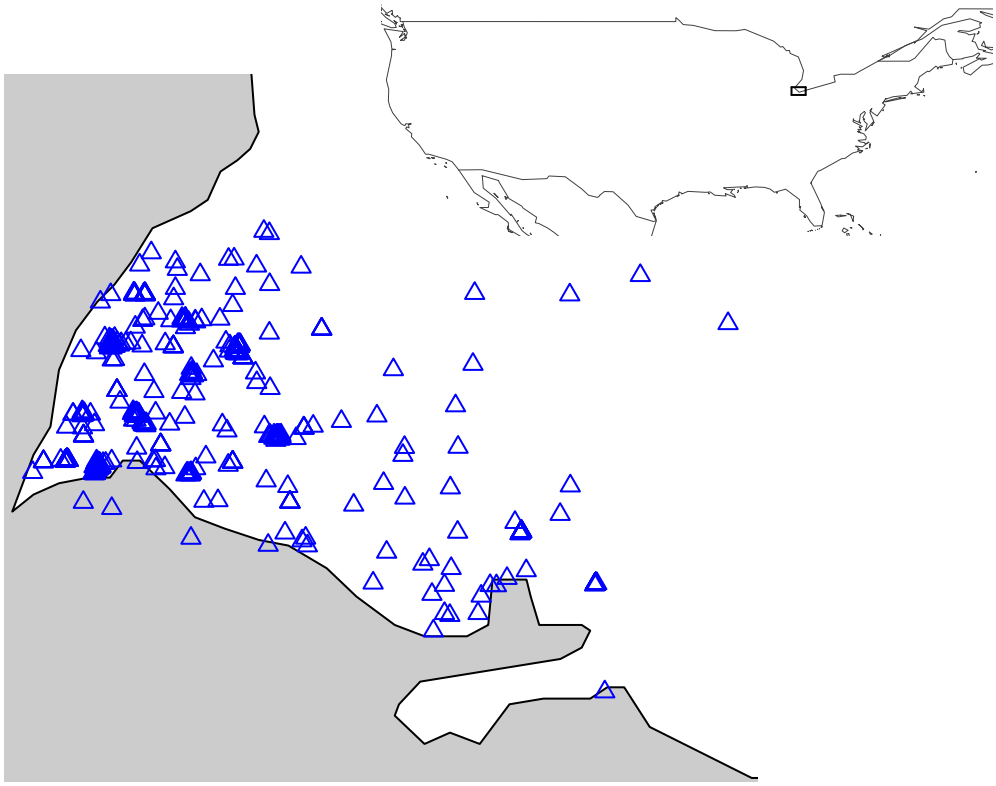
Once location errors are corrected, sampling sites are plotted on a map

```
  eriedata$Longitude <- -eriedata$Longitude
  erieLOC <- eriedata[,c("Latitude","Longitude")]
  coordinates(erieLOC) <- c("Longitude","Latitude")

### using maps:
  my.box<-function(xlim, ylim, ...){
```

```
      segments(x0=xlim, y0=rep(ylim[1],2), x1=xlim, y1=rep(ylim[2], 2), ...)
      segments(y0=ylim, x0=rep(xlim[1],2), y1=ylim, x1=rep(xlim[2], 2), ...)
  }

##tikz(file=paste(plotDIR, "sampleLOC.tex", sep="/"),
##       height=7, width=7,standAlone=F)
  par(mar=rep(0, 4))
  maps::map("usa", fill=TRUE, col="grey80", xlim=c(-83.5,-82.5),
      ylim=c(41.4, 42.1))
  plot(erieLOC, pch=2, col="blue", add=T)

  maplocs <- maps::map(projection="sp_mercator", wrap=TRUE, lwd=0.1,
                col="grey", xlim=c(-180, 0),
                interior=FALSE, orientation=c(90, 180, 0), add=TRUE,
                plot=FALSE)
  xrange <- range(maplocs$x, na.rm=TRUE)
  yrange <- range(maplocs$y, na.rm=TRUE)
  aspect <- abs(diff(yrange))/abs(diff(xrange))
  # customised to 6.5 by 4.5 figure size
  par(fig=c(0.5, 0.99, 0.99 - 0.5*aspect*4.5/6.5, 0.99),
      mar=rep(0, 4), new=TRUE)
  plot.new()
  plot.window(xlim=c(1,2.00),
              ylim=c(0.45,1))
  maps::map(projection="sp_mercator", wrap=TRUE, lwd=0.25, fill=F,
      col=gray(0.25), interior=TRUE, orientation=c(90, 180, 0),
      add=TRUE)
```

```
## Warning in maps::map(projection = "sp_mercator", wrap = TRUE, lwd = 0.25, :
## projection failed for some data
```

```
  my.box(xlim=c(1.7-0.015,1.725-0.015), ylim=c(0.79, 0.81))
```



Summary statistics is almost always a good place to start when examine a data file.

```
#summary(eriedata)
```

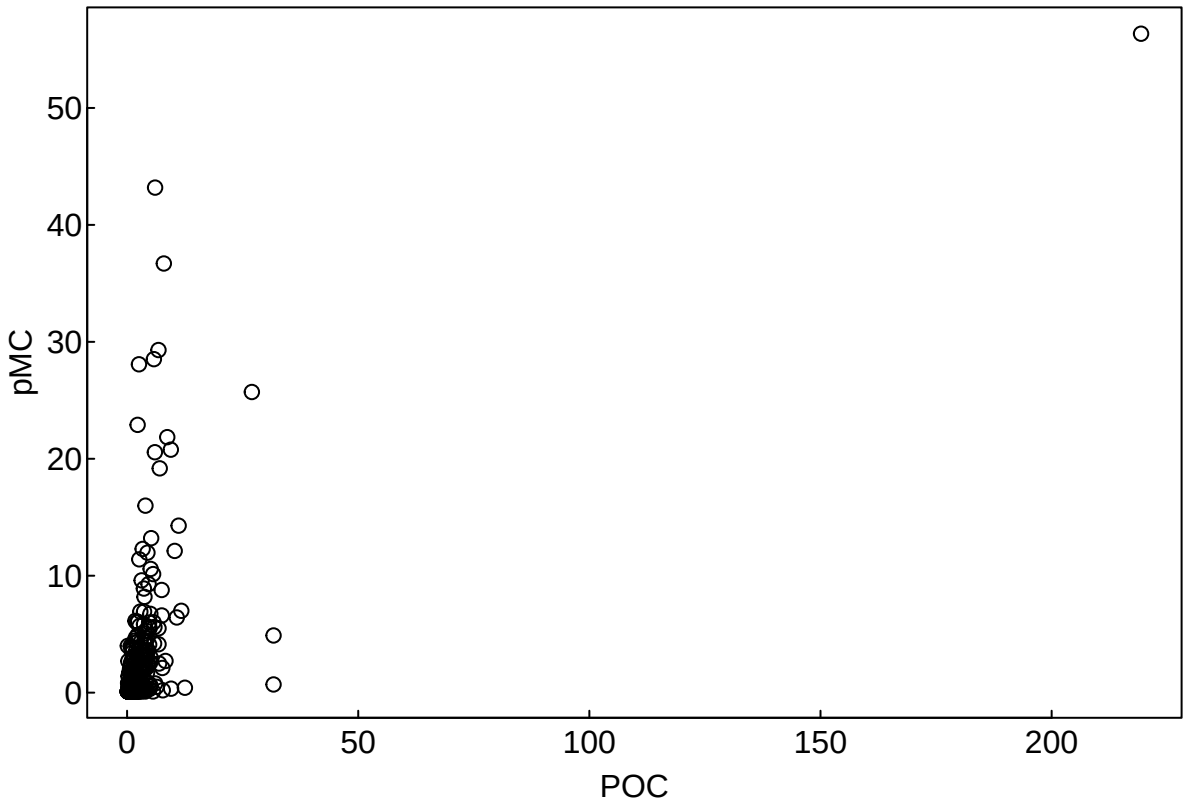
There are a large number of missing values in the data. The variable of interest is the microcystin concentration. We have two forms of MC: particular and dissolved
(`pMC` and `dMC`, respectively). The variable `dMC` has 672 missing values, while `pMC` does not have missing values. However, the method used for measuring MC has a '`detection limit.'' Typically, when the measured concentrations are below the limit, we report them as equal to the detection limit or 0 or half of the detection limit. In this case, we see the minimum`pMC' value is 0, suggesting that 0 is used for values below detection limit. Based on personal communication, I learned that the detection limit for this data set is 0.1.
In other words, the smallest non-zero concentration value should be 0.1.

```
sort(unique(eriedata$pMC[eriedata$pMC>0]))[1:10]
```

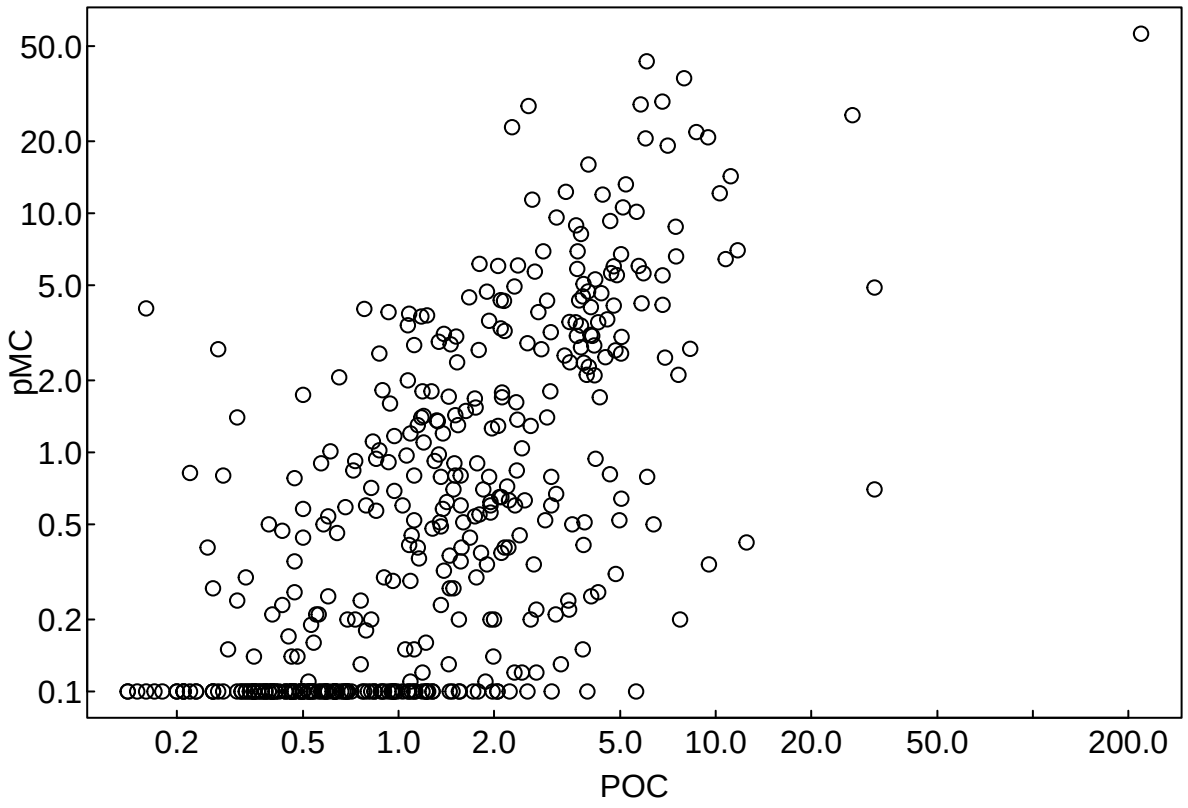
```
##  [1] 0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.10
```

However, the result from this line of code indicates otherwise, suggesting various practices were used over time. After consulting with the person responsible for the monitoring project, I replaced all values less than 0.1 with 0.1. This practice is definitely not ideal.

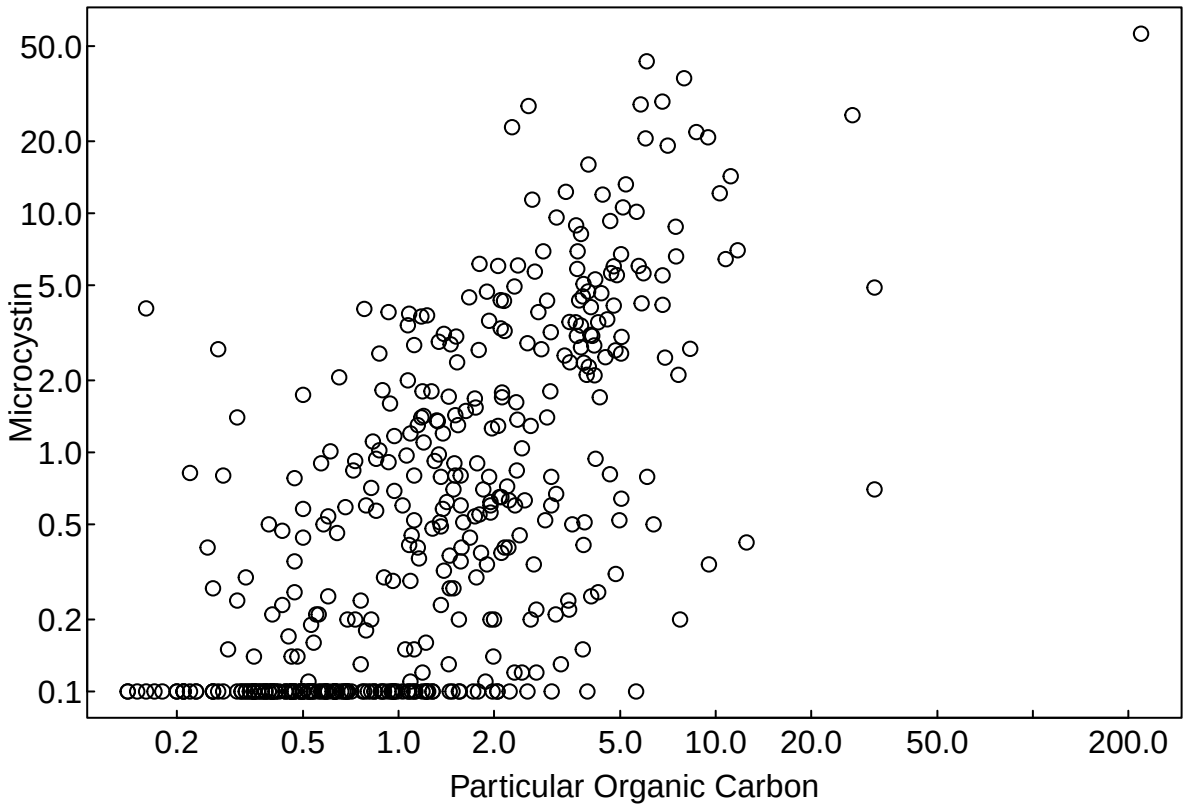
```
eriedata$pMC[eriedata$pMC<0.1] <- 0.1
par(mar=c(3,3,1,1), mgp=c(1.25, 0.125, 0), las=1, tck=0.01)
plot(pMC ~ POC, data=eriedata)
```



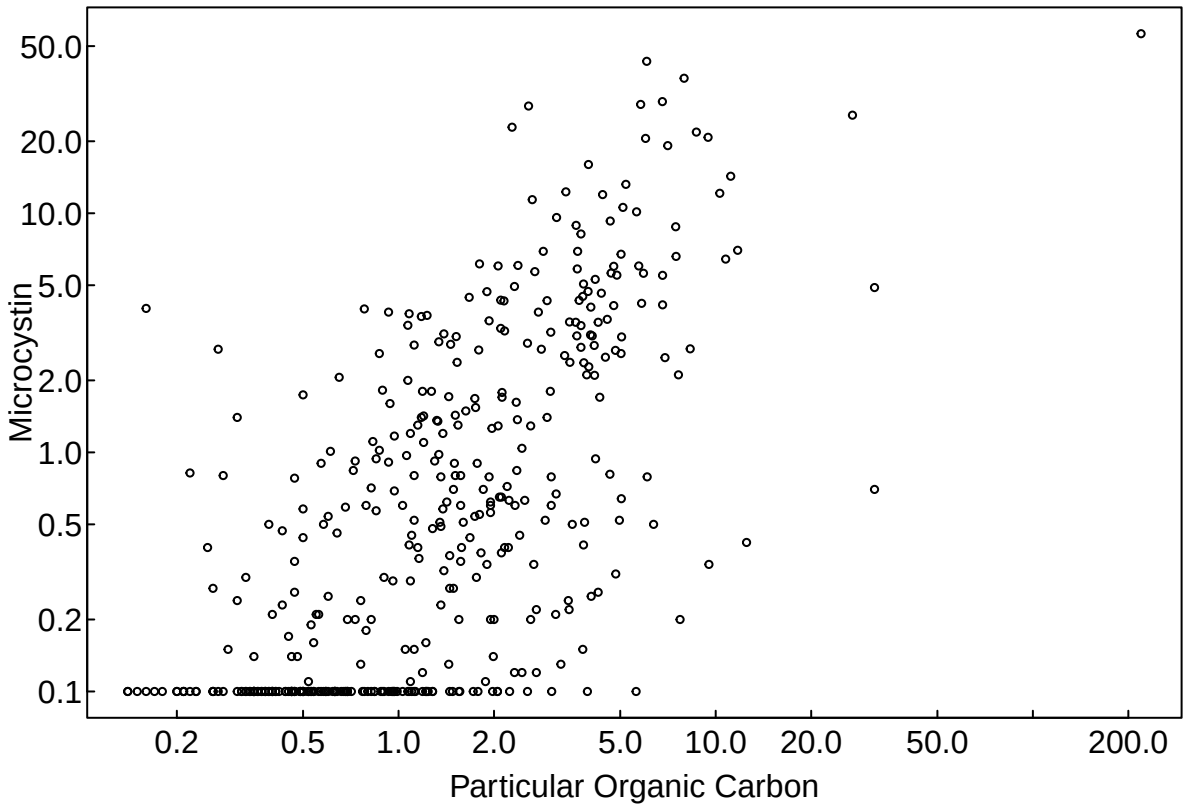

```
plot(pMC ~ POC, data=eriedata, log="xy")
```




```
plot(pMC ~ POC, data=eriedata, log="xy", xlab="Particular Organic Carbon", ylab="Microcystin")
```




```
plot(pMC ~ POC, data=eriedata, log="xy", xlab="Particular Organic Carbon", ylab="Microcystin", pch=1, co
```



2.6 MC Distribution

Knowledge the distribution of a variable is the basis of statistical inference. Statistical inference starts with a statistical distribution assumption on the variable of interest. Consequently, proposing a reasonable distribution assumption is the key to a successful statistical analysis. There are many theoretcal distributions, most of them have complicated probability density functions.

In order to make a reasonable assumption, we need to know how to summarize the distribution of the data. We want to know a few simple facts before we can propose a reasonable distribution assumption. For example, what type of data do we have, categorical or numerical, if categorical, are the categories ordered; whether the variable is limited to be positive (or, more generally, bounded); whether the distribution of a numeric variable is symmetric; and whether the variation of the variable varies. Some of the features do not need statistical analysis; for example, stream flow cannot be negative. Some can be shown using simple graphics.

2.6.1 Histogram and boxplot

Go to page 68 of K & V for histogram

Page 70 for box plot

2.6.2 Comparing to the normal distribution

The normal distribution is the most important distribution in statistics. We often need to evaluate whether the data can be approximated by the normal distribution before we can select an appropriate statistical

method. Assessing normality is typically done using the normal quantile-quantile plot (Normal Q-Q plot).

A normal Q-Q plot is based on the relationship between the standard normal variable ($z \sim N(0,1)$) and a normal distributed variable with mean μ and standard deviation σ ($y \sim N(\mu, \sigma)$):

$$y_q = \mu + \sigma z_q$$

That is, the q quantile of y is a linear function of the q quantile of z. This relationship implies that if we can pair the quantiles of the data we want to evaluate to the quantiles of the standard normal distribution, we can evaluate whether the data is approximately normal by plotting the data quantiles against the standard normal quantiles. The the points on the plot form a straight line, the data have a normal distribution.

When we have a data set, we can calculate the approximate quantile of each data point. For example, in R, the ith ranked data point has a quantile of

$$q_i = \frac{i - 0.5}{n}$$

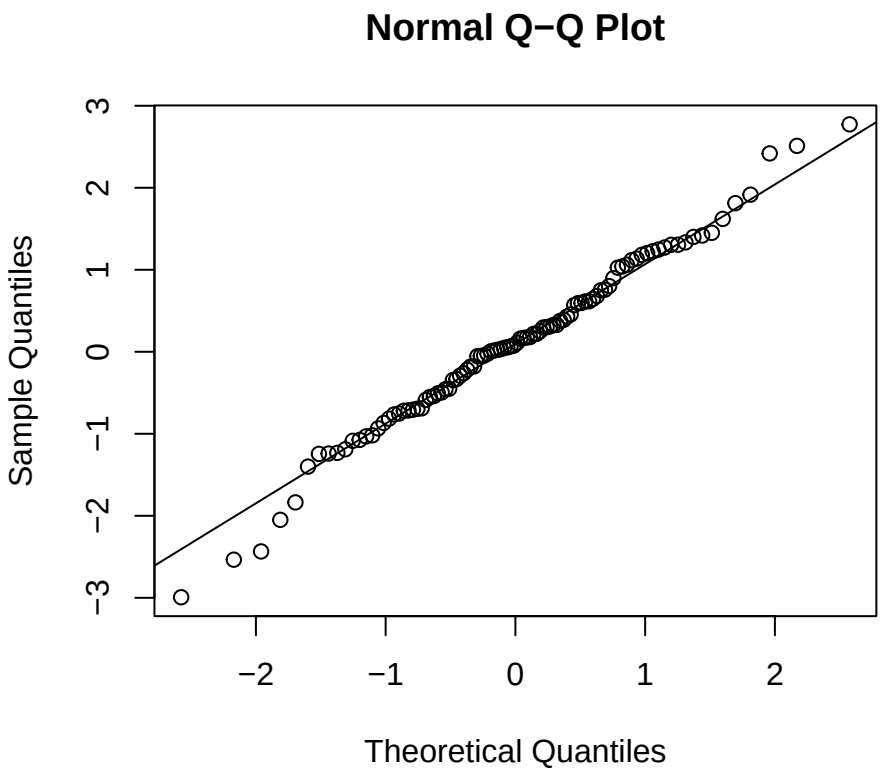
For a data set with $n = 25$, the 5th ranked data point is the 18th percentile of the data. Or, $y_{0.18} = y^{(5)}$. For a unit (standard) normal variable, the 18th percentile can be calculated usibg the function `qnorm`.

```
qnorm(0.18)
```

```
## [1] -0.9153651
```

The value -0.9153651 is paired with $y^{(5)}$. We can do the same for all data points and plot them against their respective unit normal quantiles. The process of finding quantiles is programmed in the functions `qqnorm` and `qqline`:

```
y <- rnorm(100)
qqnorm(y)
qqline(y)
```

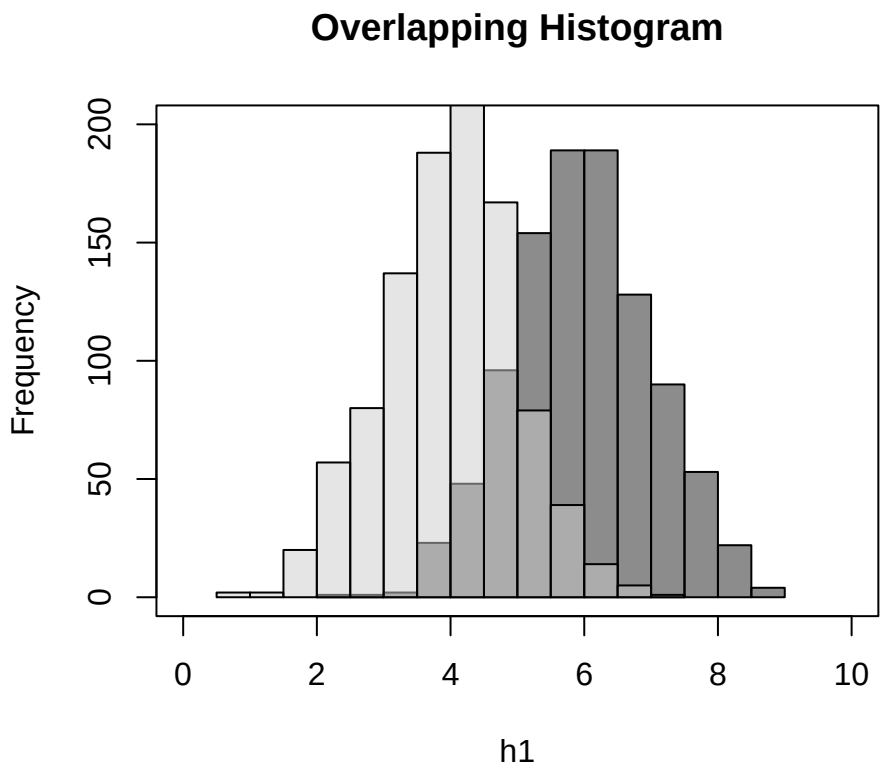


2.6.3 Comparing Two Distributions

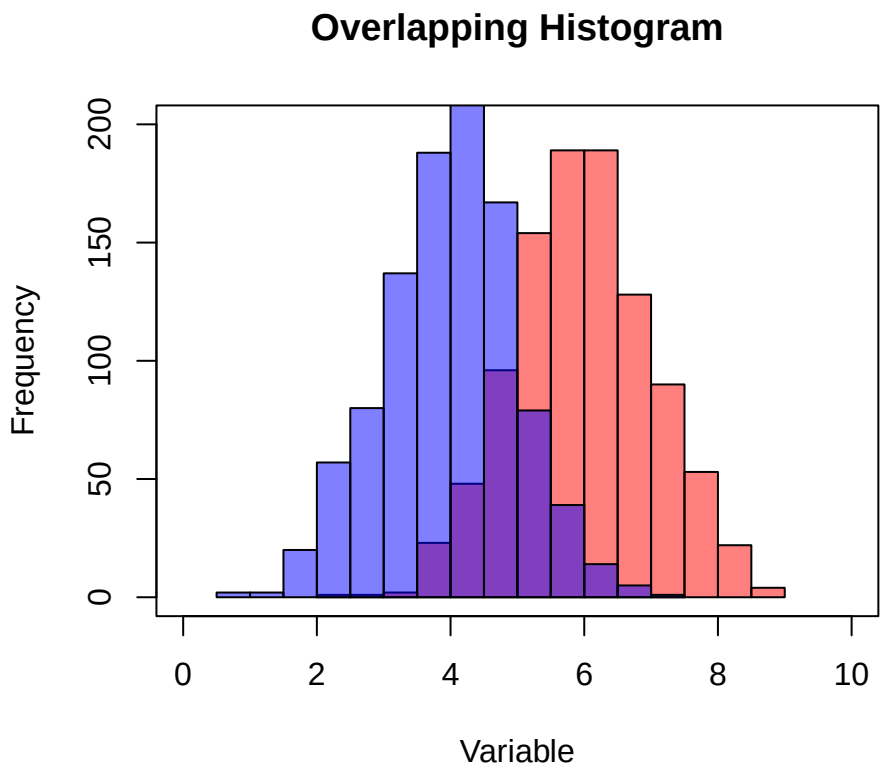
We often are interested in comparing the bloom size/volume in different months.

- Overlying two histograms

```
#Random numbers
h2<-rnorm(1000,4)
h1<-rnorm(1000,6)

# Histogram Grey Color
hist(h1, col=rgb(0.1,0.1,0.1,0.5),xlim=c(0,10), ylim=c(0,200), main="Overlapping Histogram")
hist(h2, col=rgb(0.8,0.8,0.8,0.5), add=T)
box()
```

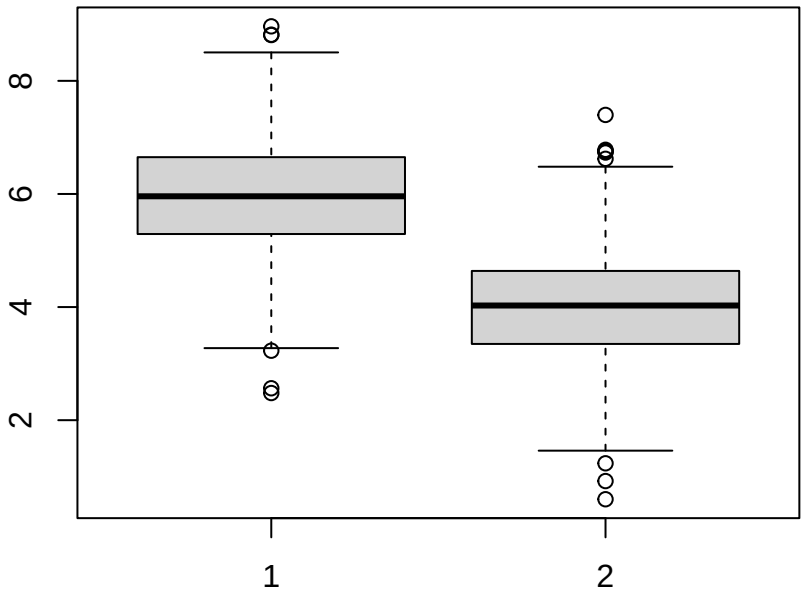



```
# Histogram Colored (blue and red)
hist(h1, col=rgb(1,0,0,0.5),xlim=c(0,10), ylim=c(0,200), main="Overlapping Histogram", xlab="Variable")
hist(h2, col=rgb(0,0,1,0.5), add=T)
box()
```

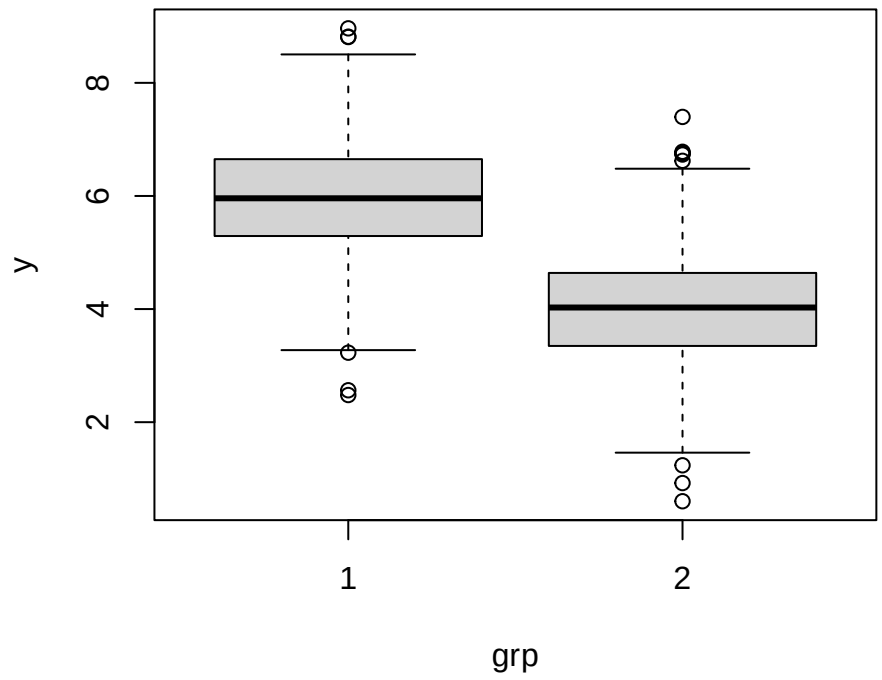


- Side-by-side box plots

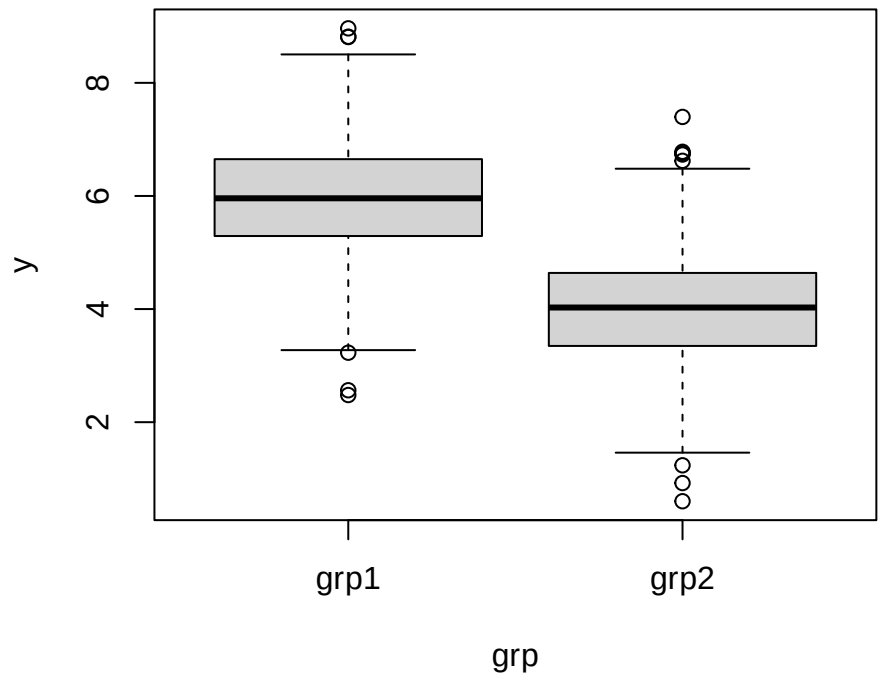
```
boxplot(h1, h2)
```



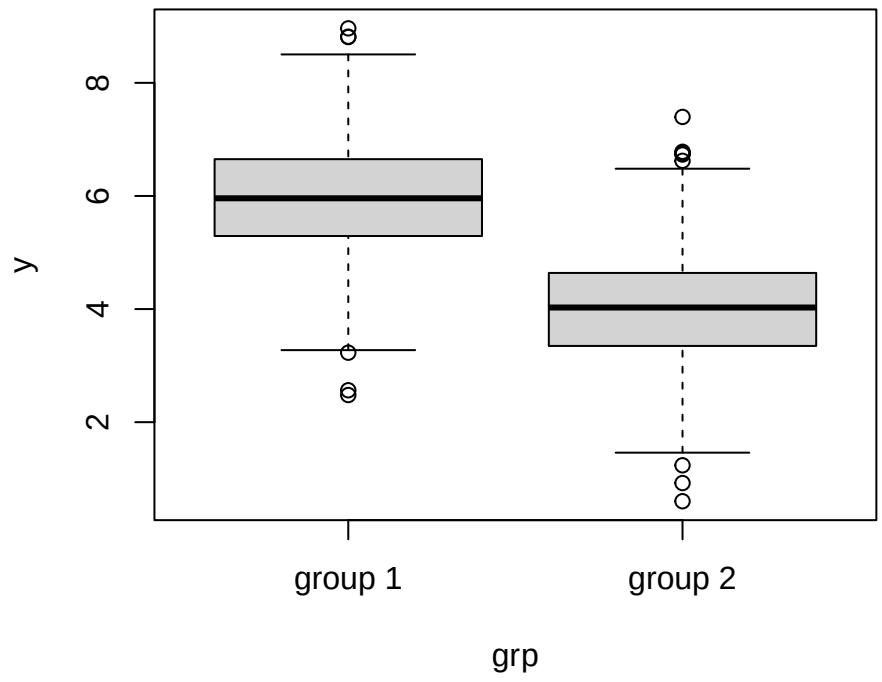

```
## or:
exdata<- data.frame(y=c(h1, h2), grp=rep(c(1,2), each=1000))
boxplot(y~grp, data=exdata)
```




```
## or:
boxplot(y~grp, data=exdata, names=c("grp1","grp2"))
```




```
## now try:
exdata$grp <- ordered(exdata$grp, labels=c("group 1", "group 2"))
boxplot(y~grp, data=exdata)
```



3 Week 3

3.1 Data Management Objectives

- Subsetting data using square brackets
- Using `tapply`
- Using logic comparisons
- Florida is always a problem state, not only in presidential election
- Setting TP criterion for the Everglades – almost a decade long litigation
- Setting nutrient criteria for coastal waters – another long legal battle
- Legal battles between EPA and the State, between the State and environmental groups, between State and industry, between environmental groups and EPA, ...

3.2 Background – EPA's Approach

Criteria for monitoring station inclusion - Stations must have > 4 observations of TN, TP, Chla, and Turbidity - All 0 concentrations observations should be dropped - At least 2 years of data - Annual means must be based on > 4 data points, at least 1 in summer and 1 in winter - Once the data are cleaned, calculate annual geometric means for each station

3.3 Using Preprocessed Data

All existing data were pooled and stations near each other are combined - The resulting data `SouthFlorida.csv` - Data summary

```
##       Year           TN.S
##  Min.   :1989   Min.   :0.0000
##  1st Qu.:1998   1st Qu.:0.1800
##  Median :2001   Median :0.2895
##  Mean   :2001   Mean   :0.3780
##  3rd Qu.:2005   3rd Qu.:0.4898
##  Max.   :2010   Max.   :4.4083
##                 NA's   :724
```

3.4 Data Processing Considerations

- Drop sites with fewer than 4 observations
- Drop sites with fewer than 4 data points for TP, TN, CHLA, or Turbidity
- Drop observations with 0 concentration values for TP, TN, CHLA, or Turb
- Drop sites with less than 2 years of records
- Drop sites without summer or winter observations

3.5 Data for Analysis

After processing, calculating annual geometric means of TP, TN, CHLA, and Turb

R Coding Considerations

- Dates – summer and winter
- Counting by site
- Calculating log-means for by site-year combination
- EPA used function `tapply`

```
#tapply(Data, INDEX, FUN, ..., simplify=TRUE)
```

3.6 tapply

3.6.1 Step 1: Dropping sites with fewer than 4 observations (rows)

- Find sites with fewer than 4 TP samples

```
site.drop <- tapply(Data$TP.S, Data$Site, FUN=function(x) sum(!is.na(x))<4)
```

- The function used in `tapply` is `sum(!is.na(x)) < 4`: returns a logic value of whether the total number of non-missing is less than 4
- The results is a vector of `TRUE` and `FALSE`, each is associated with a site name

```
sites.drop <- tapply(Data$TP.S, Data$Site,
      function(x) sum(!is.na(x)) < 4) |
              tapply(Data$TN.S, Data$Site,
      function(x) sum(!is.na(x))<4)   |
              tapply(Data$TURB.S, Data$Site,
      function(x) sum(!is.na(x))<4)   |
              tapply(Data$CHLA, Data$Site,
      function(x) sum(!is.na(x))<4)
sites.drop <- names(sites.drop)[sites.drop]
if(length(sites.drop)>0)
    Data <- Data[!is.element(Data$Site, sites.drop),]
```

3.6.2 Step 2: Dropping 0 concentration values

```
Data$TP.S[Data$TP.S==0] <- NA
Data$TN.S[Data$TN.S==0] <- NA
Data$TURB.S[Data$TURB.S==0] <- NA
Data$CHLA[Data$CHLA==0] <- NA
```

3.6.3 Step 3: Dropping Years with Fewer Than 4 Observations

```
site.yr <- paste(Data$Site, Data$Year)
siteyr.drop <-
  tapply(Data$CHLA, site.yr,
        function(x) sum(!is.na(x))<4) |
  tapply(Data$TURB.S, site.yr,
        function(x) sum(!is.na(x))<4) |
  tapply(Data$TP.S, site.yr,
        function(x) sum(!is.na(x))<4) |
  tapply(Data$TN.S, site.yr,
        function(x) sum(!is.na(x))<4)
siteyr.drop <-
  names(siteyr.drop)[siteyr.drop]
if(length(siteyr.drop)>0)
  Data <- Data[!is.element(site.yr,
                      siteyr.drop),]
```

3.6.4 Step 4: At Least One Obs in Summer and One in Winter

Define summer and winter using dates

- Convert date column into a `Date` object

```
Data$dateR <- as.Date(Data$DateT)
```

- Extract month:

```
Data$Month <- ordered(months(Data$dateR),
                      levels=month.name)
Data$Quarter <- quarters(Data$dateR)
tmp <- as.numeric(Data$Month)
Data$Summer <- 0
Data$Summer[tmp>=5 & tmp<=9] <- 1
## EPA forgot to code winter
```

Use `tapply` in Step 4

```
siteyr.summ <- paste(Data$Site, Data$Year, Data$Summer)
siteyrsum.drop <-
  tapply(Data$CHLA, siteyr.summ,
         function(x) sum(!is.na(x))==0) |
  tapply(Data$TURB.S, siteyr.summ,
         function(x) sum(!is.na(x))==0) |
  tapply(Data$TP.S, siteyr.summ,
         function(x) sum(!is.na(x))==0) |
  tapply(Data$TN.S, siteyr.summ,
         function(x) sum(!is.na(x))==0)
siteyrsum.drop <- names(siteyrsum.drop)[siteyrsum.drop]

if (length(siteyrsum.drop)>0)
  Data <- Data[!is.element(grpyr.summ, grpyrsum.drop),]
```

Step 5: Calculating Annual Geometric Means

- Annumal geometric means for each site

```
#tapply(log(Data$TP.S), Data$site.yr, mean, na.rm=T)
```

- The result is put in a data frame named `TP.log`, along with site annual means of TN, CHLA, Turb, and other parameters

3.7 Concluding Remarks

- Order of steps matters
- Several pages of code are difficult to debug
- Statistical concerns: geometric means are based on different sample sizes

4 Week 5

This is a RMarkdown document on R data objects for the week of September 24, 2019.

4.1 Subscripting

For objects with more than one elements, we use subscripting to access some elements.

4.1.1 Numerical Subscripts

We use numerical subscripts to access the elements of a vector, array, or list. The first element has subscript 1, the second has 2, and so on. Subscript 0 is ignored (without warning).
Negative subscripts indicate the ones to be left off – extract all but the ones indicated with a negative sign. However, numerical subscripts must be all positive (including 0), or all negative (including 0). Mixing negative and positive subscripts will result in error.

We can use a sequence of integers expressed by `c()`, `seq()`, and the colon operator.

4.1.2 Character Subscripts

When an object is named, a character string or a vector of character string can be as subscript to extract the named element(s). Negative character subscripts are not permitted.
To exclude elements based on their names we use function `grep`.
A call to any function that returns a character string (vector) can be used as a subscript.

4.1.3 Logical Subscripts

When using logical values (`TURE`, `FALSE`) to selectively access elements of an object, we use a logical object with the same size of the object (or part of the object) that is being subscripted. Elements corresponding to `TRUE` values in the logical object will be included, and elements corresponding to `FALSE` values will be excluded.

Logical subscripts are probably the most useful way to perform complicated tasks. To understand the use of logical subscripts, we should take a look at the logic objects. For example, suppose we are interested in the subset of large values (> 10) of the following object:

```
nums <- c(12, 9, 8, 14, 7, 16, 3, 2, 9)
```

A simple logical expression can be used to compare each of the elements to the number 10:

```
nums > 10
```

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

The logical operator `>` is vectorised – applying a logical operator to a vector will result in a vector of logical values.

```
length(nums)
```

```
## [1] 9
```

```
length(nums > 10)
```

```
## [1] 9
```

When using the resulting logical vector for subscripting, it will extract the elements for which the logical vector is true:

```
nums[nums > 10]
```

```
## [1] 12 14 16
```

Sometimes, we may want to known the indices of these elements.
The function `which` accepts a logic vector and returns a vector containing the subscripts of the elements for which the logical vector was true:

```
which(nums > 10)
```

```
## [1] 1 4 6
```

This operation is equivelent to:

```
seq(along=nums)[nums > 10]
```

```
## [1] 1 4 6
```

or

```
(1:length(nums))[nums>10]
```

```
## [1] 1 4 6
```

Logical subscripts allow for modification of elements that meet a particular condition by using an appropriately subscripted object on the left-hand side of an assignment statement. Suppose that values larger than 10 is impossible, any observation with a value of above 10 should be removes (or replaced by `NA`), we can use the logical subscripts:

```
nums[nums > 10] <- NA
nums
```

```
## [1] NA  9  8 NA  7 NA  3  2  9
```

4.1.4 Subscripting Matrices

When an object has more than 1 dimensions, subscripts can be provided for each dimension separated by a comma (or commas). For a multidimensional object, empty subscript on one dimension means that all elements of that dimension will be extracted. For example, while `x[3,4]` extracts the element in 3rd row and 4th column, `x[3,]` extracts the entire 3rd row.

```
x <- matrix(1:12, 3, 4)
x
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
x[3,4]
```

```
## [1] 12
```

```
x[3,]
```

```
## [1]  3  6  9 12
```

```
x[,c(4, 1)]
```

```
##      [,1] [,2]
## [1,]   10    1
## [2,]   11    2
## [3,]   12    3
```

When a matrix is subscripted with a single subscript, the matrix is treated (silently) as a vector composed of all the columns of the matrix. This may be useful in certain situations, but can be confusing.

```
x[10]<-NA
```

Here is an example where a single script is convenient. In missing value problems, we often want to impute missing values, either missing at random or values are known to be less than certain thrshold (e.g., method reporting limit). Numerically, these missing values are treated as unknown quantities to be estimated. Almost all numerical algarithms require a set of starting (or initial) values. I wrote a function to supply such initial values:

```
ini <- function (x, inival){
  nas <- is.na(x)
  x[nas] <- inival
  x[!nas] <- NA
  return(x)
}
```

This function can take both vectors and matrices as `x`.

```
x[10] <- NA
ini(x, 0)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   NA   NA   NA    0
## [2,]   NA   NA   NA   NA
## [3,]   NA   NA   NA   NA
```

```
y <- 1:12
y[10] <- NA

ini(y, 0.5)
```

```
##  [1]  NA  NA  NA  NA  NA  NA  NA  NA  NA 0.5  NA  NA
```

Sorting a matrix by a particular row or column is most conveniently done through subscripting. Let's use the data `stack.x` as an example.
The object includes operational data of a plant that makes nitric acid by oxidation of ammonia. It is a matrix of three columns (`"Air.Flow","Water.Temp","Acid.Conc"`). We can sort the matrix by the column `Air.Flow`:

```
stack.x.sorted <- stack.x[order(stack.x[,"Air.Flow"]), ]
head(stack.x.sorted)
```

```
##      Air.Flow Water.Temp Acid.Conc.
## [1,]       50         18         89
## [2,]       50         18         86
## [3,]       50         19         72
## [4,]       50         19         79
## [5,]       50         20         80
## [6,]       56         20         82
```

A note on `ordered`:

```
temp <- rnorm(10)
temp
```

```
##  [1]  0.1153075 -0.4762677 -1.1147394 -0.3373574 -0.5627122 -0.5948542
##  [7]  0.4271315 -0.3892567  1.7663537  1.9030413
```

```
order(temp)
```

```
##  [1]  3  6  5  2  8  4  1  7  9 10
```

Now we have sorted the data frame by `Air.Flow`. What if you want to sort it first by `Air.Flow`, then by `Water.Temp` and then by `Acid.Conc.`?

```
stack.x.sorted <-
  stack.x[order(stack.x[,1], stack.x[,2], stack.x[,3]), ]
```

Let's look at a data frame. The famous iris data was used by Fisher and Anderson to illustrate ANOVA. The data set consists of measurements in centimeters of the variables sepal length and width and petal length and width for 50 flowers from each of three species of iris. The species are *iris setosa, versicolor* and *virginica*. We can sort the data frame by columns, and use the function `do.call` to pass the column names:

```
sortframe <- function(df, ...) df[do.call(order, list(...)),]

#with(iris,
#sortframe(iris, Sepal.Width, Sepal.Length, Petal.Width, Petal.Length))
```

The use of `order` results in ascending order. We may want to sort in descending order. The order can be reversed by using `rev()`:

```
stack.x.sorted <- stack.x[rev(order(stack.x[,"Air.Flow"])), ]
head(stack.x.sorted)
```

```
##      Air.Flow Water.Temp Acid.Conc.
## [1,]       80         27         88
## [2,]       80         27         89
## [3,]       75         25         90
## [4,]       70         20         91
## [5,]       62         24         93
## [6,]       62         24         93
```

We can rewrite the function `sortframe` to sort in descending order:

```
sortframeD <- function(df, ...) df[rev(do.call(order, list(...))), ]
iris.sorted <- with(iris,
sortframeD(iris, Sepal.Width, Sepal.Length, Petal.Width, Petal.Length))
```

`drop=FALSE`

When extracting one row or one column from a matrix, the resulting object becomes a vector (reduced dimension by 1). In some cases we want to maintain the original data structure (keep a matrix a matrix).

```
stack.x[1,]
```

```
##   Air.Flow Water.Temp Acid.Conc.
##         80         27         89
```

```
stack.x[1,,drop=FALSE]
```

```
##      Air.Flow Water.Temp Acid.Conc.
## [1,]       80         27         89
```

When applied to a data frame, extracting a column may result in losing the name. Using `drop=FALSE` allows to keep the name:

```
iris[,2,drop=FALSE]
```

Using subscripts, it is easy to selectively access any combination of rows and/or columns that you need. For example, we can select all the columns of the matrix `x` for which the first column is less larger than 1:

```
x[,1] > 1
```

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

```
x[x[,1]>1, ]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    5    8   11
## [2,]    3    6    9   12
```

In a data frame, we can extract rows representing a specific site, date, or species:

```
setosa <- iris[iris$Species == "setosa", 1:4]
```

4.2 Two Useful Functions for Matrix

The functions `row` and `col` can be used to help subscripting matrices. By themselves, the functions are not very useful.
They are very helpful in handling squared matrices when carrying out matrix operations.

```
col(x)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    1    2    3    4
## [3,]    1    2    3    4
```

```
row(x)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    1    1
## [2,]    2    2    2    2
## [3,]    3    3    3    3
```

For a squared matrix:

```
z <- matrix(1:16, 4, 4)
rw <- row(z)
cl <- col(z)

offd <- rw != cl
```

We now have off-diagnal and diagnal elements:

```
z[offd]
```

```
##  [1]  2  3  4  5  7  8  9 10 12 13 14 15
```

```
z[!offd]
```

```
## [1]  1  6 11 16
```

4.3 List

List is the most general way to store a collection of objects in R. There is no limitation on the mode and size of the objects. An unstated rule of sub-scripting is that the returned object is of the same mode as the object being sub-scripted. It is natural for vectors and matrix. But for a list, each part of the list can be of different mode. For example,

```
list.example <- list (a = c('fred','sam','harry'), b=rnorm(10))
mode(list.example)
```

```
## [1] "list"
```

```
list.example[2]
```

```
## $b
##  [1]  0.87478503 -0.54975391 -0.37003241 -1.16065394  1.72065519  0.50567464
##  [7]  0.09295471  0.85239561  0.18103029 -0.05487497
```

```
mode(list.example[2])
```

```
## [1] "list"
```

Although the second element of the list is numeric, R took it as `list`:

```
mean(list.example[2])
```

```
## Warning in mean.default(list.example[2]): argument is not numeric or logical:
## returning NA
```

```
## [1] NA
```

That is, the result is a list of one element.

R provides two ways to resolve this issue. First, if the elements of a list are named, the actual elements can be accessed by separating the name of the list and the name of the element with a dollar sign ($):

```
mean(list.example$b)
```

```
## [1] 0.209218
```

When using the dollar sign is inappropriate (e.g., unnamed elements), R allows us to use the double bracket subscript operator. The double bracket operator will extract the actual list element:

```
mean(list.example[[2]])
```

```
## [1] 0.209218
```

```
mean(list.example[["b"]])
```

```
## [1] 0.209218
```

When using a list, single bracket will return a list, double bracket will return an object with the mode of the extracted element.

```
list.example[1]
```

```
## $a
## [1] "fred"  "sam"   "harry"
```

```
list.example[[1]]
```

```
## [1] "fred"  "sam"   "harry"
```

It is fine to use `list.example[1:2]`, or `list.example["a","b"]`, but `list.example[[1:2]]` will return something unexpected.

4.4 Aggregation

R has many functions for data aggregation. For simple tabulation and cross-tabulation, the function `table` is adequate.

For more complext tasks, there are two groups of functions. For array data (vectors, matrices, arrays), we can use `apply, sweep, mapply, sapply`, and `lapply`. For data frame, we use `aggregate`, `by` (which is a wrapper for `tapply`).

4.4.1 Tabulation

When the function `table` is used on a vector, it returns a named vector of counts of unique values in the vector:

```
pets <- c('gold fish', 'dog','cat', 'duck', 'gold fish', 'chicken','duck','cat','dog')
tbl <- table(pets)
tbl
```

```
## pets
##       cat   chicken       dog      duck gold fish
##         2         1         2         2         2
```

The output can be converted into a data frame:

```
as.data.frame(tbl)
```

```
##        pets Freq
## 1       cat    2
## 2   chicken    1
## 3       dog    2
## 4      duck    2
## 5 gold fish    2
```

When `table` is applied to multiple vectors, it will return a cross-tabulation. For example, a two vector situation:

```
hiinc <- state.x77[,'Income'] > median(state.x77[,'Income'])
stateinc <- table(state.region, hiinc)
stateinc
```

```
##                hiinc
## state.region    FALSE TRUE
##   Northeast         4    5
##   South            12    4
##   North Central     5    7
##   West              4    9
```

```
as.data.frame(stateinc)
```

```
##    state.region hiinc Freq
## 1     Northeast FALSE    4
## 2         South FALSE   12
## 3 North Central FALSE    5
## 4          West FALSE    4
## 5     Northeast  TRUE    5
## 6         South  TRUE    4
## 7 North Central  TRUE    7
## 8          West  TRUE    9
```

When we pass a data frame to `table`, it treats each column as a separate variable:

```
x <- data.frame(a=c(1,2,2,1,2,2,1),
                b=c(1,2,2,1,1,2,1),
                c=c(1,1,2,1,2,2,1))
x
```

```
##   a b c
## 1 1 1 1
## 2 2 2 1
## 3 2 2 2
## 4 1 1 1
## 5 2 1 2
## 6 2 2 2
## 7 1 1 1
```

```
table(x)
```

```
## , , c = 1
##
##    b
## a   1 2
##   1 3 0
##   2 0 1
##
## , , c = 2
##
##    b
```

```
## a   1 2
##   1 0 0
##   2 1 2
```

```
as.data.frame(table(x))
```

```
##   a b c Freq
## 1 1 1 1    3
## 2 2 1 1    0
## 3 1 2 1    0
## 4 2 2 1    1
## 5 1 1 2    0
## 6 2 1 2    1
## 7 1 2 2    0
## 8 2 2 2    2
```

The data frame output is especially useful in analyzing data with multiple factors. The data frame output summarizes the frequency of each unique combination.

With a two way cross-tabulation, we can also add margins of summary statistics. Let's use the data `infert`.

```
temp <- table(infert$education, infert$parity)
temp
```

```
##
##            1  2  3  4  5  6
##   0-5yrs   3  0  0  3  0  6
##   6-11yrs 42 42 21 12  3  0
##   12+ yrs 54 39 15  3  3  2
```

To add a row of margins:

```
temp1 <- addmargins(temp, 1)
```

The default summary statistics is `sum`. To change the statistics:

```
temp2 <- addmargins(temp, 1, FUN=mean)
```

To add a column of margins:

```
temp3 <- addmargins(temp, 2)
```

To add both column and row of margins:

```
temp4 <- addmargins(temp, c(1,2))
```

Some times, we are interested in having a table of proportions instead of counts. We can use the function `sweep`or `prop.table`, which accepts a table and a margin:

```
prop.table(temp,2)
```

```
##
##                    1          2          3          4          5          6
##   0-5yrs  0.03030303 0.00000000 0.00000000 0.16666667 0.00000000 0.75000000
##   6-11yrs 0.42424242 0.51851852 0.58333333 0.66666667 0.50000000 0.00000000
##   12+ yrs 0.54545455 0.48148148 0.41666667 0.16666667 0.50000000 0.25000000
```

Similar to `table`, `xtabs` also produces cross-tabulations but uses formula:

```
xtabs(~state.region+ hiinc)
```

```
##                hiinc
## state.region    FALSE TRUE
##   Northeast         4    5
##   South            12    4
##   North Central     5    7
##   West              4    9
```

If a variable is given on the left-hand side of the tilde (∼), it is interpreted as a vector of counts corresponding to the values of the variables on the right-hand side, making it very convenient to convert tabulated data into a table:

```
x
```

```
##   a b c
## 1 1 1 1
## 2 2 2 1
## 3 2 2 2
## 4 1 1 1
## 5 2 1 2
## 6 2 2 2
## 7 1 1 1
```

```
dfx <- as.data.frame(table(x))
xtabs(Freq~a+b+c, data=dfx)
```

```
## , , c = 1
##
##    b
## a   1 2
##   1 3 0
##   2 0 1
##
## , , c = 2
##
##    b
## a   1 2
##   1 0 0
##   2 1 2
```

4.4.2 General Considerations of Aggregation

- How are the groups that divide the data are defined? (list, column/row of a matrix, a or more variable of a data frame)
- Waht is the nature of the data to be operated on?
- What is the desired end result?

Groups defined by a list element. We use `sapply` or `lapply`. `lapply` always retuns a list, `sapply` may simplify its output into vectors or matrix when appropriate.

Groups defined by a row or column of a matrix. `apply` should be used.`apply` returns a vector or matrix (array).

Groups defined by one or more grouping variables. Which function to use depends on what outcome do you want.

If we are interested in calculating scalar statistics (e.g., mean, variance), we often use `aggregate` because it reurns a data frame. If the result is a vector, `tapply` is often used.

sapply`We use the data`ChickWeight‘ as an example.

```
?ChickWeight
```

```
## starting httpd help server ... done
```

```
names(ChickWeight)
```

```
## [1] "weight" "Time"   "Chick"  "Diet"
```

```
sapply(ChickWeight, class)
```

```
## $weight
## [1] "numeric"
##
## $Time
## [1] "numeric"
##
## $Chick
## [1] "ordered" "factor"
##
## $Diet
## [1] "factor"
```

Here, we have a data frame (a list) of 4 elements. Using `sapply`, we apply the same function (`class`) to each of the element.

`apply The function`apply‘ is used for aggregating a matrix by row or by column. It can be used more generally for an array (more than two dimensions).

```
apply(state.x77, 2, sum)
```

```
## Population     Income Illiteracy   Life Exp     Murder    HS Grad      Frost
##  212321.00  221790.00      58.50    3543.93     368.90    2655.40    5223.00
##       Area
## 3536794.00
```

When the function returns more than one scalar value, the output will be a matrix:

```
apply(state.x77, 2, summary)
```

```
##         Population  Income Illiteracy Life Exp Murder HS Grad  Frost      Area
## Min.        365.00 3098.00      0.500  67.9600  1.400  37.800   0.00   1049.00
## 1st Qu.    1079.50 3992.75      0.625  70.1175  4.350  48.050  66.25  36985.25
## Median     2838.50 4519.00      0.950  70.6750  6.850  53.250 114.50  54277.00
## Mean       4246.42 4435.80      1.170  70.8786  7.378  53.108 104.46  70735.88
## 3rd Qu.    4968.50 4813.50      1.575  71.8925 10.675  59.150 139.75  81162.50
## Max.      21198.00 6315.00      2.800  73.6000 15.100  67.300 188.00 566432.00
```

We can provide functions for calculating specific statistics:

```
sumfun<- function(x)
  c(n=sum(!is.na(x)), mean=mean(x, na.rm=T), sd=sd(x, na.rm=T))
apply(state.x77, 2, sumfun)
```

```
##      Population    Income Illiteracy  Life Exp   Murder   HS Grad     Frost
## n        50.000   50.0000 50.0000000 50.000000 50.00000 50.000000  50.00000
## mean   4246.420 4435.8000  1.1700000 70.878600  7.37800 53.108000 104.46000
## sd     4464.491  614.4699  0.6095331  1.342394  3.69154  8.076998  51.98085
##          Area
## n       50.00
## mean 70735.88
## sd   85327.30
```

The function we wrote takes a vector (`x`) as input. If a function takes additional input variables (e.g., the function `mean` has additional arguments such as `na.rm`), these additional inputs are entered after the function argument.

```
apply(state.x77, 2, mean, na.rm=T, trim=0.25)
```

```
##   Population       Income   Illiteracy     Life Exp       Murder      HS Grad
##  2994.576923  4462.038462     1.019231    70.886538     7.342308    54.065385
##        Frost         Area
##   109.807692 56021.961538
```

Additional uses:

```
maxes <- apply(state.x77, 2, max)
#sweep(state.x77, 2, maxes, "/")
```

Suppose that we are interested in calculating the mean for each variable in `state.x77`, using only those values which are larger than the median for that variable.

```
meds <- apply(state.x77, 2, median)
meanmed <- function(x, med) return(mean(x[x>med]))
meanmed(state.x77[,2], meds[2])
```

```
## [1] 4917.92
```

When using `apply`, we apply a function to multiple variables. If there are additional arguments (e.g., `na.rm=T` in `mean`), the additional arguments are the same to all variables. In this function, we also apply one function to several variables, but the additional argument (`meds`) has different values for different variables. In situations like this, we use `mapply`. Because `mapply` is written for list, the matrix `state.x77` must be converted into a data frame:

```
mapply(meanmed, as.data.frame(state.x77), meds)
```

```
## Population     Income Illiteracy   Life Exp     Murder    HS Grad      Frost
##   7136.160   4917.920      1.660     71.950     10.544     59.524    146.840
##       Area
## 112213.400
```

4.4.3 Mapping a Function Based on Groups

Frequently, groups are defined by mulitple variables (e.g., see `ChickWeight` and `iris`). If we are interested in calculating a scalar summary statistics, we can use `aggregate`

```
aggregate(iris[-5], iris[5], mean)
```

```
##      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa        5.006       3.428        1.462       0.246
## 2 versicolor        5.936       2.770        4.260       1.326
## 3  virginica        6.588       2.974        5.552       2.026
```

```
#aggregate(ChickWeight$weight, ChickWeight[c('Time','Diet')], mean)
```

(Read `?aggregate` and find out why `aggregate(iris[,-5], iris[,5], mean)` won't work.)

(Note the difference in output format.)

When groups are defined by one vector, we use `tapply`.

```
sepalL <- tapply(iris$Sepal.Length, iris$Species, mean)
sepalL.range <- tapply(iris$Sepal.Length, iris$Species,range)
```

The object `sepalL.range` is a list of three elements. It is often convenient to convert it to a data frame:

```
sepalL.range<-data.frame(group=dimnames(sepalL.range)[[1]],
                         matrix(unlist(sepalL.range), ncol=2, byrow=T))
```

We can also use `tapply` with more than one grouping varaibles, just as `aggregate`.

```
iris2 <- data.frame(value = unlist(iris[,-5]),
                    Species = rep(iris$Species, dim(iris)[2] -1),
                    Variable = rep(names(iris)[-5], each=dim(iris)[1]))

iris.range <- tapply(iris2$value, iris2[c('Species','Variable')], range)
iris.range
```

```
##             Variable
## Species      Petal.Length Petal.Width Sepal.Length Sepal.Width
##   setosa     numeric,2    numeric,2   numeric,2    numeric,2
##   versicolor numeric,2    numeric,2   numeric,2    numeric,2
##   virginica  numeric,2    numeric,2   numeric,2    numeric,2
```

```
iris.range[['setosa', 'Petal.Length']]
```

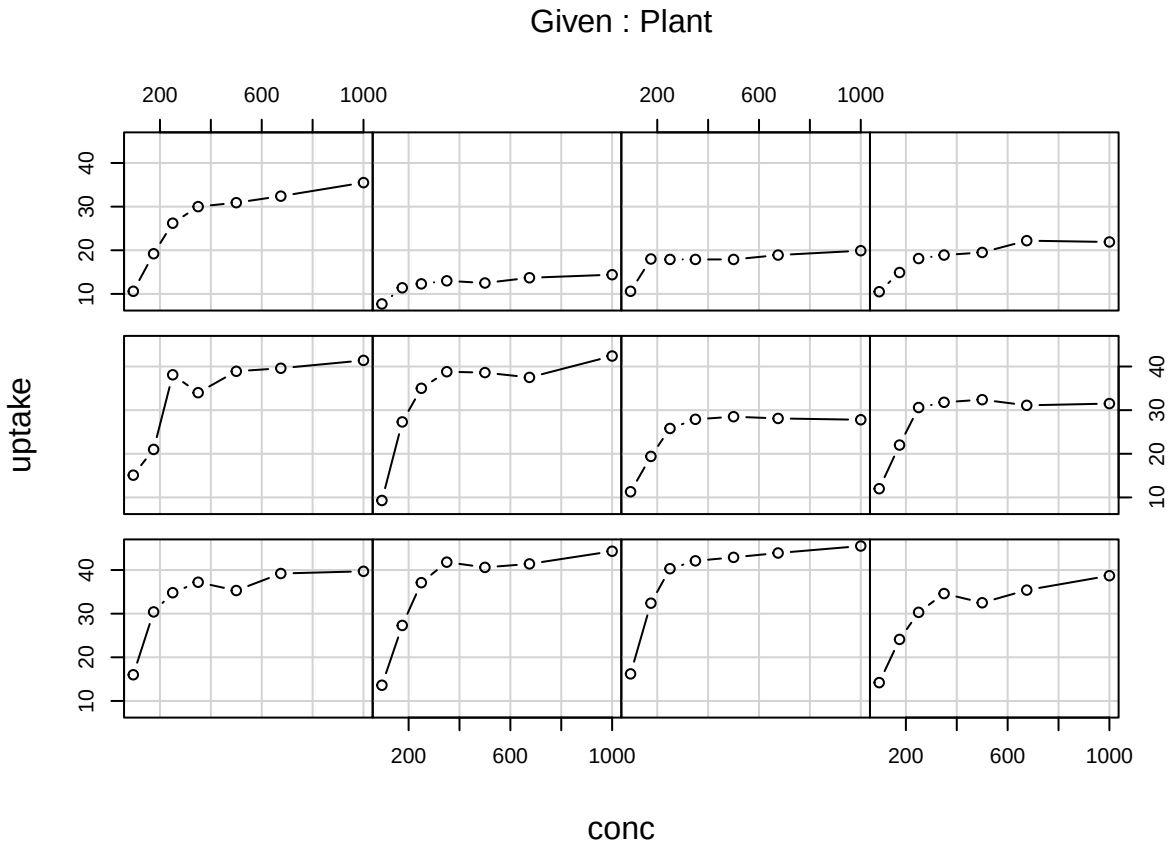
```
## [1] 1.0 1.9
```

To make it useful, we convert it to a data frame:

```
iris.range.df <- data.frame(expand.grid(dimnames(iris.range)),
                            matrix(unlist(iris.range), byrow=T, ncol=2))
```

In many cases, we want to subtract group means from the data. For example, the data set `CO2` consists of results from an experiment for studying cold tolerance of two types of grass under two levels of temperature treatment. The objective is to learn about CO_2 uptake by the plants at various ambient concentrations. It is a nonlinear regression problem. There were 12 plants in the experiment. The plot below shows the response of each plant.

```
coplot(uptake ~ conc | Plant, data = CO2, show.given = F, type = "b")
```



We are interested in the effect of temperature treatment and grass type on the CO_2 uptake process. In a regression problem, it is often necessary to 'center'' the data, that is, subtracting the mean

or median of each treatment-type combination from the data. We can calculate the means usingtapply‘:

```
mns <- tapply(CO2$uptake, CO2[c('Type', 'Treatment')], mean)
mns
```

```
##              Treatment
## Type          nonchilled  chilled
##   Quebec        35.33333 31.75238
##   Mississippi   25.95238 15.81429
```

Now the questions is how do we subtract these means from their respective uptake data. When using `tapply`, we provide a function. If calling `tapply` without a function, a vector of indeces will be returned:

```
inds <- tapply(CO2$uptake, CO2[c('Type', 'Treatment')])
inds
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
## [39] 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 4 4 4 4 4 4 4 4 4 4 4 4
## [77] 4 4 4 4 4 4 4 4
```

These indices can be used as a subscript:

```
mns[inds]
```

```
##  [1] 35.33333 35.33333 35.33333 35.33333 35.33333 35.33333 35.33333 35.33333
##  [9] 35.33333 35.33333 35.33333 35.33333 35.33333 35.33333 35.33333 35.33333
## [17] 35.33333 35.33333 35.33333 35.33333 35.33333 31.75238 31.75238 31.75238
## [25] 31.75238 31.75238 31.75238 31.75238 31.75238 31.75238 31.75238 31.75238
## [33] 31.75238 31.75238 31.75238 31.75238 31.75238 31.75238 31.75238 31.75238
## [41] 31.75238 31.75238 25.95238 25.95238 25.95238 25.95238 25.95238 25.95238
## [49] 25.95238 25.95238 25.95238 25.95238 25.95238 25.95238 25.95238 25.95238
## [57] 25.95238 25.95238 25.95238 25.95238 25.95238 25.95238 25.95238 15.81429
## [65] 15.81429 15.81429 15.81429 15.81429 15.81429 15.81429 15.81429 15.81429
## [73] 15.81429 15.81429 15.81429 15.81429 15.81429 15.81429 15.81429 15.81429
## [81] 15.81429 15.81429 15.81429 15.81429
```

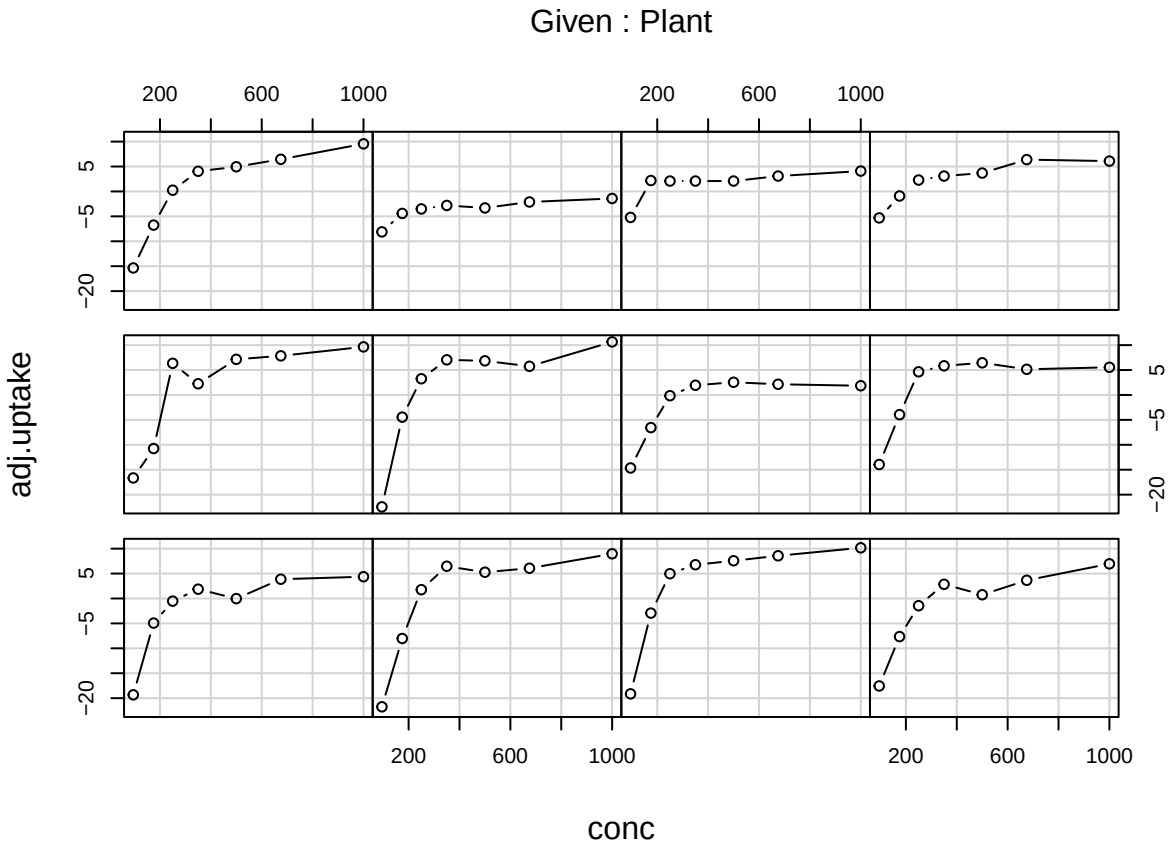
which returns the calculated means for each type-treatment combination. Now the centering is as simple as:

```
adj.uptake <- CO2$uptake - mns[inds]
adj.uptake
```

```
##  [1] -19.33333333  -4.93333333  -0.53333333   1.86666667  -0.03333333
##  [6]   3.86666667   4.36666667 -21.73333333  -8.03333333   1.76666667
## [11]   6.46666667   5.26666667   6.06666667   8.96666667 -19.13333333
## [16]  -2.93333333   4.96666667   6.76666667   7.56666667   8.56666667
## [21]  10.16666667 -17.55238095  -7.65238095  -1.45238095   2.84761905
## [26]   0.74761905   3.64761905   6.94761905 -22.45238095  -4.45238095
## [31]   3.24761905   7.04761905   6.84761905   5.74761905  10.64761905
## [36] -16.65238095 -10.75238095   6.34761905   2.24761905   7.14761905
## [41]   7.84761905   9.64761905 -15.35238095  -6.75238095   0.24761905
```

```
## [46]   4.04761905   4.94761905   6.44761905   9.54761905 -13.95238095
## [51]  -3.95238095   4.64761905   5.84761905   6.44761905   5.14761905
## [56]   5.54761905 -14.65238095  -6.55238095  -0.15238095   1.94761905
## [61]   2.54761905   2.14761905   1.84761905  -5.31428571  -0.91428571
## [66]   2.28571429   3.08571429   3.68571429   6.38571429   6.08571429
## [71]  -8.11428571  -4.41428571  -3.51428571  -2.81428571  -3.31428571
## [76]  -2.11428571  -1.41428571  -5.21428571   2.18571429   2.08571429
## [81]   2.08571429   2.08571429   3.08571429   4.08571429
```

```
coplot(adj.uptake ~ conc | Plant, data = CO2, show.given = F, type = "b")
```



5 Week 6: Reshape

5.1 Package `reshape`

Package `reshape` provides a unified approach to aggregation, based on an extended formula notation. The core idea is to create a "melted" version of a dataset using the `melt` function, which can then be cast (using function `cast`) into an object with the desired orientation. This means that we want to rearrange the form (not the content) of the data to suit the need of analysis.

Before using the package, we need to familiarize the "anatomy" of a dataset. We started by thinking of data interms of vector, matrix, and data frame. In a data frame, observations are in rows and variables are in columns. To reshape a dataset, we can classify variables into "identifier" and "measured" variables.

An identifier (`id`) identifies the unit that the measurements take place. Id variables are usually discrete. A measured variable represents what is measured on that unit, typically the response or predictor variables.

For example, in the `iris` dataset, we have fout measured variables (`Sepal.Length, Sepal.Width, Petal.Length, Petal.Width`) and one id variables (`Species`). Later we reshaped the data to have three columns, `Value, Variable`, and `Species`. In this reshaped dataset, the column `Value` is the measured variable and `Variables, Species` are id variables. In other words, we can generalize the data classification to one measured variable and the rest are id variables.

5.1.1 Melting data

The dataset with one column of measured varaible values and id variables is the most basic data form which we can generate new forms.

To produce the basic form of dataset is not a simple task. The process of creating it is called melting. The `reshape` package provides the function `melt` to carry out the task. For example,

```
iris.melt <- melt(iris, id = "Species",
                  measured=c("Sepal.Width", "Sepal.Length", "Petal.Length","Petal.width"))
head(iris.melt)
```

```
##   Species     variable value
## 1  setosa Sepal.Length   5.1
## 2  setosa Sepal.Length   4.9
## 3  setosa Sepal.Length   4.7
## 4  setosa Sepal.Length   4.6
## 5  setosa Sepal.Length   5.0
## 6  setosa Sepal.Length   5.4
```

The only assumption when using `melt` is that all measured variables are in the same type (e.g., numeric, factor, date).

The dataset `iris.melt` is now a molten data, ready to be casted into different forms. In some cases, your datasets are already in the molten form. All you need to do is to make sure that the value column is named `value`.

5.1.2 Casting molten data

We now have a molten dataset and we can cast it into different forms. All we need is to tell the function `cast` what form we need using a formula.

The casting formula describes the shape of the output format. The general form of a formula includes two groups of variables separated by a tilde. Variables on the left-hand side of the tilde are column variables and those on the right-hand side are row variables. For example,

```
cast(iris.melt, Species ~ variable, length)
```

```
##      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa           50          50           50          50
## 2 versicolor           50          50           50          50
## 3  virginica           50          50           50          50
```

```
cast(iris.melt, Species ~ variable, mean)
```

```
##      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa        5.006       3.428        1.462       0.246
## 2 versicolor        5.936       2.770        4.260       1.326
## 3  virginica        6.588       2.974        5.552       2.026
```

Sumamrizes the sample size and mean.

The function `dcast` casts a molten data into data frame format. The minimum input are (1) a molten data and (2) a formula. The formula can also take $\cdot$ (a single dot) and $\cdots$ (three dots) to represent no variable and all other variables (not already included in the formula).

```
cast(iris.melt, Species ~ variable)
```

```
## Aggregation requires fun.aggregate: length used as default
```

```
##      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa           50          50           50          50
## 2 versicolor           50          50           50          50
## 3  virginica           50          50           50          50
```

```
cast(iris.melt, ... ~ variable)
```

```
## Aggregation requires fun.aggregate: length used as default
```

```
##      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa           50          50           50          50
## 2 versicolor           50          50           50          50
## 3  virginica           50          50           50          50
```

```
cast(iris.melt, Species+variable ~ .)
```

```
## Aggregation requires fun.aggregate: length used as default
```

```
##       Species     variable (all)
## 1      setosa Sepal.Length    50
## 2      setosa  Sepal.Width    50
## 3      setosa Petal.Length    50
## 4      setosa  Petal.Width    50
## 5  versicolor Sepal.Length    50
## 6  versicolor  Sepal.Width    50
## 7  versicolor Petal.Length    50
## 8  versicolor  Petal.Width    50
## 9   virginica Sepal.Length    50
## 10  virginica  Sepal.Width    50
## 11  virginica Petal.Length    50
## 12  virginica  Petal.Width    50
```

What if we want to return to the original data frame? In this case, the molten data lacks one piece of information:

```
iris$rep <- rep(1:50, 3)
iris.melt2 <- melt(iris, id=c("Species", "rep"),
                   measured=c("Sepal.Width", "Sepal.Length", "Petal.Length","Petal.width"))
cast(iris.melt2, Species+rep ~ variable)
```

```
##        Species rep Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1       setosa   1          5.1         3.5          1.4         0.2
## 2       setosa   2          4.9         3.0          1.4         0.2
## 3       setosa   3          4.7         3.2          1.3         0.2
## 4       setosa   4          4.6         3.1          1.5         0.2
## 5       setosa   5          5.0         3.6          1.4         0.2
## 6       setosa   6          5.4         3.9          1.7         0.4
## 7       setosa   7          4.6         3.4          1.4         0.3
## 8       setosa   8          5.0         3.4          1.5         0.2
## 9       setosa   9          4.4         2.9          1.4         0.2
## 10      setosa  10          4.9         3.1          1.5         0.1
## 11      setosa  11          5.4         3.7          1.5         0.2
## 12      setosa  12          4.8         3.4          1.6         0.2
## 13      setosa  13          4.8         3.0          1.4         0.1
## 14      setosa  14          4.3         3.0          1.1         0.1
## 15      setosa  15          5.8         4.0          1.2         0.2
## 16      setosa  16          5.7         4.4          1.5         0.4
## 17      setosa  17          5.4         3.9          1.3         0.4
## 18      setosa  18          5.1         3.5          1.4         0.3
## 19      setosa  19          5.7         3.8          1.7         0.3
## 20      setosa  20          5.1         3.8          1.5         0.3
## 21      setosa  21          5.4         3.4          1.7         0.2
## 22      setosa  22          5.1         3.7          1.5         0.4
## 23      setosa  23          4.6         3.6          1.0         0.2
## 24      setosa  24          5.1         3.3          1.7         0.5
## 25      setosa  25          4.8         3.4          1.9         0.2
## 26      setosa  26          5.0         3.0          1.6         0.2
## 27      setosa  27          5.0         3.4          1.6         0.4
## 28      setosa  28          5.2         3.5          1.5         0.2
## 29      setosa  29          5.2         3.4          1.4         0.2
## 30      setosa  30          4.7         3.2          1.6         0.2
## 31      setosa  31          4.8         3.1          1.6         0.2
## 32      setosa  32          5.4         3.4          1.5         0.4
## 33      setosa  33          5.2         4.1          1.5         0.1
## 34      setosa  34          5.5         4.2          1.4         0.2
## 35      setosa  35          4.9         3.1          1.5         0.2
## 36      setosa  36          5.0         3.2          1.2         0.2
## 37      setosa  37          5.5         3.5          1.3         0.2
## 38      setosa  38          4.9         3.6          1.4         0.1
## 39      setosa  39          4.4         3.0          1.3         0.2
## 40      setosa  40          5.1         3.4          1.5         0.2
## 41      setosa  41          5.0         3.5          1.3         0.3
## 42      setosa  42          4.5         2.3          1.3         0.3
## 43      setosa  43          4.4         3.2          1.3         0.2
## 44      setosa  44          5.0         3.5          1.6         0.6
## 45      setosa  45          5.1         3.8          1.9         0.4
## 46      setosa  46          4.8         3.0          1.4         0.3
## 47      setosa  47          5.1         3.8          1.6         0.2
```

```
## 48      setosa  48          4.6         3.2          1.4         0.2
## 49      setosa  49          5.3         3.7          1.5         0.2
## 50      setosa  50          5.0         3.3          1.4         0.2
## 51  versicolor   1          7.0         3.2          4.7         1.4
## 52  versicolor   2          6.4         3.2          4.5         1.5
## 53  versicolor   3          6.9         3.1          4.9         1.5
## 54  versicolor   4          5.5         2.3          4.0         1.3
## 55  versicolor   5          6.5         2.8          4.6         1.5
## 56  versicolor   6          5.7         2.8          4.5         1.3
## 57  versicolor   7          6.3         3.3          4.7         1.6
## 58  versicolor   8          4.9         2.4          3.3         1.0
## 59  versicolor   9          6.6         2.9          4.6         1.3
## 60  versicolor  10          5.2         2.7          3.9         1.4
## 61  versicolor  11          5.0         2.0          3.5         1.0
## 62  versicolor  12          5.9         3.0          4.2         1.5
## 63  versicolor  13          6.0         2.2          4.0         1.0
## 64  versicolor  14          6.1         2.9          4.7         1.4
## 65  versicolor  15          5.6         2.9          3.6         1.3
## 66  versicolor  16          6.7         3.1          4.4         1.4
## 67  versicolor  17          5.6         3.0          4.5         1.5
## 68  versicolor  18          5.8         2.7          4.1         1.0
## 69  versicolor  19          6.2         2.2          4.5         1.5
## 70  versicolor  20          5.6         2.5          3.9         1.1
## 71  versicolor  21          5.9         3.2          4.8         1.8
## 72  versicolor  22          6.1         2.8          4.0         1.3
## 73  versicolor  23          6.3         2.5          4.9         1.5
## 74  versicolor  24          6.1         2.8          4.7         1.2
## 75  versicolor  25          6.4         2.9          4.3         1.3
## 76  versicolor  26          6.6         3.0          4.4         1.4
## 77  versicolor  27          6.8         2.8          4.8         1.4
## 78  versicolor  28          6.7         3.0          5.0         1.7
## 79  versicolor  29          6.0         2.9          4.5         1.5
## 80  versicolor  30          5.7         2.6          3.5         1.0
## 81  versicolor  31          5.5         2.4          3.8         1.1
## 82  versicolor  32          5.5         2.4          3.7         1.0
## 83  versicolor  33          5.8         2.7          3.9         1.2
## 84  versicolor  34          6.0         2.7          5.1         1.6
## 85  versicolor  35          5.4         3.0          4.5         1.5
## 86  versicolor  36          6.0         3.4          4.5         1.6
## 87  versicolor  37          6.7         3.1          4.7         1.5
## 88  versicolor  38          6.3         2.3          4.4         1.3
## 89  versicolor  39          5.6         3.0          4.1         1.3
## 90  versicolor  40          5.5         2.5          4.0         1.3
## 91  versicolor  41          5.5         2.6          4.4         1.2
## 92  versicolor  42          6.1         3.0          4.6         1.4
## 93  versicolor  43          5.8         2.6          4.0         1.2
## 94  versicolor  44          5.0         2.3          3.3         1.0
## 95  versicolor  45          5.6         2.7          4.2         1.3
## 96  versicolor  46          5.7         3.0          4.2         1.2
## 97  versicolor  47          5.7         2.9          4.2         1.3
## 98  versicolor  48          6.2         2.9          4.3         1.3
## 99  versicolor  49          5.1         2.5          3.0         1.1
## 100 versicolor  50          5.7         2.8          4.1         1.3
## 101  virginica   1          6.3         3.3          6.0         2.5
```

```
## 102  virginica   2          5.8          2.7          5.1          1.9
## 103  virginica   3          7.1          3.0          5.9          2.1
## 104  virginica   4          6.3          2.9          5.6          1.8
## 105  virginica   5          6.5          3.0          5.8          2.2
## 106  virginica   6          7.6          3.0          6.6          2.1
## 107  virginica   7          4.9          2.5          4.5          1.7
## 108  virginica   8          7.3          2.9          6.3          1.8
## 109  virginica   9          6.7          2.5          5.8          1.8
## 110  virginica  10          7.2          3.6          6.1          2.5
## 111  virginica  11          6.5          3.2          5.1          2.0
## 112  virginica  12          6.4          2.7          5.3          1.9
## 113  virginica  13          6.8          3.0          5.5          2.1
## 114  virginica  14          5.7          2.5          5.0          2.0
## 115  virginica  15          5.8          2.8          5.1          2.4
## 116  virginica  16          6.4          3.2          5.3          2.3
## 117  virginica  17          6.5          3.0          5.5          1.8
## 118  virginica  18          7.7          3.8          6.7          2.2
## 119  virginica  19          7.7          2.6          6.9          2.3
## 120  virginica  20          6.0          2.2          5.0          1.5
## 121  virginica  21          6.9          3.2          5.7          2.3
## 122  virginica  22          5.6          2.8          4.9          2.0
## 123  virginica  23          7.7          2.8          6.7          2.0
## 124  virginica  24          6.3          2.7          4.9          1.8
## 125  virginica  25          6.7          3.3          5.7          2.1
## 126  virginica  26          7.2          3.2          6.0          1.8
## 127  virginica  27          6.2          2.8          4.8          1.8
## 128  virginica  28          6.1          3.0          4.9          1.8
## 129  virginica  29          6.4          2.8          5.6          2.1
## 130  virginica  30          7.2          3.0          5.8          1.6
## 131  virginica  31          7.4          2.8          6.1          1.9
## 132  virginica  32          7.9          3.8          6.4          2.0
## 133  virginica  33          6.4          2.8          5.6          2.2
## 134  virginica  34          6.3          2.8          5.1          1.5
## 135  virginica  35          6.1          2.6          5.6          1.4
## 136  virginica  36          7.7          3.0          6.1          2.3
## 137  virginica  37          6.3          3.4          5.6          2.4
## 138  virginica  38          6.4          3.1          5.5          1.8
## 139  virginica  39          6.0          3.0          4.8          1.8
## 140  virginica  40          6.9          3.1          5.4          2.1
## 141  virginica  41          6.7          3.1          5.6          2.4
## 142  virginica  42          6.9          3.1          5.1          2.3
## 143  virginica  43          5.8          2.7          5.1          1.9
## 144  virginica  44          6.8          3.2          5.9          2.3
## 145  virginica  45          6.7          3.3          5.7          2.5
## 146  virginica  46          6.7          3.0          5.2          2.3
## 147  virginica  47          6.3          2.5          5.0          1.9
## 148  virginica  48          6.5          3.0          5.2          2.0
## 149  virginica  49          6.2          3.4          5.4          2.3
## 150  virginica  50          5.9          3.0          5.1          1.8
```

5.1.3 The French Fries Example

```
ffm <- melt(french_fries, id=1:4, na.rm=T)

### Counting nonmissing values:
cast(ffm, .~., length)
```

```
##   value (all)
## 1 (all)  3471
```

```
### counting by treatment
cast(ffm, treatment ~ . , length)
```

```
##   treatment (all)
## 1         1  1159
## 2         2  1157
## 3         3  1155
```

```
cast(ffm, . ~ treatment, length)
```

```
##   value    1    2    3
## 1 (all) 1159 1157 1155
```

```
### rep by treatment
cast(ffm, rep ~ treatment)
```

```
## Aggregation requires fun.aggregate: length used as default
```

```
##   rep   1   2   3
## 1   1 579 578 575
## 2   2 580 579 580
```

```
### treatment by rep:
cast(ffm, treatment ~ rep)
```

```
## Aggregation requires fun.aggregate: length used as default
```

```
##   treatment   1   2
## 1         1 579 580
## 2         2 578 579
## 3         3 575 580
```

```
cast(ffm, treatment + rep ~ .)
```

```
## Aggregation requires fun.aggregate: length used as default
```

```
##   treatment rep (all)
## 1         1   1   579
## 2         1   2   580
## 3         2   1   578
## 4         2   2   579
## 5         3   1   575
## 6         3   2   580
```

```
cast(ffm, rep + treatment ~ .)
```

```
## Aggregation requires fun.aggregate: length used as default
```

```
##   rep treatment (all)
## 1   1         1   579
## 2   1         2   578
## 3   1         3   575
## 4   2         1   580
## 5   2         2   579
## 6   2         3   580
```

```
cast(ffm, . ~ rep+treatment)
```

```
## Aggregation requires fun.aggregate: length used as default
```

```
##   value 1_1 1_2 1_3 2_1 2_2 2_3
## 1 (all) 579 578 575 580 579 580
```

Summary:

```
cast(ffm, rep+treatment~variable)
```

```
## Aggregation requires fun.aggregate: length used as default
```

```
##   rep treatment potato buttery grassy rancid painty
## 1   1         1    116     115    116    116    116
## 2   1         2    116     114    116    116    116
## 3   1         3    115     115    115    115    115
## 4   2         1    116     116    116    116    116
## 5   2         2    116     116    116    116    115
## 6   2         3    116     116    116    116    116
```

```
cast(ffm, rep+treatment~variable, mean)
```

```
##   rep treatment   potato  buttery    grassy   rancid   painty
## 1   1         1 6.772414 1.797391 0.4456897 4.283621 2.727586
## 2   1         2 7.158621 1.989474 0.6905172 3.712069 2.315517
## 3   1         3 6.937391 1.805217 0.5895652 3.752174 2.038261
## 4   2         1 7.003448 1.762931 0.8525862 3.847414 2.439655
## 5   2         2 6.844828 1.958621 0.6353448 3.537069 2.597391
## 6   2         3 6.998276 1.631034 0.7706897 3.980172 3.008621
```

```
cast(ffm, rep+treatment~variable, sd)
```

```
##   rep treatment   potato  buttery    grassy   rancid   painty
## 1   1         1 3.689906 2.625297 0.8700217 4.046296 3.296126
## 2   1         2 3.541090 2.522584 1.3266485 3.771130 3.534442
## 3   1         3 3.453030 2.388894 1.0575023 3.522244 2.860670
## 4   2         1 3.799655 2.318115 1.7311701 3.680205 3.369192
## 5   2         2 3.634720 2.461391 1.1791857 3.833934 3.488765
## 6   2         3 3.437921 2.151740 1.5519983 3.852367 3.734599
```

```
cast(ffm, treatment ~ ., c(mean, sd))
```

```
##   treatment     mean       sd
## 1         1 3.194478 3.772217
## 2         2 3.146413 3.750471
## 3         3 3.151688 3.662970
```

```
cast(ffm, treatment ~ ., summary)
```

```
##   treatment Min. X1st.Qu. Median     Mean X3rd.Qu. Max.
## 1         1    0        0    1.6 3.194478      5.4 14.9
## 2         2    0        0    1.4 3.146413      5.4 14.9
## 3         3    0        0    1.5 3.151688      5.7 14.5
```

```
cast(ffm, treatment ~ variable + result_variable, c(mean, sd))
```

```
##   treatment potato_mean potato_sd buttery_mean buttery_sd grassy_mean grassy_sd
## 1         1    6.887931  3.738860     1.780087   2.470476   0.6491379  1.382168
## 2         2    7.001724  3.583886     1.973913   2.486508   0.6629310  1.252670
## 3         3    6.967965  3.438088     1.717749   2.269627   0.6805195  1.329240
##   rancid_mean rancid_sd painty_mean painty_sd
## 1    4.065517  3.865389    2.583621  3.328766
## 2    3.624569  3.795435    2.455844  3.506980
## 3    3.866667  3.685455    2.525541  3.356512
```

```
cast(ffm, treatment + variable ~ result_variable, c(mean, sd))
```

```
##    treatment variable      mean       sd
## 1          1   potato 6.8879310 3.738860
## 2          1  buttery 1.7800866 2.470476
## 3          1   grassy 0.6491379 1.382168
## 4          1   rancid 4.0655172 3.865389
## 5          1   painty 2.5836207 3.328766
## 6          2   potato 7.0017241 3.583886
## 7          2  buttery 1.9739130 2.486508
## 8          2   grassy 0.6629310 1.252670
## 9          2   rancid 3.6245690 3.795435
## 10         2   painty 2.4558442 3.506980
## 11         3   potato 6.9679654 3.438088
## 12         3  buttery 1.7177489 2.269627
## 13         3   grassy 0.6805195 1.329240
## 14         3   rancid 3.8666667 3.685455
## 15         3   painty 2.5255411 3.356512
```

Conditional operator

```
cast(ffm, treatment~rep|variable, mean)
```

```
## $potato
##   treatment        1        2
## 1         1 6.772414 7.003448
## 2         2 7.158621 6.844828
## 3         3 6.937391 6.998276
##
## $buttery
##   treatment        1        2
## 1         1 1.797391 1.762931
## 2         2 1.989474 1.958621
## 3         3 1.805217 1.631034
##
## $grassy
##   treatment         1         2
## 1         1 0.4456897 0.8525862
## 2         2 0.6905172 0.6353448
## 3         3 0.5895652 0.7706897
##
## $rancid
##   treatment        1        2
## 1         1 4.283621 3.847414
## 2         2 3.712069 3.537069
## 3         3 3.752174 3.980172
##
## $painty
##   treatment        1        2
## 1         1 2.727586 2.439655
## 2         2 2.315517 2.597391
## 3         3 2.038261 3.008621
```

5.2 Package `reshape2`

Reshape2 is a reboot of the reshape package, a new package for reshaping data that is much more focused and faster.

This version improves speed at the cost of functionality.

What's new in reshape2:

– considerably faster and more memory efficient thanks to a much better underlying algorithm that uses the power and speed of subsetting to the fullest extent, in most cases only making a single copy of the data. `cast`is replaced by two functions depending on the output type: dcast produces data frames, and acast produces matrices/arrays.

– multidimensional margins are now possible: grand_row and grand_col have been dropped: now the name of the margin refers to the variable that has its value set to (all).

– some features have been removed such as the | cast operator, and the ability to return multiple values from an aggregation function.

– a new cast syntax which allows you to reshape based on functions of variables (based on the same underlying syntax as plyr): better development practices like namespaces and tests.

5.3 Reshaping Data

THe advantage of using R for statistical analysis is that R comes with functions to carry out statistical analysis. But each function requires a specific input format. As a result, reshaping data to suit for the need of a function is the most commonly performed task. In most cases, R functions require input using data frame. We will focus on working with data frames.

5.3.1 Modifying data frame variables

Data frames are lists. New variables can be creates by simply assigning their value to a column that doesn't already exist in the data frame. Let's use the loblolly pine growth data as an example. This dataset has three columns recording the growth of loblolly pine trees over time.

```
head(Loblolly)
```

```
## Grouped Data: height ~ age | Seed
##    height age Seed
## 1    4.51   3  301
## 15  10.89   5  301
## 29  28.72  10  301
## 43  41.74  15  301
## 57  52.70  20  301
## 71  60.92  25  301
```

Variable transformation is probably the most common modification to a dataset. We add the transformed variable into an additional column:

```
Loblolly$logheight <- log(Loblolly$height)
```

(When we use data from R or R packages, all changes we made will be kept in the memory of the current R session. Data from the system won't be changed.)

When the data frame name is long, we often use the function `transform` to reduce the amount of typing.

```
Loblolly <- transform(Loblolly, logheight=log(height))
```

We access variables in a data frame through the data frame name plus the `$` operator and variable names. The function `with` helps to reduce the repeated typing of the data frame name. When using `with`, R puts the data frame temperarily on the top of the search list:

```
logheight<-with(Loblolly, log(height))
```

When this line is submitted, R looks into the data frame `Loblolly` first to find `height`. If a wrong name was typed, R would not be able to find it in the data frame, R will continue to search outside the data frame.

```
height2 <- runif(20, 3, 85)
logheight2 <- with(Loblolly, log(height2))
```

We can remove a column of a data frame by setting its values to be `NULL`:

```
head(Loblolly)
```

```
##    height age Seed logheight
## 1    4.51   3  301  1.506297
## 15  10.89   5  301  2.387845
## 29  28.72  10  301  3.357594
## 43  41.74  15  301  3.731460
## 57  52.70  20  301  3.964615
## 71  60.92  25  301  4.109562
```

```
Loblolly$logheight <- NULL
head(Loblolly)
```

```
##    height age Seed
## 1    4.51   3  301
## 15  10.89   5  301
## 29  28.72  10  301
## 43  41.74  15  301
## 57  52.70  20  301
## 71  60.92  25  301
```

When a similar operation is needed for several columns and we want to overwrite the original versions, we can put the multiple columns we want to replace on the left hand side of the assignment operator. For example, if we want to convert the unit of iris measurements from centimeters to inches, we need to divide each of the 4 measurement variables by 2.54:

```
iris[,-5] <- sapply(iris[,-5], function(x) x/2.54)
```

Recoding a variable is another form of data transformation. For example, when the actual value of sepal length is not important, but whether the length is above or below 6 cm is important, we would like to have a column to indicate whether sepal length is above 6 or not. This is a simple logic comparison:

```
largesepal <- iris$Sepal.Length > 6
```

In some cases, the function `ifelse` is very handy. The function takes a logical vector as its first argument and two other arguments: the first provides a value for the case where elements of the input logical vector are true, and the second for the case where they are false. For example, suppose that we have experiments carried out in 10 sites and we realized that sites 2 and 4 can be grouped in one group and the rest in another group. The grouping can be created as follows:

```
newgroup <- ifelse(sites %in% c(2,4), 1, 2)
```

To make thing more complicated, suppose that sites 2 and 4 are in group 1, sites 5 and 8 are in group 2 and the rest are in group 3:

```
newgroup <- ifelse(sites %in% c(2,4), 1,
                   ifelse (sites %in% c(5, 8), 2, 3))
```

5.3.2 Reshaping a data frame

Not all data frame have all the necessary identifiers. For example, the `iris` data frame lacks the flower indentifier. Let's work with the iris data for one species:

```
iris.setosa <- iris[iris$Species=="setosa",-5]
head(iris.setosa)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width       rep
## 1     2.007874    1.377953    0.5511811  0.07874016 0.3937008
## 2     1.929134    1.181102    0.5511811  0.07874016 0.7874016
## 3     1.850394    1.259843    0.5118110  0.07874016 1.1811024
## 4     1.811024    1.220472    0.5905512  0.07874016 1.5748031
## 5     1.968504    1.417323    0.5511811  0.07874016 1.9685039
## 6     2.125984    1.535433    0.6692913  0.15748031 2.3622047
```

Many R functions for statistical analysis requires that the data frame be reshaped to have a column of measured value and a column of variable.

```
setosaM <- melt(iris.setosa,
                measured=c("Sepal.Width", "Sepal.Length", "Petal.Length","Petal.width"))
```

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

In this case, a much simpler approach would be to use the function `stack`:

```
setosaS <- stack(iris.setosa)
```

The function `unstack` returns the data to its original form:

```
setosaUS <- unstack(setosaS, values~ind)
```

5.3.3 Function `reshape`

Not to be confused with the package `reshape`, the function `reshape` is written for data with repeated measures to convert a data frame between long and wide forms. A long form data frame puts multiple measurements of a subject (replicates) in separate rows, while the wide form puts them in the same row. Here is an example of long form:

```
obs <- data.frame(subj = rep(1:4, each = 3),
                  time=rep(1:3, 4),
                  x=rnorm(12), y=rnorm(12))
head(obs)
```

```
##   subj time          x          y
## 1    1    1  0.2473890  0.9810028
## 2    1    2  1.4415252  0.6753019
## 3    1    3 -0.8611905 -0.6418240
## 4    2    1  0.8903991  0.6148049
## 5    2    2  0.3444934  0.8270787
## 6    2    3  0.7771511  0.2595607
```

A wide format has 4 rows, one for each subject:

```
obs.wide <- reshape(obs, idvar="subj", v.names=c("x","y"),
                         timevar="time", direction="wide")
obs.wide
```

```
##    subj        x.1        y.1        x.2         y.2        x.3        y.3
## 1     1  0.2473890  0.9810028  1.4415252  0.67530185 -0.8611905 -0.6418240
## 4     2  0.8903991  0.6148049  0.3444934  0.82707867  0.7771511  0.2595607
## 7     3 -0.4464762 -1.6600949 -0.6725280 -0.23735434 -1.6392050 -0.9801708
## 10    4  0.6461872  2.0127724 -0.1322582 -0.09296755 -0.1057505  1.7255400
```

```
str(obs.wide)
```

```
## 'data.frame':    4 obs. of  7 variables:
##  $ subj: int  1 2 3 4
##  $ x.1 : num  0.247 0.89 -0.446 0.646
##  $ y.1 : num  0.981 0.615 -1.66 2.013
##  $ x.2 : num  1.442 0.344 -0.673 -0.132
##  $ y.2 : num  0.675 0.827 -0.237 -0.093
##  $ x.3 : num  -0.861 0.777 -1.639 -0.106
##  $ y.3 : num  -0.642 0.26 -0.98 1.726
##  - attr(*, "reshapeWide")=List of 5
##   ..$ v.names: chr [1:2] "x" "y"
##   ..$ timevar: chr "time"
##   ..$ idvar  : chr "subj"
##   ..$ times  : int [1:3] 1 2 3
##   ..$ varying: chr [1:2, 1:3] "x.1" "y.1" "x.2" "y.2" ...
```

```
reshape(obs.wide, idvar="subj", direction="long")
```

```
##      subj time          x           y
## 1.1     1    1  0.2473890  0.98100276
## 2.1     2    1  0.8903991  0.61480493
## 3.1     3    1 -0.4464762 -1.66009493
## 4.1     4    1  0.6461872  2.01277239
## 1.2     1    2  1.4415252  0.67530185
## 2.2     2    2  0.3444934  0.82707867
## 3.2     3    2 -0.6725280 -0.23735434
## 4.2     4    2 -0.1322582 -0.09296755
## 1.3     1    3 -0.8611905 -0.64182402
## 2.3     2    3  0.7771511  0.25956075
## 3.3     3    3 -1.6392050 -0.98017083
## 4.3     4    3 -0.1057505  1.72554000
```

5.3.4 Combining Data Frames

The two frequently used functions are `rbind` (binding data frames by row) and `cbind` (binding by column).

```
x <- data.frame(a=c("A","B","C"), b=c(1,2,3))
y <- data.frame(a=c("D","E","F","G"), b=c(3,4,5,6))

z <- rbind(x, y)
```

When using `rbind`, the two data frames must have the same names. When using `cbind`, the two data frames must have the same number of rows (or one is a multipler of the other):

```
cbind(y, z=c(1,2))
```

```
##   a b z
## 1 D 3 1
## 2 E 4 2
## 3 F 5 1
## 4 G 6 2
```

```
cbind(y, z=rep(1:2, 2))
```

```
##   a b z
## 1 D 3 1
## 2 E 4 2
## 3 F 5 1
## 4 G 6 2
```

But not:

```
cbind(x, z=c(1,2))
```

Merging two data frames based on a common factor is often necessary. The function `merge` provides a flexible means for merging.

```
x<-data.frame(a=c(1,2,4,5,6), x=c(9,12,14,21,8))
y<-data.frame(a=c(1,3,4,6), y=c(8, 14, 19, 2))
merge(x, y)
```

```
##   a  x  y
## 1 1  9  8
## 2 4 14 19
## 3 6  8  2
```

The result may not be what you expect.

By default, `merge` will find the common name in `x` and `y` to use as the variable for merging (argument `by`). If the merging variable has different names in `x` and `y`, we can use argument `by.x` and `by.y`. The argument `all` (default `all=FALSE`) is a logical value indicating whether all rows in both `x` and `y` should be included:

```
merge(x, y, all=T)
```

```
##   a  x  y
## 1 1  9  8
## 2 2 12 NA
## 3 3 NA 14
## 4 4 14 19
## 5 5 21 NA
## 6 6  8  2
```

If only all rows of `x` (`y`) are needed, we specify `all.x=T` (`all.y=T`).

```
merge(x, y, all.x=T)
merge(x, y, all.y=T)
```

One more example of `merge`:

```
authors <- data.frame(
    surname = I(c("Tukey", "Venables", "Tierney", "Ripley", "McNeil")),
    nationality = c("US", "Australia", "US", "UK", "Australia"),
    deceased = c("yes", rep("no", 4)))
books <- data.frame(
    name = I(c("Tukey", "Venables", "Tierney",
             "Ripley", "Ripley", "McNeil", "R Core")),
    title = c("Exploratory Data Analysis",
              "Modern Applied Statistics ...",
              "LISP-STAT",
              "Spatial Statistics", "Stochastic Simulation",
              "Interactive Data Analysis",
              "An Introduction to R"),
    other.author = c(NA, "Ripley", NA, NA, NA, NA,
                     "Venables & Smith"))

m1 <- merge(authors, books, by.x="surname", by.y="name")
m2 <- merge(books, authors, by.x = "name", by.y = "surname")
m3 <- merge(books, authors, by.x = "name", by.y = "surname", all=T)
```

5.3.5 Merging using index

The function `merge` is convenient. In many cases, especially when we want to mearge data by a set of unique index (e.g., site names), we can simply use subscripting. For example:

```
x <- data.frame(site=c("S3","S2","S1"), Ag=c(20, 31, 19))
y <- data.frame(site=c("S1","S1","S1","S2","S2","S3","S3","S1","S2","S3"),
                y=rnorm(10))
```

Because the site names are the same, we can first sort `x` by site:

```
x <- x[order(x$site),]
x
```

```
##   site Ag
## 3   S1 19
## 2   S2 31
## 1   S3 20
```

then:

```
oo <- as.numeric(ordered(y$site))
oo
```

```
##  [1] 1 1 1 2 2 3 3 1 2 3
```

```
y$Ag <- x$Ag[oo]
y
```

```
##    site           y Ag
## 1    S1 -0.29704850 19
## 2    S1 -1.86891556 19
## 3    S1  0.48784323 19
## 4    S2  1.26442246 31
## 5    S2  0.09352748 31
## 6    S3  0.11303946 20
## 7    S3 -1.10709195 20
## 8    S1  0.50223919 19
## 9    S2 -1.30150275 31
## 10   S3  0.52096383 20
```

6 Week 8

6.1 Factors

Factors are categorical variables, taking on a limited number of unique values. In statistical modeling, factor variables are used in many ways. Properly handle factor variables is an important aspect of data management.

In R, a factor variable is stored as a vector of integers, each with a corresponding character value to use when displayed. To create a factor, we use `factor`, which requires a vector. A factor variable has two attributes: "levels" and "nlevels". The functions `levels` and `nlevels` returns these two attributes. Levels of a factor are names of the limited number of unique character values. When creating a factor variable using function `factor`, we can specify the levels using `levels=` argument to give a vector of all possible values of the variable in the order you want. If the order of elvels are important, we should use an ordered factor by specifying `ordered=T` as an argument when creating a factor using the `factor` function.

```
data <- c(1,2,2,3,1,2,3,3,1,2,3,3,1)
fdata <- factor(data)
fdata
```

```
##  [1] 1 2 2 3 1 2 3 3 1 2 3 3 1
## Levels: 1 2 3
```

To change the names of level:

```
rdata <- factor(data, labels = c("I","II","III"))
rdata
```

```
##  [1] I   II  II  III I   II  III III I   II  III III I
## Levels: I II III
```

or

```
levels(fdata) <- c("a","b","c")
fdata
```

```
##  [1] a b b c a b c c a b c c a
## Levels: a b c
```

A factor variable is the most efficient means for storing character variables. Each unique character value is stored only once and data itself is stored as a vector of integers. As a result, when reading data using `read.table`, R will automatically convert character variables into factors. You can turn this feature off by using either `as.is=TRUE` or `stringsAsFactors=FALSE`.

By default, R converts a character vector into a factor, and presenting in the natural alphabetic order. For example,

```
mons <- c("March","April","January","November","January","September","October","September",
          "August","January","February","July","June", "May", "March","April","January",
          "November","January","September","October","September", "August","January",
          "February","July","June", "May","December","December")
table(mons)
```

```
## mons
##     April    August  December  February   January      July      June     March
##         2         2         2         2         6         2         2         2
##       May  November   October September
##         2         2         2         4
```

```
mons <- factor(mons)
mons
```

```
##  [1] March     April     January   November  January   September October
##  [8] September August    January   February  July      June      May
## [15] March     April     January   November  January   September October
## [22] September August    January   February  July      June      May
## [29] December  December
## 12 Levels: April August December February January July June March ... September
```

The ordering is alphabetic. Months have an ordering. To make the correct order, we should use an ordered factor:

```
mons <- factor(mons, levels=c("January","February","March","April","May","June",
                              "July","August","September","October","November","December"),
               ordered=T)
mons
```

```
##  [1] March     April     January   November  January   September October
##  [8] September August    January   February  July      June      May
## [15] March     April     January   November  January   September October
## [22] September August    January   February  July      June      May
## [29] December  December
## 12 Levels: January < February < March < April < May < June < ... < December
```

```
mons[1] < mons[2]
```

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

With an ordered factor, the presentation uses the specified order:

```
table(mons)
```

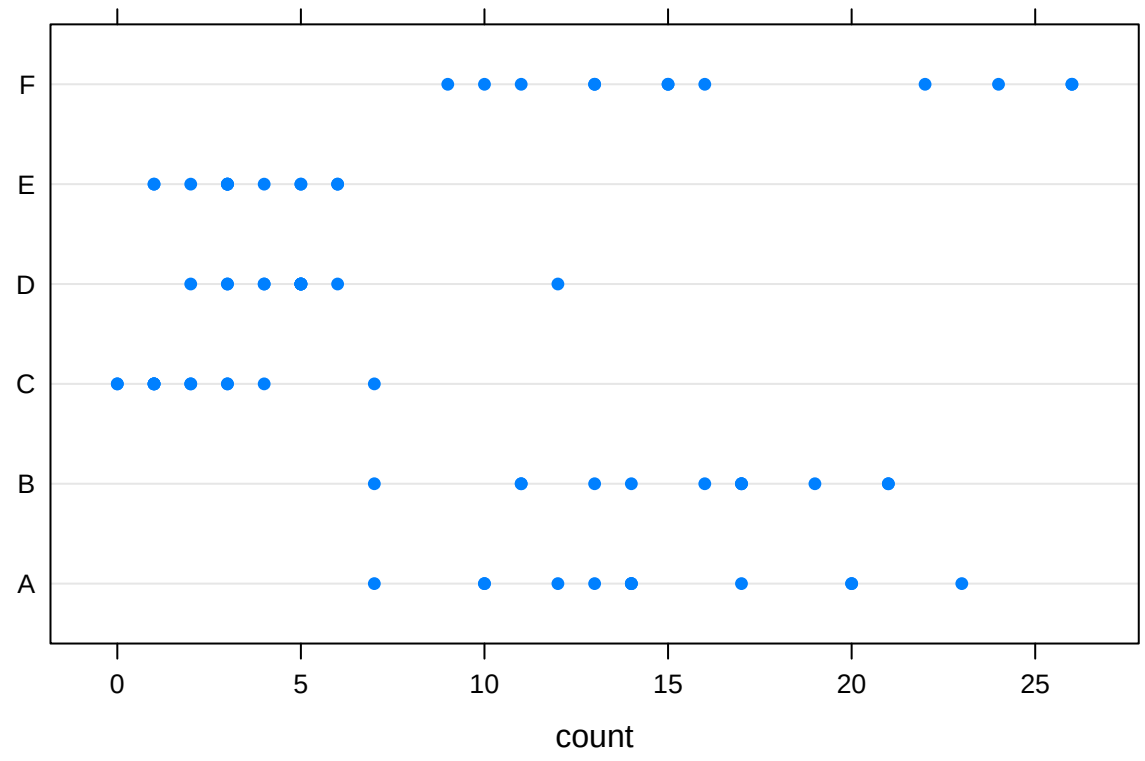
```
## mons
##   January  February     March     April       May      June      July    August
##         6         2         2         2         2         2         2         2
## September   October  November  December
##         4         2         2         2
```

Months have a natural order. Some times the order of a factor variable should be determined by some properties of the data. For example, the data set `InsectSpray` in R records the number of insects killed in agriculture field experiments with 6 insecticides, labeled alphabetically.

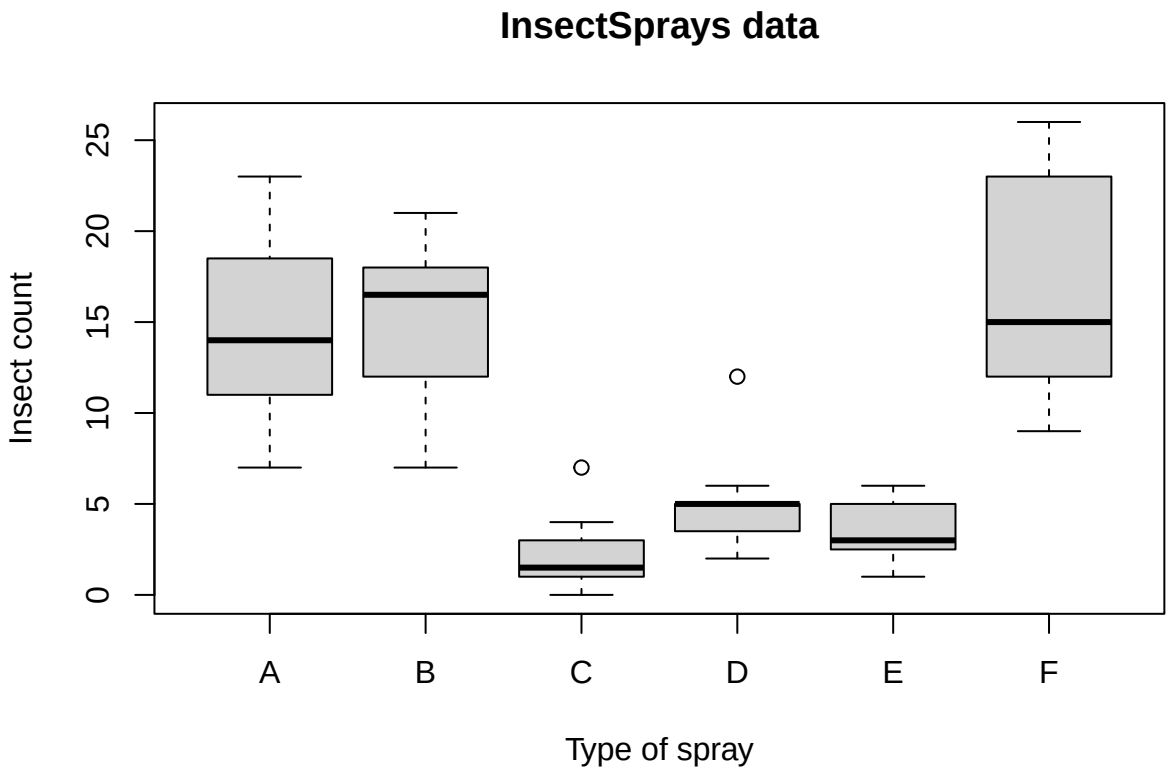
```
packages(lattice)
levels(InsectSprays$spray)
```

```
## [1] "A" "B" "C" "D" "E" "F"
```

```
dotplot(spray ~ count, data=InsectSprays)
```



```
boxplot(count ~ spray, data=InsectSprays,  xlab = "Type of spray", ylab = "Insect count",
        main = "InsectSprays data", varwidth = TRUE, col = "lightgray")
```



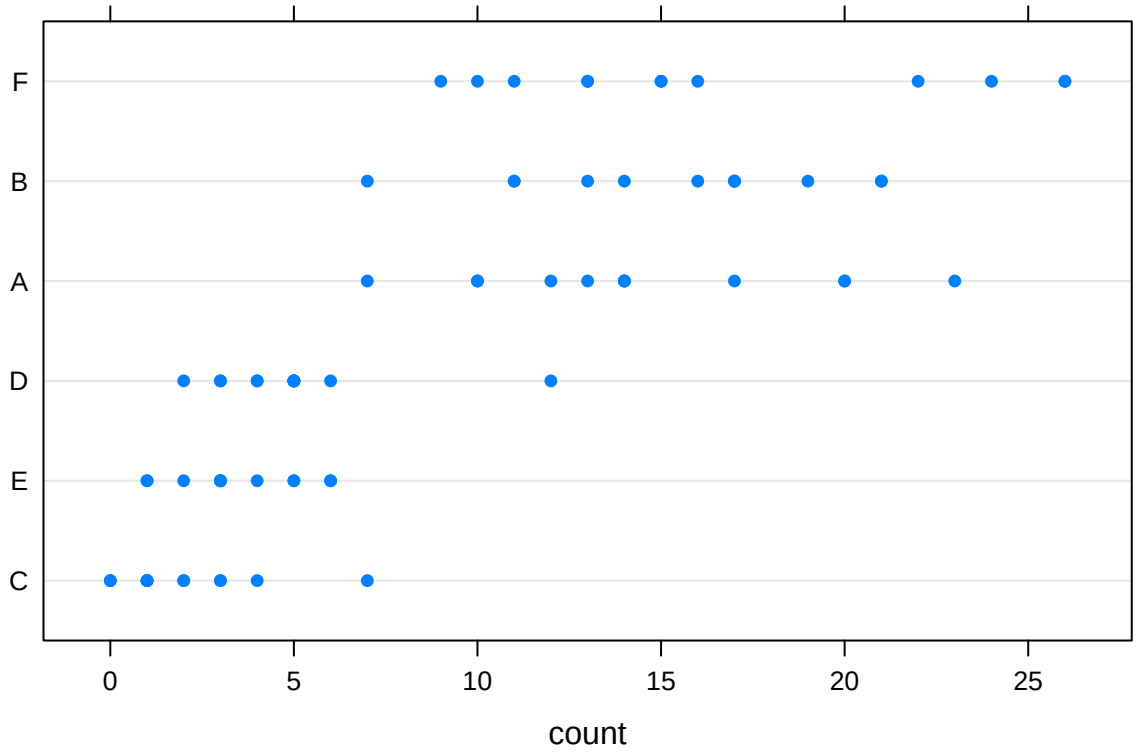
The label has no meaning. But ordering the type sprays by the mean number of counts would help visualizing the relative effectiveness of these sprays. To reorder a factor variable, we can use the function `reorder`:

```
InsectSprays$spray <- reorder(InsectSprays$spray, InsectSprays$count, mean)
levels(InsectSprays$spray)
```

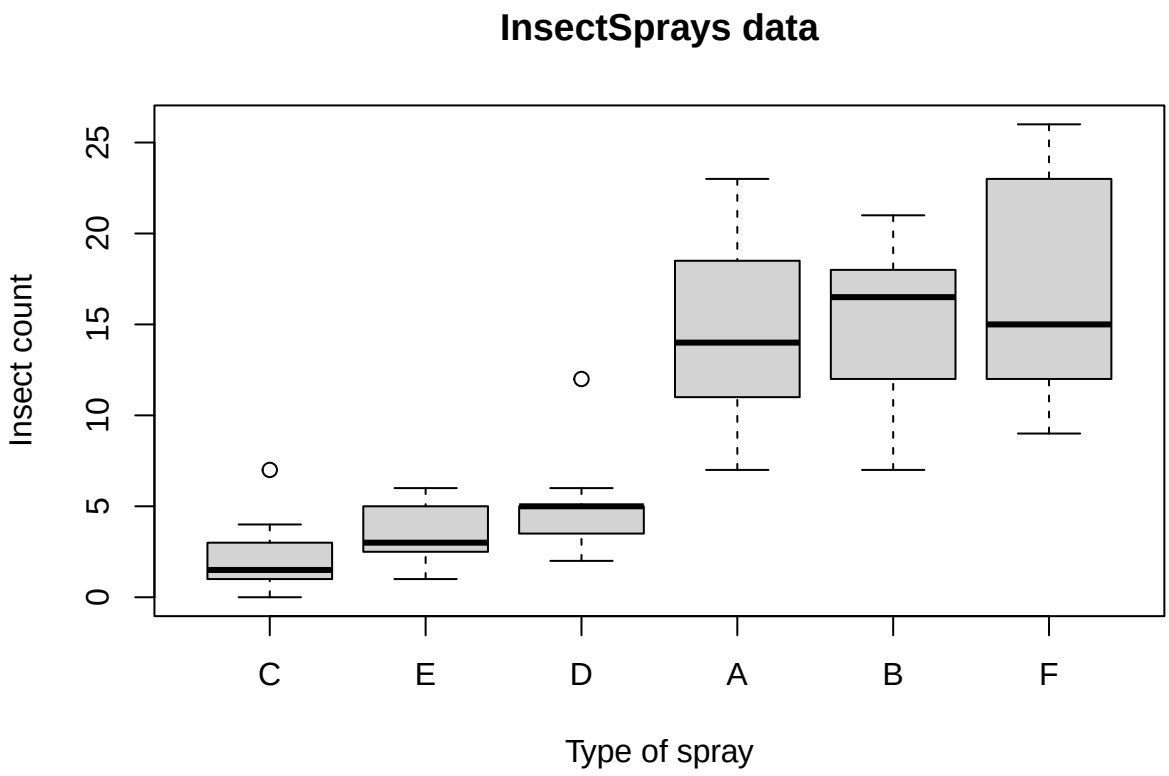
```
## [1] "C" "E" "D" "A" "B" "F"
```

The function takes three arguments: the factor variable to be reordered, a vector of values that the reordering is based, and a function to operate on those values for each factor level.

```
dotplot(spray ~ count, data=InsectSprays)
```

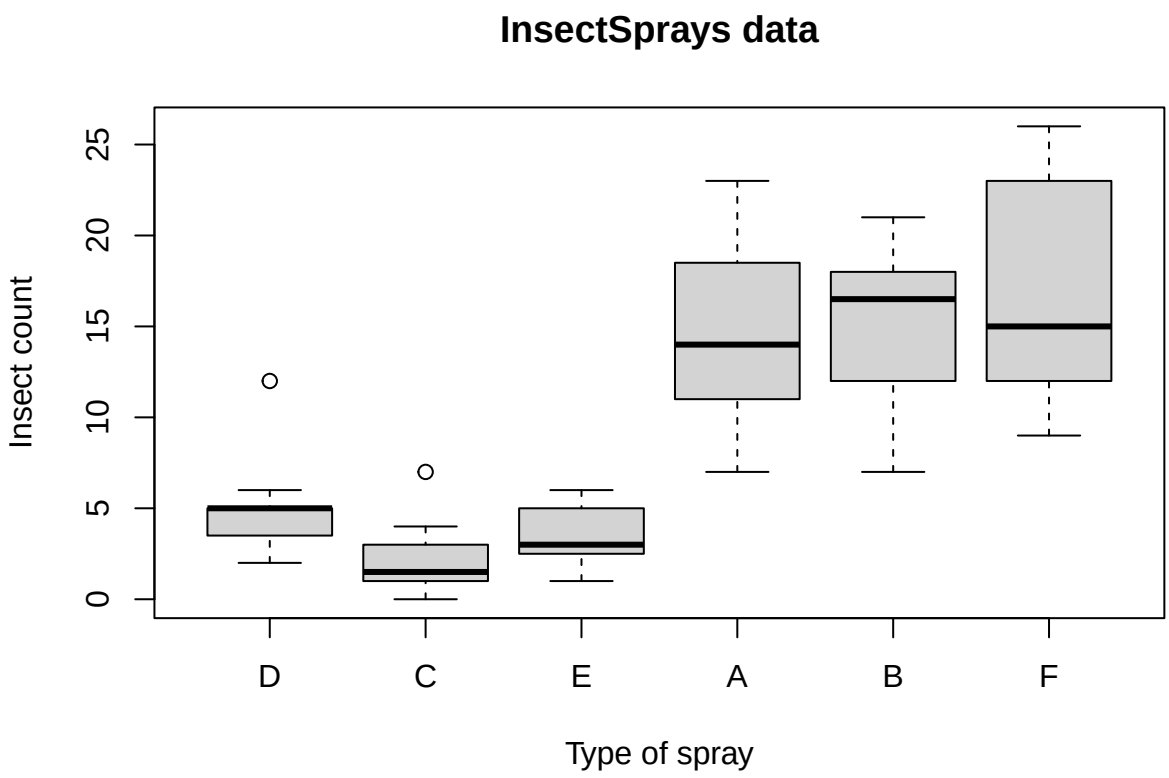



```
boxplot(count ~ spray, data=InsectSprays,  xlab = "Type of spray", ylab = "Insect count",
        main = "InsectSprays data", varwidth = TRUE, col = "lightgray")
```



In many situations, we are interested in setting one particular level of a factor variable as the reference level (e.g., the control). This can be done by using the function `relevel`.

```
InsectSprays$spray <- relevel(InsectSprays$spray, "D")
boxplot(count ~ spray, data=InsectSprays,
        xlab = "Type of spray",
        ylab = "Insect count",
        main = "InsectSprays data",
        varwidth = TRUE, col = "lightgray")
```



We just reset the spray type "D" as the reference level. It is now the first level.

6.1.1 Numeric Factors

It is useful in some cases to convert a numeric variable to a factor. For example, when site names are numeric numbers, we often convert them to factors. We also convert year into factors when year is used as a variable to divide data into groups. Once a numeric variable is converted into a factor, no numeric operation can be performed.

```
yr <- c(1980, 1980, 1967, 1966, 1967, 2001, 2001, 1966)
yr <- factor(yr, ordered=T)
```

The variable `yr` is now an ordered factor variable. No calculation can be performed:

```
yr[7]-yr[1]
```

```
## Warning in Ops.ordered(yr[7], yr[1]): '-' is not meaningful for ordered factors
```

```
## [1] NA
```

To return a numeric factor to its original numeric values, we cannot simply use the function `as.numeric`:

```
as.numeric(yr)
```

```
## [1] 3 3 2 1 2 4 4 1
```

It returns the internal integer values of the variable. We need to first convert an ordered factor to a simple character vector before applying the `as.numeric` function:

```
yr <- as.numeric(as.character(yr))
yr
```

```
## [1] 1980 1980 1967 1966 1967 2001 2001 1966
```

```
yr[7] - yr[1]
```

```
## [1] 21
```

This operation can be very useful in handling inadvertent non-numeric characters in a numeric variable. When reading the data file using `read.table` or `read.csv`, R takes the variable as factor.

```
temp <- factor(c(1,2,3, "n"))
as.numeric(as.character(temp))
```

```
## Warning: NAs introduced by coercion
```

```
## [1]  1  2  3 NA
```

The operation converts the errant characters into `NA`.

6.1.2 Manipulating Factors

When a factor is created, all of its levels are stored along with the factor, and if subsets of the factor are extracted, they will retain all of the original levels. This can create problems when constructing model matrices and may or may not be useful when displaying the data. Let's see an example:

```
letrs <- sample(letters, size = 100, replace = T)
 ## sampling 100 letters with replacement
letrs <- factor(letrs)
tb1<- table(letrs[1:10])
tb1
```

```
##
## a b c d e f g h i j k l m n o p q r s t u v w x y z
## 0 2 0 0 0 0 1 0 0 0 0 1 1 0 1 0 0 1 0 0 2 1 0 0 0 0
```

The levels (26 letters) are retained in the subset, even though only 8 of the 26 levels were represented in the subset. To drop the levels not included in the subset, we can either use `drop=TRUE` in the sub scripting operator:

```
letrs <- sample(letters, size = 100, replace = T)
letrs <- factor(letrs)
letrs.subset <- letrs[1:10, drop=TRUE]
table(letrs.subset)
```

```
## letrs.subset
## c d e l n p r s t z
## 1 1 1 1 1 1 1 1 1 1
```

or use the `factor` function again to create a new factor variable:

```
letrs.subset2 <- factor(letrs[1:10])
table(letrs.subset2)
```

```
## letrs.subset2
## c d e l n p r s t z
## 1 1 1 1 1 1 1 1 1 1
```

By default, when a factor variable is created using the function `factor`, missing values (`NA`) are excluded from factor levels. To create a factor that includes missing values from a numeric variable, use `exclude=NULL`.

When combining factor variables with different levels, we often encounter errors.

```
fact1 <- factor(sample(letters, size=10, replace=T))
fact2 <- factor(sample(letters, size=10, replace=T))
fact1
```

```
##  [1] n f h s y x z p y l
## Levels: f h l n p s x y z
```

```
fact2
```

```
##  [1] b x r k h n m r q p
## Levels: b h k m n p q r x
```

```
c(fact1, fact2)
```

```
##  [1] n f h s y x z p y l b x r k h n m r q p
## Levels: f h l n p s x y z b k m q r
```

The outcome is not what we wanted because level 1 in `fact1` is f and level 1 in `fact2` is b. As a result, the `1` in the first 10 elements is not the same as the `1` in the second 10 elements. In other words, the combined factors have more then the maximum level of the two factors. To combine two factor variables, we need to first return each of them into character vectors, combine them and convert the combined into factor:

```
fact1 <- levels(fact1)[fact1]
fact2 <- levels(fact2)[fact2]

fact12 <- factor(c(fact1, fact2))
```

6.1.3 Creating Factors from Continuous Variables

The function `cut` is used to convert a numeric variable into a factor. In the function, we use the argument `breaks =` to describe how ranges of numbers will be converted to factor values. When a number is provided (e.g., `breaks=10`), the resulting factor will be created by dividing the range of the variable into that number of equal-length intervals; if a vector is provided, the values of that vector is used to determine the break points.

Let's consider the data set `women` (reporting average heights and weights for American women).

```
wfact <- cut(women$weight, 4)
wfact
```

```
##  [1] (115,127] (115,127] (115,127] (115,127] (115,127] (127,140] (127,140]
##  [8] (127,140] (127,140] (140,152] (140,152] (140,152] (152,164] (152,164]
## [15] (152,164]
## Levels: (115,127] (127,140] (140,152] (152,164]
```

```
table(wfact)
```

```
## wfact
## (115,127] (127,140] (140,152] (152,164]
##         5         4         3         3
```

We may want to have breakpoints as some round values. The function `pretty` performs this function:

```
pretty(women$weight, 3)
```

It chooses round numbers as 1, 2, or 5 times a power of 10. In this case, the round number is 5 (or 5 times 10^0). We can specify the desired number of intervals (here 3). The function, however, finds the round number that will give the number of intervals closest to the number we want.

```
wfact <- cut(women$weight, pretty(women$weight, 3))
wfact
```

```
##  [1] (100,120] (100,120] (100,120] (120,140] (120,140] (120,140] (120,140]
##  [8] (120,140] (120,140] (140,160] (140,160] (140,160] (140,160] (140,160]
## [15] (160,180]
## Levels: (100,120] (120,140] (140,160] (160,180]
```

```
table(wfact)
```

```
## wfact
## (100,120] (120,140] (140,160] (160,180]
##         3         6         5         1
```

The `labels=` argument to the function `cut` allows us to specify the levels of the factors:

```
wfact <- cut(women$weight, 3, labels=c("Low","Medium","High"))
wfact
```

```
##  [1] Low    Low    Low    Low    Low    Low    Medium Medium Medium Medium
## [11] Medium High   High   High   High
## Levels: Low Medium High
```

```
table(wfact)
```

```
## wfact
##    Low Medium   High
##      6      5      4
```

To produce factors based on percentiles of the data, we can use the `quantile` function to generate the breakpoints:

```
wfact <- cut(women$weight, quantile(women$weight, (0:4)/4))
wfact
```

```
##  [1] <NA>      (115,124] (115,124] (115,124] (124,135] (124,135] (124,135]
##  [8] (124,135] (135,148] (135,148] (135,148] (148,164] (148,164] (148,164]
## [15] (148,164]
## Levels: (115,124] (124,135] (135,148] (148,164]
```

```
table(wfact)
```

```
## wfact
## (115,124] (124,135] (135,148] (148,164]
##         3         4         3         4
```

6.1.4 Factors Based on Dates and Times

We often want to create factors based on one of components of a date object (e.g., months, week days). We can extract that component from the date object and convert it into factor.

```
everyday <- seq(from =as.Date("2014-1-1"),
                to=as.Date("2014-12-31"),
                by="day")
## extracting month
mnth <- format(everyday, "%b")
months <- factor(mnth, ordered=T)
## not quite what we wanted
months <- factor(mnth, levels=month.abb, ordered=T)
```

The `cut` function understands time units of `month, days, weeks`, and `years` through the `breaks=` argument.

```
wks <- cut(everyday, breaks="week")
head(wks)
```

```
## [1] 2013-12-30 2013-12-30 2013-12-30 2013-12-30 2013-12-30 2014-01-06
## 53 Levels: 2013-12-30 2014-01-06 2014-01-13 2014-01-20 ... 2014-12-29
```

```
mnths1 <- cut(everyday, breaks="month")
mnths2 <- cut(everyday, breaks="month", labels=month.abb)
```

The `cut` function also takes multiples of units. For example, a quarter has three months:

```
qtrs <- cut(everyday, "3 months")
head(qtrs)
```

```
## [1] 2014-01-01 2014-01-01 2014-01-01 2014-01-01 2014-01-01 2014-01-01
## Levels: 2014-01-01 2014-04-01 2014-07-01 2014-10-01
```

6.1.5 Interactions

In statistical modeling, we often want to study the effects of two or more factor variables and their interactions. When studying the interaction effect of two factor variables, we need to create a combination factor. We can use the function `interaction`. Consider the data frame `CO2` on CO_2 uptake in grass plants. The two factor variables are `Type` and `Plant`.

```
data(CO2)
nlevels(CO2$Plant)
```

```
## [1] 12
```

```
nlevels(CO2$Type)
```

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

```
newfact <- interaction(CO2$Type, CO2$Plant)
nlevels(newfact)
```

```
## [1] 24
```

```
table(newfact)
```

```
## newfact
##      Quebec.Qn1 Mississippi.Qn1      Quebec.Qn2 Mississippi.Qn2      Quebec.Qn3
##               7               0               7               0               7
## Mississippi.Qn3      Quebec.Qc1 Mississippi.Qc1      Quebec.Qc3 Mississippi.Qc3
##               0               7               0               7               0
##      Quebec.Qc2 Mississippi.Qc2      Quebec.Mn3 Mississippi.Mn3      Quebec.Mn2
##               7               0               0               7               0
## Mississippi.Mn2      Quebec.Mn1 Mississippi.Mn1      Quebec.Mc2 Mississippi.Mc2
##               7               0               7               0               7
##      Quebec.Mc3 Mississippi.Mc3      Quebec.Mc1 Mississippi.Mc1
##               0               7               0               7
```

To remove combinations that never occur, we use `drop=TRUE`:

```
newfact <- interaction(CO2$Type, CO2$Plant, drop=TRUE)
nlevels(newfact)
```

```
## [1] 12
```

```
table(newfact)
```

```
## newfact
##      Quebec.Qn1      Quebec.Qn2      Quebec.Qn3      Quebec.Qc1      Quebec.Qc3
##               7               7               7               7               7
##      Quebec.Qc2 Mississippi.Mn3 Mississippi.Mn2 Mississippi.Mn1 Mississippi.Mc2
##               7               7               7               7               7
## Mississippi.Mc3 Mississippi.Mc1
##               7               7
```

```
newfact <- interaction(CO2$Type, CO2$Plant)
```

6.2 Character Manipulation

R is also good at manipulating characters. Because of R's factorized operations, R's character manipulating functions can be very powerful.

Here are some useful functions for characters.

To count the number characters in each element of a character vector, we use `nchar`.

```
state.name
```

```
##  [1] "Alabama"        "Alaska"         "Arizona"        "Arkansas"
##  [5] "California"     "Colorado"       "Connecticut"    "Delaware"
##  [9] "Florida"        "Georgia"        "Hawaii"         "Idaho"
## [13] "Illinois"       "Indiana"        "Iowa"           "Kansas"
## [17] "Kentucky"       "Louisiana"      "Maine"          "Maryland"
## [21] "Massachusetts"  "Michigan"       "Minnesota"      "Mississippi"
## [25] "Missouri"       "Montana"        "Nebraska"       "Nevada"
## [29] "New Hampshire"  "New Jersey"     "New Mexico"     "New York"
## [33] "North Carolina" "North Dakota"   "Ohio"           "Oklahoma"
## [37] "Oregon"         "Pennsylvania"   "Rhode Island"   "South Carolina"
## [41] "South Dakota"   "Tennessee"      "Texas"          "Utah"
## [45] "Vermont"        "Virginia"       "Washington"     "West Virginia"
## [49] "Wisconsin"      "Wyoming"
```

```
nchar(state.name)
```

```
##  [1]  7  6  7  8 10  8 11  8  7  7  6  5  8  7  4  6  8  9  5  8 13  8  9 11  8
## [26]  7  8  6 13 10 10  8 14 12  4  8  6 12 12 14 12  9  5  4  7  8 10 13  9  7
```

To display character strings, we use `cat` or `print`

```r
x <- 7
y <- 10
cat("x should be greater than y, but x = ", x, "and y = ",y, "\n")
```

```
## x should be greater than y, but x =  7 and y =  10
```

The `fill=` argument limits the number of character for each line:

```r
cat("Long strings can", "be displayed over",
    "several lines using","the fill = argument",
    fill=40)
```

```
## Long strings can be displayed over
## several lines using the fill = argument
```

The `cat` function also accepts `file=` argument so that we can save the text into a file.

Also we can use `cat` to concatenate character strings, the function `paste` is a more flexible one. When using `paste`, we can pass a number of objects and those that are not characters will be evaluated and the results converted into characters.

```r
paste("one", 1, "two", 2, "three", 3)
```

```
## [1] "one 1 two 2 three 3"
```

```r
paste("one", 1, "two", 2, "three", 3, sep="-")
```

```
## [1] "one-1-two-2-three-3"
```

When we have a vector of characters and we want to combine the elements into one character string, we use argument `collapse=`:

```r
paste(c("one", "two", "three"), collapse="-")
```

```
## [1] "one-two-three"
```

When multiple arguments are passed to `paste`, it will vectorize the operation, recycling shorter elements when necessary. This makes generating variable names easy:

```r
paste("X", 1:4, sep="")
```

```
## [1] "X1" "X2" "X3" "X4"
```

```r
paste(c("X","Y"), 1:5, sep="")
```

```
## [1] "X1" "Y2" "X3" "Y4" "X5"
```

The `sep=` argument controls what is placed between each set of values that are combined, and the `collapse=` argument can be used to specify a value to use when joining those individual values to create a single string:

```
paste(c("X","Y"), 1:5, sep="_", collapse="|")
```

```
## [1] "X_1|Y_2|X_3|Y_4|X_5"
```

```
paste(c("X", "Y"), 1:5, "^", c("A","B"),  sep="_", collapse="|")
```

```
## [1] "X_1_^_A|Y_2_^_B|X_3_^_A|Y_4_^_B|X_5_^_A"
```

Without the `collapse=` argument, the individual pasted pieces are returned separately.

```
st1 <- paste(c("X","Y"), 1:5, sep="_", collapse="|")
st2 <- paste(c("X","Y"), 1:5, sep="_")
length(st1)
```

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

```
length(st2)
```

```
## [1] 5
```

To work with parts of character values, we use the function `substring` to extract a subset of the character string. The function takes `first=` and `last=` as arguments, specifying the first and last characters.

```
substring(state.name, 2, 7)
```

```
##  [1] "labama" "laska"  "rizona" "rkansa" "alifor" "olorad" "onnect" "elawar"
##  [9] "lorida" "eorgia" "awaii"  "daho"   "llinoi" "ndiana" "owa"    "ansas"
## [17] "entuck" "ouisia" "aine"   "arylan" "assach" "ichiga" "inneso" "ississ"
## [25] "issour" "ontana" "ebrask" "evada"  "ew Ham" "ew Jer" "ew Mex" "ew Yor"
## [33] "orth C" "orth D" "hio"    "klahom" "regon"  "ennsyl" "hode I" "outh C"
## [41] "outh D" "enness" "exas"   "tah"    "ermont" "irgini" "ashing" "est Vi"
## [49] "iscons" "yoming"
```

The function returns as many as it finds with no padding provided.

We can also vectorizing the `first=` and `last=` arguments:

```
mystring <- "dog cat duck"
substring(mystring, c(1,5,9), c(3,7,12))
```

```
## [1] "dog"  "cat"  "duck"
```

For finding locations of particular characters within a character string, the string first needs to be converted to a character vector containing individual characters. This can be done by passing a vector consisting of all the characters to be processed as both the `first=` and `last=` arguments, and then applying `which` to the result:

```
state <- "Mississippi"
slength <- nchar(state)
lstr <- substring(state, 1:slength, 1:slength)
lstr
```

```
##  [1] "M" "i" "s" "s" "i" "s" "s" "i" "p" "p" "i"
```

```
which(lstr == "s")
```

```
## [1] 3 4 6 7
```

We can also change part of a character string by using `substring`.

```
mystring <- "dog cat duck"
substring(mystring, 5, 7) <- "feline"
mystring
mystring <- "dog cat duck"
substring(mystring, 5, 7) <- "d"
mystring
```

6.2.1 Regular Expressions in R

When working with a data frame with many variables we may want to extract columns with names matching a specific pattern. For example, land use data from the National Land use Land Cover Database used in the EUSE data are named `PNLCD` plus a number (representing the land use category). To specify all land use land cover variables, we can use something similar to a wildcard notation `PNLCD*`.

```
euse.ag <- read.csv(paste(dataDIR, "City_AG.csv", sep="/"), header=T)
euse.ag
```

```
##   CITY SUID     PNLCD7    PNLCD8   PNLCD78
## 1  BOS NECB  0.3375359  7.521565  7.859101
## 2  RAL ALBE  5.0754986 21.494498 26.569997
## 3  ATL ACFB  7.1979288 16.511416 23.709344
## 4  BIR MOBL  3.6328628 16.987809 20.620672
## 5  MGB WMIC  0.9334345 76.057259 76.990694
## 6  DEN SPLT 63.5333298 22.640489 86.173819
## 7  DFW TRIN 20.8222135 54.798380 75.620594
## 8  SLC GRSL  0.0000000  2.252252  2.252252
## 9  POR WILL  7.4810446 12.115522 19.596567
```

For this simple example, we can use the following lines to identify columns with names start with `PNLCD`:

```
substring(names(euse.ag), 1, 5)=="PNLCD"
```

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

```
## hence, the following works:
euse.ag[, substring(names(euse.ag), 1, 5)=="PNLCD"]
```

```
##       PNLCD7    PNLCD8   PNLCD78
## 1  0.3375359  7.521565  7.859101
## 2  5.0754986 21.494498 26.569997
## 3  7.1979288 16.511416 23.709344
## 4  3.6328628 16.987809 20.620672
## 5  0.9334345 76.057259 76.990694
## 6 63.5333298 22.640489 86.173819
## 7 20.8222135 54.798380 75.620594
## 8  0.0000000  2.252252  2.252252
## 9  7.4810446 12.115522 19.596567
```

In a more complicated data (e.g., community survey data with columns of species names), the function `substring` can be too restrictive. In computer programming, matching text to a specific pattern is carried out by using regular expressions.

Regular expressions are a method of expressing patterns in character values which can be used to extract parts of strings or to modify those strings in some way. You are probably familiar with wildcard notations such as `*.txt` to find all text files in a file manager. You can think of regular expressions as wildcards on steroids. You either love the stuff immediately or you need to use it for some reason. Either way, it is a geeky subject. But regular expression is often useful when we need to search text (e.g., variable names) that matches certain patterns.

Regular expressions are composed of three components: literal characters (matched by a single character string), character classes (matched by any of a number of characters), and modifiers (operate on the first two components).

Literal characters are the text or part of the text we want to find. As many punctuation marks are used as regular expression modifiers, to use them as a literal characters, we need to add a backslash in front of them to retain their literal meanings. For example, `*` is a wildcard notation `*.txt`, indicating any text. If we want to search `*` in the text, we need to tell the computer that the `*` is literal, using `\*`. The following characters must always be preceded by a backslash to retain their literal meaning:

```
. ^ $ + ? * ( ) [ ] { } | \
```

A character class is formed using square brackets (`[]`) surrounding the characters. For example, if we want to find either `a`, `b`, or `3`, we use `[ab3]`. Dash can be used inside of character classes to represent a range of values such as `[a-z]` or `[5-9]`. If a dash is to be literally included in a character class, it should be either the first character in the class or it should be preceded by a backslash.

Characters and character classes are the basic building blocks of regular expressions. These operators are:

Modifier	Meaning
`^`	anchors expression to beginning of target
`$`	anchors expression to end of target
`.`	matches any single character except newline
`|`	separates any single character except new line
`()`	groups of patterns together
`*`	matches 0 or more occurrences of preceding entity
`?`	matches 0 or 1 occurrence of preceding entity
`+`	matches 1 or more occurrences of preceding entity
`{n}`	matches exactly n occurrences of preceding entity
`{n,}`	matches at least n occurrences of preceding entity
`{n,m}`	matches between n and m occurrences

In the EUSE example, we want to extract columns with names start with `PNLCD` can be specified as `^PNLCD`. Using the function `grep`, we can compare the regular expression to the names of the EUSE data:

```
grep('^PNLCD', names(euse.ag))
```

```
## [1] 3 4 5
```

```
grep('^PNLCD', names(euse.ag), value=T)
```

```
## [1] "PNLCD7"  "PNLCD8"  "PNLCD78"
```

```
grep('[$0-9]', names(euse.ag))
```

```
## [1] 3 4 5
```

Some more complicated examples: - A string with two digits followed by one or more letters – `[0-9][0-9][a-zA-Z]+` - A string with three consecutive occurrences of `abc` – `(abc){3}` - A file name consists of all letters and ending with `.jpg` – `^[a-zA-z]+\\.jpg$` - A character string ends with one of the three animal names `chicken`, `duck`, and `pig` – `[a-zA-Z]+chicken|duck|pig$`

```
grep('[$0-9]{2}', names(euse.ag))
```

```
## [1] 5
```

```
grep('[$0-9]{2}', names(euse.ag), value = T)
```

```
## [1] "PNLCD78"
```

In one data set I worked with, one name of a location was entered as `name` and `name` (with an extra space). In another data set, the crop name was entered `soybean` by some and `soybeans` by others. R treats the two names in these two data sets as different sites or crops. Using regular expression, we can remove the extra character:

```
data$crop[grep('soybean.*', data$crop)] <- "soybeen"
```

Obviously, we can accomplish the same without using regular expression. The point is, using regular expression gives us much more flexibility.

Regular expressions are supported in the R functions `strsplit, grep, sub`, and `gsub`, as well as in the `regexpr` and `gregexpr` functions which are the main tools for working with regular expressions in R.

6.2.2 Breaking Apart Character Values

The `strsplit` function can use a character string or regular expression to divide up a character string into smaller pieces.

```
sentence <- "R is a free software environment for statistical computing."
parts <- strsplit(sentence, ' ')
parts
```

```
## [[1]]
## [1] "R"           "is"          "a"           "free"        "software"
## [6] "environment" "for"         "statistical" "computing."
```

The first argument of `strsplit` is the character string to break up, and the second argument is the character value or regular expression which should be used to break up the string into parts. It returns the results in a list. In the above example, we have one character string and the result is a list of 1.

```
more <- c(sentence, "It compiles and runs on a wide variety of UNIX platforms")
parts2 <- strsplit(more, " ")
length(parts2)
```

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

```
parts2
```

```
## [[1]]
## [1] "R"           "is"          "a"           "free"        "software"
## [6] "environment" "for"         "statistical" "computing."
##
## [[2]]
##  [1] "It"        "compiles"  "and"       "runs"      "on"        "a"
##  [7] "wide"      "variety"   "of"        "UNIX"      "platforms"
```

When the structure of the output is not important, we can use `unlist` to combine the output

```
allparts <- unlist(parts2)
allparts
```

```
##  [1] "R"           "is"          "a"           "free"        "software"
##  [6] "environment" "for"         "statistical" "computing."  "It"
## [11] "compiles"    "and"         "runs"        "on"          "a"
## [16] "wide"        "variety"     "of"          "UNIX"        "platforms"
```

6.2.3 Substitutions and Tagging

Using functions `sub` and `gsub` we can substitute text based on regular expressions. These two functions accept a regular expression, a string containing what will be substituted for the regular expression, and the string or strings to operate on. The `sub` function changes only the first occurrence of the regular expression, while the `gsub` function performs the substitution on all occurrences within the string.

An important use of these functions concerns numeric data, containing characters such as commas, dollar signs, or < signs. For example, we want to remove the dollar signs from the following vector:

```
values <- c("$11,123","$12,234")
gsub('[$,]',"", values)
```

```
## [1] "11123" "12234"
```

A powerful feature of substitution functions is known as tagging of regular expression. When part of a regular expression is surrounded by parentheses, that part can be used as a substitution pattern by representing it as a backslash followed by a number. The first tagged pattern is represented by `\\1`, the second by `\\2`, and so on. For example, in a data of drinking water quality, I have to deal with overwhelming amount of values below method detection limits. These limits change over time as measurement techniques improve. Almost all variables (chemicals) were read into R as characters because of values below detection limits, represented by a less than sign (`<`) plus the detection limit (e.g., `<0.01`). Because the detection limit changes, we can't replace it with a fixed value. To tag the value:

```
values <- c("0.2","0.3","<0.01")
gsub('<([0-9.]+)','\\1', values)
```

```
## [1] "0.2"  "0.3"  "0.01"
```

```
as.numeric(gsub('<([0-9.]+)','\\1', values))
```

```
## [1] 0.20 0.30 0.01
```

Here is another example. In finalcial data, a loss (negative number) is often entered as a number in parentheses.

```
values <- c("23.52", "(22.10)", "45.35")
```

To tag the number inside a pair of parentheses, we use `\\(([0-9.]+)\\)`. The part inside the unescaped parentheses is tagged as `\\1`:

```
gsub('\\(([0-9.]+)\\)', '-\\1', values)
```

```
## [1] "23.52"  "-22.10" "45.35"
```

6.3 Looping

'Looping', 'cycling', 'iterating' or just replicating instructions is quite an old practice that originated well before the invention of computers. It is nothing more than automating a certain multi-step process by organizing sequences of actions ('batch' processes) and grouping the parts in need of repetition. Even for 'calculating machines,' as computers were called in the pre-electronics era, pioneers like Ada Lovelace, Charles Babbage and others, devised methods to implement such iterations.

In modern – and not so modern – programming languages, where a program is a sequence of instructions, labeled or not, the loop is an evolution of the 'jump' instruction which asks the machine to jump to a predefined label along a sequence of instructions. The beloved and nowadays deprecated goto found in Assembler, Basic, Fortran and similar programming languages of that generation, was a means to tell the computer to jump to a specified instruction label: so, by placing that label before the location of the goto instruction, one obtained a repetition of the desired instruction a loop.

Yet, the control of the sequence of operations (the program flow control) would soon become cumbersome with the increase of goto and corresponding labels within a program. Specifically, one risked of loosing control of all the gotos that transferred control to a specific label (e.g. "from how many places did we get here"?) Then, to improve the clarity of programs, all modern programming languages were equipped with special constructs that allowed the repetition of instructions or blocks of instructions. In the present days, a number of variants exist: - Loops that execute for a prescribed number of times, as controlled by a counter

or an index, incremented at each iteration cycle; these pertain to the for family; - Loops based on the onset and verification of a logical condition (for example, the value of a control variable), tested at the start or at the end of the loop construct; these variants belong to the while or repeat family of loops, respectively.

Loops in R Every time some operation/s has to be repeated, a loop may come in handy. We only need to specify how many times or upon which conditions those operations need execution: we assign initial values to a control loop variable, perform the loop and then, once finished, we typically do something with the results.

But when are we supposed to use loops? Couldn't we just replicate the desired instruction for the sufficient number of times?

Well, our personal and arbitrary rule of thumb is that if you need to perform an action (say) three times or more, then a loop would serve you better; it makes the code more compact, readable and maintainable and you may save some typing. Say you discover that a certain instruction needs to be repeated once more than initially foreseen: instead of re-writing the full instruction, you may just alter the value of a variable in the test condition.

There are at least three loop constructs in R: - `for` loop - `while` loop - `repeat` loop

In all loop constructs, we have a number of operations (code) to be repeatedly run until a condition is met. ### The `for` loop In a `for` loop, the condition is presented in form of a vector. The most frequently used vector is `1:n`. The code `for (i in 1:n){...}` instructs R to run the code inside the curly brackets when `i` is part of the vector `1:n`. For example,

```
set.seed(101)
u1 <- rnorm(30)
usq <- 0
for (i in 1:10)
  {
  usq[i] <- u1[i] * u1[i]
  }
usq
```

```
##  [1] 0.10629979 0.30521410 0.45554919 0.04594998 0.09657751 1.37819684
##  [7] 0.38290089 0.01270903 0.84094088 0.04984474
```

```
i
```

```
## [1] 10
```

A bad example of using the `for` loop is to replace, e.g., -999.999, with NAs:

```
data <- c(rnorm(100), -999.999)
n <- length(data)
for (i in 1:n){
  if (data[i] == -999.999) data[i] <- NA
}
```

This is a bad example because we have simpler means to deal with the problem (`data <- ifelse(data== -999.999, NA, data)`) However, something similar to the "bad" `for` loop example can be very effective. In the Midterm exam, we need to replace missing values labeled using several different codings (-999.999, -99.99, -99) with NAs. Instead of going through columns of the data frame one at a time, we can use a `for` loop. Suppose that the data frame name is `df` and we need to through several columns:

```
cols <- c(3, 5:10)
for (i in cols){
  df[,i] <- ifelse(df[,i] == -999.999 | df[,i] == -99.99 | df[,i] == -99, NA, df[,i])
}
```

In this example, we know which culumns needs to be checked and use the `for` loop to go through them one at a time.
The `for` loop is by far the most popular and its construct implies that the number of iterations is fixed and known in advance. For example, conduct a Monte Carlo simulation with a predetermined number of iterations.

6.3.1 `while` and `repeat`

In a `for` loop, we know exactly how many iterations. In some cases, we want the loop to repeat until we meet conditions unknown to us before the looping starts. The `while` loop has the following construct:

```
while (condition){
...
}
```

The `condition` is evaluated first. If it is met, the code inside the curly brackets will not be executed. Otherwise, R evaluates code in the code block and reevaluates the condition. The process continues until the condition is met.

7 Week 9

7.1 Package `tidyr`

`tidyr` is a package that makes it easy to "tidy" your data. Tidy data is data that's easy to work with: it's easy to manage (with `dplyr`), visualise (with `ggplot2` or `ggvis`) and model (with R's hundreds of modelling packages). The two most important properties of tidy data are:

- Each column is a variable.
- Each row is an observation.

Arranging your data in this way makes it easier to work with because you have a consistent way of referring to variables (as column names) and observations (as row indices). When use tidy data and tidy tools, you spend less time worrying about how to feed the output from one function into the input of another, and more time answering your questions about the data.

To tidy messy data, you first identify the variables in your dataset, then use the tools provided by `tidyr` to move them into columns. `tidyr` provides three main functions for tidying your messy data: `gather()`, `separate()`, and `spread()`.

`gather()` takes multiple columns, and gathers them into key-value pairs: it makes "wide" data longer. Other names for `gather` include `melt` (reshape2), `pivot` (spreadsheets) and `fold` (databases). Here's an example how you might use `gather()` on a made-up dataset. In this experiment we've given three people two different drugs and recorded their heart rate:

```
messy <- data.frame(
  name = c("Wilbur", "Petunia", "Gregory"),
  a = c(67, 80, 64),
  b = c(56, 90, 50)
)
messy
```

```
##      name  a  b
## 1  Wilbur 67 56
## 2 Petunia 80 90
## 3 Gregory 64 50
```

We have three variables (`name`, `drug`, and `heartrate`), but only `name` is currently in a column. We use `gather()` to gather the `a` and `b` columns into key-value pairs of `drug` and `heartrate`:

```
tidy <- gather(messy, drugs, heartrate, -1)

## or

tidy <- messy %>%
              gather(drug, heartrate, a:b)
```

Sometimes two variables are clumped together in one column. `separate()` allows you to tease them apart (`extract()` works similarly but uses `regexp` groups instead of a splitting pattern or position). Take this example from stackoverflow (modified slightly for brevity). We have some measurements of how much time people spend on their phones, measured at two locations (work and home), at two times. Each person has been randomly assigned to either treatment or control.

```
set.seed(10)
messy <- data.frame(
  id = 1:4,
  trt = sample(rep(c('control', 'treatment'), each = 2)),
  work.T1 = runif(4),
  home.T1 = runif(4),
  work.T2 = runif(4),
  home.T2 = runif(4)
)
```

To tidy this data, we first use `gather()` to turn columns `work.T1`, `home.T1`, `work.T2` and `home.T2` into a key-value pair of key and time. (Only the first eight rows are shown to save space.)

```
tidier <- messy %>%
  gather(key, time, -id, -trt)
tidier %>% head(8)
```

```
##   id       trt     key       time
## 1  1 treatment work.T1 0.08513597
## 2  2   control work.T1 0.22543662
## 3  3   control work.T1 0.27453052
## 4  4 treatment work.T1 0.27230507
## 5  1 treatment home.T1 0.61582931
```

```
## 6  2   control home.T1 0.42967153
## 7  3   control home.T1 0.65165567
## 8  4 treatment home.T1 0.56773775
```

Next we use `separate()` to split the key into location and time, using a regular expression to describe the character that separates them.

```
tidy <- tidier %>%
  separate(key, into = c("location", "time"), sep = "\\.")
tidy %>% head(8)
```

```
##   id       trt location time
## 1  1 treatment     work   T1
## 2  2   control     work   T1
## 3  3   control     work   T1
## 4  4 treatment     work   T1
## 5  1 treatment     home   T1
## 6  2   control     home   T1
## 7  3   control     home   T1
## 8  4 treatment     home   T1
```

The last tool, `spread()`, takes two columns (a key-value pair) and spreads them in to multiple columns, making "long" data wider. Spread is known by other names in other places: it's `cast` in `reshape2`, `unpivot` in spreadsheets and `unfold` in databases. `spread()` is used when you have variables that form rows instead of columns. You need `spread()` less frequently than `gather()` or `separate()` so to learn more, check out the documentation and the demos.

Just as `reshape2` did less than `reshape`, `tidyr` does less than `reshape2`. It's designed specifically for tidying data, not general reshaping. In particular, existing methods only work for data frames, and `tidyr` never aggregates. This makes each function in `tidyr` simpler: each function does one thing well.

You can learn more about the underlying principles in the tidy data paper by Wickham. To see more examples of data tidying, read the vignette, `vignette("tidy-data")`, or check out the demos, `demo(package = "tidyr")`.

7.2 Reshape: Long versus Wide

Wide data has a column for each variable. For example, this is wide-format data:

```
#   ozone   wind  temp
# 1 23.62 11.623 65.55
# 2 29.44 10.267 79.10
# 3 59.12  8.942 83.90
# 4 59.96  8.794 83.97
```

And this is long-format data:

```
#    variable  value
# 1     ozone 23.615
# 2     ozone 29.444
# 3     ozone 59.115
# 4     ozone 59.962
```

```
# 5      wind 11.623
# 6      wind 10.267
# 7      wind  8.942
# 8      wind  8.794
# 9      temp 65.548
# 10     temp 79.100
# 11     temp 83.903
# 12     temp 83.968
```

Long-format data has a column for possible variable types and a column for the values of those variables. Long-format data isn't necessarily only two columns. For example, we might have ozone measurements for each day of the year. In that case, we could have another column for day. In other words, there are different levels of "longness". The ultimate shape you want to get your data into will depend on what you are doing with it.

It turns out that you need wide-format data for some types of data analysis and long-format data for others. In reality, you need long-format data much more commonly than wide-format data. For example, `ggplot2` requires long-format data (technically tidy data), `plyr` requires long-format data, and most modelling functions (such as `lm()`, `glm()`, and `gam()`) require long-format data. But people often find it easier to record their data in wide format.

7.2.1 Wide- to long-format data: the `melt` function

For this example we'll work with the `airquality` dataset that is built into R. First we'll change the column names to lower case to make them easier to work with. Then we'll look at the data:

```
names(airquality) <- tolower(names(airquality))
head(airquality)
```

```
##   ozone solar.r wind temp month day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

What happens if we run the function `melt` with all the default argument values?

```
aql <- melt(airquality) # [a]ir [q]uality [l]ong format
```

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

```
head(aql)
```

```
##   variable value
## 1    ozone    41
## 2    ozone    36
## 3    ozone    12
## 4    ozone    18
## 5    ozone    NA
## 6    ozone    28
```

```
tail(aql)
```

```
##     variable value
## 913      day    25
## 914      day    26
## 915      day    27
## 916      day    28
## 917      day    29
## 918      day    30
```

By default, `melt` has assumed that all columns with numeric values are variables with values. Often this is what you want. Maybe here we want to know the values of `ozone`, `solar.r`, `wind`, and `temp` for each month and day. We can do that with `melt` by telling it that we want `month` and `day` to be "ID variables." ID variables are the variables that identify individual rows of data.

```
aql <- melt(airquality, id.vars = c("month", "day"))
head(aql)
```

```
##   month day variable value
## 1     5   1    ozone    41
## 2     5   2    ozone    36
## 3     5   3    ozone    12
## 4     5   4    ozone    18
## 5     5   5    ozone    NA
## 6     5   6    ozone    28
```

What if we wanted to control the column names in our long-format data? `melt` lets us set those too all in one step:

```
aql <- melt(airquality, id.vars = c("month", "day"),
  variable.name = "climate_variable",
  value.name = "climate_value")
head(aql)
```

```
##   month day variable value
## 1     5   1    ozone    41
## 2     5   2    ozone    36
## 3     5   3    ozone    12
## 4     5   4    ozone    18
## 5     5   5    ozone    NA
## 6     5   6    ozone    28
```

7.2.2 Long- to wide-format data: the cast functions

Whereas going from wide- to long-format data is pretty straightforward, going from long- to wide-format data can take a bit more thought. It usually involves some head scratching and some trial and error for all but the simplest cases. Let's go through some examples.

In `reshape2` there are multiple `cast` functions. Since you will most commonly work with `data.frame` objects, we'll explore the `dcast` function. (There is also `acast` to return a vector, matrix, or array.)

Let's take the long-format `airquality` data and cast it into some different wide formats. To start with, we'll recover the same format we started with and compare the two.

`dcast` uses a formula to describe the shape of the data. The arguments on the left refer to the ID variables and the arguments on the right refer to the measured variables. Coming up with the right formula can take some trial and error at first. So, if you're stuck don't feel bad about just experimenting with formulas. There are usually only so many ways you can write the formula.

Here, we need to tell `dcast` that `month` and `day` are the ID variables (we want a column for each) and that variable describes the measured variables. Since there is only one remaining column, `dcast` will figure out that it contains the values themselves. We could explicitly declare this with `value.var`. (And in some cases it will be necessary to do so.)

```
aql <- melt(airquality, id.vars = c("month", "day"))
aqw <- dcast(aql, month + day ~ variable)
head(aqw)
```

```
##   month day ozone solar.r wind temp
## 1     5   1    41     190  7.4   67
## 2     5   2    36     118  8.0   72
## 3     5   3    12     149 12.6   74
## 4     5   4    18     313 11.5   62
## 5     5   5    NA      NA 14.3   56
## 6     5   6    28      NA 14.9   66
```

```
head(airquality) # original data
```

```
##   ozone solar.r wind temp month day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

So, besides re-arranging the columns, we've recovered our original data.

One confusing "mistake" you might make is casting a dataset in which there is more than one value per data cell. For example, this time we won't include day as an ID variable:

```
dcast(aql, month ~ variable)
```

```
## Aggregation function missing: defaulting to length
```

```
##   month ozone solar.r wind temp
## 1     5    31      31   31   31
## 2     6    30      30   30   30
## 3     7    31      31   31   31
## 4     8    31      31   31   31
## 5     9    30      30   30   30
```

When you run this in R, you'll notice the warning message:

```
> # Aggregation function missing: defaulting to length
```

And if you look at the output, the cells are filled with the number of data rows for each month-climate combination. The numbers we're seeing are the number of days recorded in each month. When you cast your data and there are multiple values per cell, you also need to tell `dcast` how to aggregate the data. For example, maybe you want to take the mean, or the median, or the sum. Let's try the last example, but this time we'll take the mean of the climate values. We'll also pass the option `na.rm = TRUE` through the `...` argument to remove NA values. (The `...` let's you pass on additional arguments to your `fun.aggregate` function, here `mean`.)

```
dcast(aql, month ~ variable, fun.aggregate = mean, na.rm = TRUE)
```

```
##   month    ozone  solar.r      wind     temp
## 1     5 23.61538 181.2963 11.622581 65.54839
## 2     6 29.44444 190.1667 10.266667 79.10000
## 3     7 59.11538 216.4839  8.941935 83.90323
## 4     8 59.96154 171.8571  8.793548 83.96774
## 5     9 31.44828 167.4333 10.180000 76.90000
```

Unlike `melt`, there are some other fancy things you can do with `dcast` that I'm not covering here. It's worth reading the help file `?dcast`. For example, you can compute summaries for rows and columns, subset the columns, and fill in missing cells in one call to `dcast`.

7.3 `tidyr` versus `reshape2`

What is the difference between the two packages `tidyr` and `reshape2`? Let us examine where their purposes overlap and where they differ by comparing the functions `gather()`, `separate()`, and `spread()`, from `tidyr`, with the functions `melt()`, `colsplit()`, and `dcast()`, from `reshape2`.

7.3.1 Data tidying

Data tidying is the operation of transforming data into a clear and simple form that makes it easy to work with. "Tidy data" represent the information from a dataset as data frames where each row is an observation and each column contains the values of a variable (i.e. an attribute of what we are observing). Compare the two data frames below to get an idea of the differences: `example.tidy` is the tidy version of `example.messy`, the same information is organized in two different ways.

```
example.messy
##              treatmenta treatmentb
## John Smith           NA          2
## Jane Doe             16         11
## Mary Johnson          3          1
example.tidy
##           name trt result
## 1   John Smith   a     NA
## 2     Jane Doe   a     16
## 3 Mary Johnson   a      3
## 4   John Smith   b      2
## 5     Jane Doe   b     11
## 6 Mary Johnson   b      1
```

7.3.2 From the wide to the long format: `gather()` vs `melt()`

We now begin by reviewing how we can bring data from the "wide" to the "long" format.

Let's start loading some data: we have measurements of how much time people spend on their phones, measured at two locations (work and home), at two times. Each person has been randomly assigned to either treatment or control.

```
set.seed(10)
messy <- data.frame(id = 1:4,
                    trt = sample(rep(c('control', 'treatment'), each = 2)),
                    work.T1 = runif(4),
                    home.T1 = runif(4),
                    work.T2 = runif(4),
                    home.T2 = runif(4))
messy
```

```
##   id       trt    work.T1   home.T1   work.T2    home.T2
## 1  1 treatment 0.08513597 0.6158293 0.1135090 0.05190332
## 2  2   control 0.22543662 0.4296715 0.5959253 0.26417767
## 3  3   control 0.27453052 0.6516557 0.3580500 0.39879073
## 4  4 treatment 0.27230507 0.5677378 0.4288094 0.83613414
```

Our first step is to put the data in the tidy format, to do that we use `tidyr`'s functions `gather()` and `separate()`. Following Wickham's tidy data definition, this data frame is not tidy because some variable values are in the column names. We bring this messy data frame from the wide to the long format by using the `gather()` function. We want to gather all the columns, except for the `id` and `trt` ones, in two columns key and value:

```
gathered.messy <- gather(messy, key="variable", value="value", -id, -trt)
head(gathered.messy)
```

```
##   id       trt variable      value
## 1  1 treatment  work.T1 0.08513597
## 2  2   control  work.T1 0.22543662
## 3  3   control  work.T1 0.27453052
## 4  4 treatment  work.T1 0.27230507
## 5  1 treatment  home.T1 0.61582931
## 6  2   control  home.T1 0.42967153
```

Note that in `gather()` we used bare variable names to specify the names of the key, value, `id` and `trt` columns.

We can get the same result with the `melt()` function from `reshape2`:

```
molten.messy <- melt(messy,
                     variable.name = "variable",
                     value.names = "value",
                     id.vars = c("id", "trt"))
head(molten.messy)
```

```
##   id       trt variable      value
## 1  1 treatment  work.T1 0.08513597
```

```
## 2  2   control  work.T1 0.22543662
## 3  3   control  work.T1 0.27453052
## 4  4 treatment  work.T1 0.27230507
## 5  1 treatment  home.T1 0.61582931
## 6  2   control  home.T1 0.42967153
```

We now compare the two functions by running them over the data without any further parameter and see what happen:

```
head(gather(messy))
```

```
##   key     value
## 1  id         1
## 2  id         2
## 3  id         3
## 4  id         4
## 5 trt treatment
## 6 trt   control
```

```
head(melt(messy))
```

```
## Using trt as id variables
```

```
##         trt variable      value
## 1 treatment       id 1.00000000
## 2   control       id 2.00000000
## 3   control       id 3.00000000
## 4 treatment       id 4.00000000
## 5 treatment  work.T1 0.08513597
## 6   control  work.T1 0.22543662
```

We see a different behaviour: `gather()` has brought messy into a long data format with a warning by treating all columns as variable, while `melt()` has treated `trt` as an "id variables". Id columns are the columns that contain the identifier of the observation that is represented as a row in our data set. Indeed, if `melt()` does not receive any `id.variables` specification, then it will use the factor or character columns as id variables. `gather()` requires the columns that needs to be treated as ids to be removed, all the other columns are going to be used as key-value pairs.

Despite those last different results, we have seen that the two functions can be used to perform the exactly same operations on data frames, and only on data frames! Indeed, `gather()` cannot handle matrices or arrays, while `melt()` can as shown below.

```
set.seed(3)
M <- matrix(rnorm(6), ncol = 3)
dimnames(M) <- list(letters[1:2], letters[1:3])
melt(M)
```

```
## Warning in type.convert.default(X[[i]], ...): 'as.is' should be specified by the
## caller; using TRUE
```

```
## Warning in type.convert.default(X[[i]], ...): 'as.is' should be specified by the
## caller; using TRUE
```

```
##   X1 X2       value
## 1  a  a -0.96193342
## 2  b  a -0.29252572
## 3  a  b  0.25878822
## 4  b  b -1.15213189
## 5  a  c  0.19578283
## 6  b  c  0.03012394

> gather(M)
Error in UseMethod("gather_") :
  no applicable method for 'gather_' applied to an object of class "c('matrix', 'double', 'numeric')"
>
```

7.3.3 Split a column: `separate()` vs `colsplit()`

Our next step is to split the column key into two different columns in order to separate the location and time variables and obtain a tidy data frame:

```
tidy <- separate(gathered.messy,
                 variable, into = c("location", "time"), sep = "\\.")
res.tidy <- cbind(molten.messy[1:2],
                  colsplit(molten.messy[, 3], "\\.", c("location", "time")),
                  molten.messy[4])
```

```
## Warning in type.convert.default(as.character(x)): 'as.is' should be specified by
## the caller; using TRUE

## Warning in type.convert.default(as.character(x)): 'as.is' should be specified by
## the caller; using TRUE
```

```
head(tidy)
```

```
##   id       trt location time      value
## 1  1 treatment     work   T1 0.08513597
## 2  2   control     work   T1 0.22543662
## 3  3   control     work   T1 0.27453052
## 4  4 treatment     work   T1 0.27230507
## 5  1 treatment     home   T1 0.61582931
## 6  2   control     home   T1 0.42967153
```

```
head(res.tidy)
```

```
##   id       trt location time      value
## 1  1 treatment     work   T1 0.08513597
## 2  2   control     work   T1 0.22543662
## 3  3   control     work   T1 0.27453052
## 4  4 treatment     work   T1 0.27230507
## 5  1 treatment     home   T1 0.61582931
## 6  2   control     home   T1 0.42967153
```

Again, the result is the same but we need a workaround: because `colsplit()` operates only on a single column we use `cbind()` to insert the new two columns in the data frame. `separate()` performs all the operation at once reducing the possibility of making mistakes.

7.3.4 From the long to the wide format: `spread()` vs `dcast()`

Finally, we compare `spread()` with `dcast()` using the data frame example for the `spread()` documentation itself. Briefly, `spread()` is complementary to `gather()` and brings data from the long to the wide format.

```
set.seed(14)
stocks <- data.frame(time = as.Date('2018-01-01') + 0:9,
                     X = rnorm(10, 0, 1),
                     Y = rnorm(10, 0, 2),
                     Z = rnorm(10, 0, 4))
stocksm <- gather(stocks, stock, price, -time)
spread.stock <- spread(stocksm, stock, price)
head(spread.stock)
```

```
##         time           X          Y          Z
## 1 2018-01-01 -0.66184983 -0.7656438 -5.0672590
## 2 2018-01-02  1.71895416  0.5988432 -0.7943331
## 3 2018-01-03  2.12166699  1.3484795  0.5554631
## 4 2018-01-04  1.49715368 -0.5856326 -1.1173440
## 5 2018-01-05 -0.03614058  0.9761067  2.8356777
## 6 2018-01-06  1.23194518  1.7656036 -3.0664418
```

```
cast.stock <- dcast(stocksm, formula = time ~ stock, value.var = "price")
head(cast.stock)
```

```
##         time           X          Y          Z
## 1 2018-01-01 -0.66184983 -0.7656438 -5.0672590
## 2 2018-01-02  1.71895416  0.5988432 -0.7943331
## 3 2018-01-03  2.12166699  1.3484795  0.5554631
## 4 2018-01-04  1.49715368 -0.5856326 -1.1173440
## 5 2018-01-05 -0.03614058  0.9761067  2.8356777
## 6 2018-01-06  1.23194518  1.7656036 -3.0664418
```

Again, the same result produced by spread() can be obtained using dcast() by specifying the correct formula.

In the next session, we are going to modify the formula parameter in order to perform some data aggregation and compare further the two packages.

7.4 Data aggregation

Up to now we made `reshape2` following `tidyr`, showing that everything you can do with `tidyr` can be achieved by `reshape2`, too, at the price of a some workarounds. As we now go on with our simple example we will get out of the purposes of `tidyr` and have no more functions available for our needs. Now we have a tidy data set – one observation per row and one variable per column – to work with. We show some aggregations that are possible with `dcast()` using the `tips` data frame from `reshape2`. `tips` contains the information one waiter recorded about each tip he received over a period of a few months working in one restaurant.

```
head(tips)
```

```
##   total_bill  tip    sex smoker day   time size
```

```
## 1      16.99 1.01 Female     No Sun Dinner    2
## 2      10.34 1.66   Male     No Sun Dinner    3
## 3      21.01 3.50   Male     No Sun Dinner    3
## 4      23.68 3.31   Male     No Sun Dinner    2
## 5      24.59 3.61 Female     No Sun Dinner    4
## 6      25.29 4.71   Male     No Sun Dinner    4
```

```
m.tips <- melt(tips)
```

```
## Using sex, smoker, day, time as id variables
```

```
## Using sex, smoker, day, time as id variables
head(m.tips)
```

```
##      sex smoker day   time   variable value
## 1 Female     No Sun Dinner total_bill 16.99
## 2   Male     No Sun Dinner total_bill 10.34
## 3   Male     No Sun Dinner total_bill 21.01
## 4   Male     No Sun Dinner total_bill 23.68
## 5 Female     No Sun Dinner total_bill 24.59
## 6   Male     No Sun Dinner total_bill 25.29
```

We use `dcast()` to get information on the average total bill, tip and group size per day and time:

```
dcast(m.tips, day+time ~ variable, mean)
```

```
##    day   time total_bill      tip     size
## 1  Fri Dinner   19.66333 2.940000 2.166667
## 2  Fri  Lunch   12.84571 2.382857 2.000000
## 3  Sat Dinner   20.44138 2.993103 2.517241
## 4  Sun Dinner   21.41000 3.255132 2.842105
## 5 Thur Dinner   18.78000 3.000000 2.000000
## 6 Thur  Lunch   17.66475 2.767705 2.459016
```

Averages per smoker or not in the group.

```
dcast(m.tips, smoker ~ variable, mean)
```

```
##   smoker total_bill      tip     size
## 1     No   19.18828 2.991854 2.668874
## 2    Yes   20.75634 3.008710 2.408602
```

There is no function in the `tidyr` package that allows us to perform a similar operation, the reason is that `tidyr` is designed only for data tidying and not for data reshaping.

8 Week 10: dplyr

```
## Warning in library(package, lib.loc = lib.loc, character.only = TRUE,
## logical.return = TRUE, : there is no package called 'WHO_0.2.1.tar.gz'
```

```
## Warning in library(package, lib.loc = lib.loc, character.only = TRUE,
## logical.return = TRUE, : there is no package called 'WHO_0.2.1.tar.gz'
```

When working with data you must:

- Figure out what you want to do.
- Describe those tasks in the form of a computer program.
- Execute the program.

The `dplyr` package makes these steps fast and easy:

- By constraining your options, it simplifies how you can think about common data manipulation tasks.
- It provides simple "verbs", functions that correspond to the most common data manipulation tasks, to help you translate those thoughts into code.
- It uses efficient data storage backends, so you spend less time waiting for the computer.

This week, we learn `dplyr`'s basic set of tools, and how to apply them to data frames.

By "constraining" our options, I mean that we often use the "split-apply-combine" strategy for data management. Splitting a large data into pieces, upon which, we can carry out operations easily, and the results are then combined. This strategy is useful when we need to carry out some operation repeatedly on subsets of the data. This need can appear in all stages of a data analysis problem. For example, - When preparing data, we often need to perform group-wise ranking, standardization, normalization, detrending, or creating new variables that are easily calculated on a per-group basis. - When creating summaries. - In modeling or data analysis when we need to fit models for each group.

Using loops is one way of performing group-wise operations. However, code for looping is difficult to read and prone to error. In R, looping is also slow. R provides a number of functions (e.g., `apply`) for group-wise computation without using looping. However, these functions are limited to simple operations. The R package `dplyr` is written and maintained by Hadley Wickham. It provides some great, easy-to-use functions that are very handy when performing exploratory data analysis and manipulation. The idea behind the package is the data analysis strategy of "split-apply-combine," which divides a large data frame into pieces, apply the necessary operation to individual pieces, and combine the results to produce the desired data.

Before going in to the details, let's use the data `airquality` to see how the package works.

The head of the dataset looks like this:

```
head(airquality)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

`dplyr` can work with data frames as is, but if you're dealing with large data, it's worthwhile to convert them to a `tbl_df`:

```
airquality <- tbl_df(airquality)
```

```
## Warning: `tbl_df()` was deprecated in dplyr 1.0.0.
## Please use `tibble::as_tibble()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was generated.
```

`tbl_df` is a wrapper around a data frame that won't accidentally print a lot of data to the screen.

8.1 Filter

The `filter` function will return all the rows that satisfy a following condition. For example below will return all the rows where Temp is larger than 70.

```
filter(airquality, Temp > 70)
```

```
## # A tibble: 120 x 6
##    Ozone Solar.R  Wind  Temp Month   Day
##    <int>   <int> <dbl> <int> <int> <int>
##  1    36     118   8      72     5     2
##  2    12     149  12.6    74     5     3
##  3     7      NA   6.9    74     5    11
##  4    11     320  16.6    73     5    22
##  5    45     252  14.9    81     5    29
##  6   115     223   5.7    79     5    30
##  7    37     279   7.4    76     5    31
##  8    NA     286   8.6    78     6     1
##  9    NA     287   9.7    74     6     2
## 10    NA     186   9.2    84     6     4
## # ... with 110 more rows
```

Another example of filter is to return all the rows where Temp is larger than 80 and Month is after May.

```
filter(airquality, Temp > 80 & Month > 5)
```

```
## # A tibble: 67 x 6
##    Ozone Solar.R  Wind  Temp Month   Day
##    <int>   <int> <dbl> <int> <int> <int>
##  1    NA     186   9.2    84     6     4
##  2    NA     220   8.6    85     6     5
##  3    29     127   9.7    82     6     7
##  4    NA     273   6.9    87     6     8
##  5    71     291  13.8    90     6     9
##  6    39     323  11.5    87     6    10
##  7    NA     259  10.9    93     6    11
##  8    NA     250   9.2    92     6    12
##  9    23     148   8      82     6    13
## 10    NA     138   8      83     6    30
## # ... with 57 more rows
```

8.2 Mutate

`mutate` is used to add new variables to the data. For example lets adds a new column that displays the temperature in Celsius.

```
mutate(airquality, TempInC = (Temp - 32) * 5 / 9)
```

```
## # A tibble: 153 x 7
##    Ozone Solar.R  Wind  Temp Month   Day TempInC
##    <int>   <int> <dbl> <int> <int> <int>   <dbl>
```

```
##  1    41     190   7.4    67     5     1    19.4
##  2    36     118   8      72     5     2    22.2
##  3    12     149  12.6    74     5     3    23.3
##  4    18     313  11.5    62     5     4    16.7
##  5    NA      NA  14.3    56     5     5    13.3
##  6    28      NA  14.9    66     5     6    18.9
##  7    23     299   8.6    65     5     7    18.3
##  8    19      99  13.8    59     5     8    15
##  9     8      19  20.1    61     5     9    16.1
## 10    NA     194   8.6    69     5    10    20.6
## # ... with 143 more rows
```

We can also write our own function to make the transformation. This allows us to do more complicated data transformation.

```
FtoC <- function(x) (x-32)*5/9
mutate(airquality, TempInC=FtoC(Temp))
```

```
## # A tibble: 153 x 7
##    Ozone Solar.R  Wind  Temp Month   Day TempInC
##    <int>   <int> <dbl> <int> <int> <int>   <dbl>
##  1    41     190   7.4    67     5     1    19.4
##  2    36     118   8      72     5     2    22.2
##  3    12     149  12.6    74     5     3    23.3
##  4    18     313  11.5    62     5     4    16.7
##  5    NA      NA  14.3    56     5     5    13.3
##  6    28      NA  14.9    66     5     6    18.9
##  7    23     299   8.6    65     5     7    18.3
##  8    19      99  13.8    59     5     8    15
##  9     8      19  20.1    61     5     9    16.1
## 10    NA     194   8.6    69     5    10    20.6
## # ... with 143 more rows
```

8.3 Summarise

The `summarise` function is used to summarise multiple values into a single value. It is very powerful when used in conjunction with the other functions in the `dplyr` package, as demonstrated below. `na.rm = TRUE` will remove all NA values while calculating the mean, so that it doesn't produce spurious results.

```
summarise(airquality, mean(Temp, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   `mean(Temp, na.rm = TRUE)`
##                        <dbl>
## 1                       77.9
```

8.4 Group By

The `group_by` function is used to group data by one or more variables. Will group the data together based on the Month, and then the `summarise` function is used to calculate the mean temperature in each month.

```
summarise(group_by(airquality, Month), mean(Temp, na.rm = TRUE))
```

```
## # A tibble: 5 x 2
##   Month `mean(Temp, na.rm = TRUE)`
##   <int>                      <dbl>
## 1     5                       65.5
## 2     6                       79.1
## 3     7                       83.9
## 4     8                       84.0
## 5     9                       76.9
```

8.5 Sample

The `sample` function is used to select random rows from a table. The first line of code randomly selects ten rows from the dataset, and the second line of code randomly selects 15 rows (10% of the original 153 rows) from the dataset.

```
sample_n(airquality, size = 10)
```

```
## # A tibble: 10 x 6
##    Ozone Solar.R  Wind  Temp Month   Day
##    <int>   <int> <dbl> <int> <int> <int>
##  1    NA      31  14.9    77     6    29
##  2    NA     101  10.9    84     7     4
##  3    NA     255  12.6    75     8    23
##  4    NA     332  13.8    80     6    14
##  5    19      99  13.8    59     5     8
##  6    78      NA   6.9    86     8     4
##  7    47      95   7.4    87     9     5
##  8    59      51   6.3    79     8    17
##  9    22      71  10.3    77     8    16
## 10    80     294   8.6    86     7    24
```

```
sample_frac(airquality, size = 0.1)
```

```
## # A tibble: 15 x 6
##    Ozone Solar.R  Wind  Temp Month   Day
##    <int>   <int> <dbl> <int> <int> <int>
##  1    NA     138   8      83     6    30
##  2     9      24  10.9    71     9    14
##  3    36     118   8      72     5     2
##  4    NA      47  10.3    73     6    27
##  5    35     274  10.3    82     7    17
##  6   168     238   3.4    81     8    25
##  7    59      51   6.3    79     8    17
##  8    84     237   6.3    96     8    30
##  9    20     252  10.9    80     9     7
## 10    65     157   9.7    80     8    14
## 11    NA     264  14.3    79     6     6
## 12    37     279   7.4    76     5    31
## 13    NA     322  11.5    79     6    15
```

```
## 14    NA      64  11.5    79     8    15
## 15    30     322  11.5    68     5    19
```

This functionality is useful for resampling-based analysis (e.g., bootstrapping).

8.6 Count

The `count` function tallies observations based on a group. It is slightly similar to the table function in the base package. For example:

```
count(airquality, Month)
```

```
## # A tibble: 5 x 2
##   Month     n
##   <int> <int>
## 1     5    31
## 2     6    30
## 3     7    31
## 4     8    31
## 5     9    30
```

This means that there are 31 rows with Month = 5, 30 rows with Month = 6, and so on.

8.7 Arrange

The `arrange` function is used to arrange rows by variables. Currently, the airquality dataset is arranged based on Month, and then Day. We can use the arrange function to arrange the rows in the descending order of Month, and then in the ascending order of Day.

```
arrange(airquality, desc(Month), Day)
```

```
## # A tibble: 153 x 6
##    Ozone Solar.R  Wind  Temp Month   Day
##    <int>   <int> <dbl> <int> <int> <int>
##  1    96     167   6.9    91     9     1
##  2    78     197   5.1    92     9     2
##  3    73     183   2.8    93     9     3
##  4    91     189   4.6    93     9     4
##  5    47      95   7.4    87     9     5
##  6    32      92  15.5    84     9     6
##  7    20     252  10.9    80     9     7
##  8    23     220  10.3    78     9     8
##  9    21     230  10.9    75     9     9
## 10    24     259   9.7    73     9    10
## # ... with 143 more rows
```

8.8 Pipe

The pipe or chain operator, represented by `%>%` can be used to chain code together. It is very useful when you are performing several operations on data, and don't want to save the output at each intermediate step.

For example, let's say we want to remove all the data corresponding to Month = 5, group the data by month, and then find the mean of the temperature each month. The conventional way to write the code for this would be:

```
filteredData <- filter(airquality, Month != 5)
groupedData <- group_by(filteredData, Month)
summarise(groupedData, mean(Temp, na.rm = TRUE))
```

```
## # A tibble: 4 x 2
##   Month `mean(Temp, na.rm = TRUE)`
##   <int>                      <dbl>
## 1     6                       79.1
## 2     7                       83.9
## 3     8                       84.0
## 4     9                       76.9
```

With piping, the above code can be rewritten as:

```
airquality %>%
    filter(Month != 5) %>%
    group_by(Month) %>%
    summarise(mean(Temp, na.rm = TRUE))
```

```
## # A tibble: 4 x 2
##   Month `mean(Temp, na.rm = TRUE)`
##   <int>                      <dbl>
## 1     6                       79.1
## 2     7                       83.9
## 3     8                       84.0
## 4     9                       76.9
```

This is a very basic example, and the usefulness may not be very apparent, but as the number of operations/functions perfomed on the data increase, the pipe operator becomes more and more useful!

8.9 Explaining `dplyr` – A More Complicated Example

8.9.1 Data: `nycflights13`

To explore the basic data manipulation verbs of `dplyr`, we'll start with the built in `nycflights13` data frame. This dataset contains all 336,776 flights that departed from New York City in 2013. The data comes from the US Bureau of Transportation Statistics, and is documented in `?nycflights13`.

```
packages(nycflights13)
```

```
## Loading required package: nycflights13
```

```
## Warning: package 'nycflights13' was built under R version 4.1.2
```

```
dim(flights)
```

```
## [1] 336776     19
```

```
head(flights)
```

```
## # A tibble: 6 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
## 1  2013     1     1      517            515         2      830            819
## 2  2013     1     1      533            529         4      850            830
## 3  2013     1     1      542            540         2      923            850
## 4  2013     1     1      544            545        -1     1004           1022
## 5  2013     1     1      554            600        -6      812            837
## 6  2013     1     1      554            558        -4      740            728
## # ... with 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

8.10 Single table verbs

Package `dplyr` aims to provide a function for each basic verb of data manipulation:

`filter()` (and `slice()`)

`arrange()`

`select()` (and `rename()`)

`distinct()`

`mutate()` (and `transmute()`)

`summarise()`

`sample_n()` (and `sample_frac()`)

8.11 Filter rows with `filter()`

`filter()` allows you to select a subset of rows in a data frame. The first argument is the name of the data frame. The second and subsequent arguments are the expressions that filter the data frame:

For example, we can select all flights on January 1st with:

```
filter(flights, month == 1, day == 1)
```

```
## # A tibble: 842 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
```

```
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 832 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

This is equivalent to the more verbose code in base R:

```
flights[flights$month == 1 & flights$day == 1, ]
```

```
## # A tibble: 842 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 832 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

`filter()` works similarly to `subset()` except that you can give it any number of filtering conditions, which are joined together with & (not && which is easy to do accidentally!). You can also use other boolean operators:

```
filter(flights, month == 1 | month == 2)
```

```
## # A tibble: 51,955 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 51,945 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

To select rows by position, use `slice()`:

```
slice(flights, 1:10)
```

```
## # A tibble: 10 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

8.11.1 Arrange rows with `arrange()`

`arrange()` works similarly to `filter()` except that instead of filtering or selecting rows, it reorders them. It takes a data frame, and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

```
arrange(flights, year, month, day)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Use `desc()` to order a column in descending order:

```
arrange(flights, desc(arr_delay))
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     9      641            900      1301     1242           1530
##  2  2013     6    15     1432           1935      1137     1607           2120
##  3  2013     1    10     1121           1635      1126     1239           1810
##  4  2013     9    20     1139           1845      1014     1457           2210
##  5  2013     7    22      845           1600      1005     1044           1815
##  6  2013     4    10     1100           1900       960     1342           2211
##  7  2013     3    17     2321            810       911      135           1020
##  8  2013     7    22     2257            759       898      121           1026
##  9  2013    12     5      756           1700       896     1058           2020
## 10  2013     5     3     1133           2055       878     1250           2215
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

`dplyr::arrange()` works the same way as `plyr::arrange()`. It's a straightforward wrapper around `order()` that requires less typing. The previous code is equivalent to:

```
flights[order(flights$year, flights$month, flights$day), ]
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

```
flights[order(flights$arr_delay, decreasing = TRUE), ]
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     9      641            900      1301     1242           1530
##  2  2013     6    15     1432           1935      1137     1607           2120
##  3  2013     1    10     1121           1635      1126     1239           1810
##  4  2013     9    20     1139           1845      1014     1457           2210
##  5  2013     7    22      845           1600      1005     1044           1815
##  6  2013     4    10     1100           1900       960     1342           2211
##  7  2013     3    17     2321            810       911      135           1020
##  8  2013     7    22     2257            759       898      121           1026
```

```
##  9  2013    12     5      756           1700       896     1058           2020
## 10  2013     5     3     1133           2055       878     1250           2215
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

or

```
flights[order(-flights$arr_delay), ]
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     9      641            900      1301     1242           1530
##  2  2013     6    15     1432           1935      1137     1607           2120
##  3  2013     1    10     1121           1635      1126     1239           1810
##  4  2013     9    20     1139           1845      1014     1457           2210
##  5  2013     7    22      845           1600      1005     1044           1815
##  6  2013     4    10     1100           1900       960     1342           2211
##  7  2013     3    17     2321            810       911      135           1020
##  8  2013     7    22     2257            759       898      121           1026
##  9  2013    12     5      756           1700       896     1058           2020
## 10  2013     5     3     1133           2055       878     1250           2215
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

8.11.2 Select columns with `select()`

Often you work with large datasets with many columns but only a few are actually of interest to you. `select()` allows you to rapidly zoom in on a useful subset using operations that usually only work on numeric variable positions:

```
# Select columns by name
dplyr::select(flights, year, month, day)
```

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
```

```
# Select all columns between year and day (inclusive)
dplyr::select(flights, year:day)
```

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
```

```
# Select all columns except those from year to day (inclusive)
dplyr::select(flights, -(year:day))
```

```
## # A tibble: 336,776 x 16
##    dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay carrier
##       <int>          <int>     <dbl>    <int>          <int>     <dbl> <chr>
##  1      517            515         2      830            819        11 UA
##  2      533            529         4      850            830        20 UA
##  3      542            540         2      923            850        33 AA
##  4      544            545        -1     1004           1022       -18 B6
##  5      554            600        -6      812            837       -25 DL
##  6      554            558        -4      740            728        12 UA
##  7      555            600        -5      913            854        19 B6
##  8      557            600        -3      709            723       -14 EV
##  9      557            600        -3      838            846        -8 B6
## 10      558            600        -2      753            745         8 AA
## # ... with 336,766 more rows, and 9 more variables: flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

This function works similarly to the select argument in `base::subset()`. Because the `dplyr` philosophy is to have small functions that do one thing well, it's its own function in `dplyr`.

There are a number of helper functions you can use within `select()`, like `starts_with()`, `ends_with()`, `matches()`, and `contains()`. These let you quickly match larger blocks of variables that meet some criterion. See `?select` for more details.

You can rename variables with `select()` by using named arguments:

```
#select(flights, tail_num = tailnum)
```

But because `select()` drops all the variables not explicitly mentioned, it's not that useful. Instead, use `rename()`:

```
#rename(flights, tail_num = tailnum)
```

8.11.3 Extract distinct (unique) rows

Use `distinct()` to find unique values in a table:

```
distinct(flights, tailnum)
```

```
## # A tibble: 4,044 x 1
##    tailnum
##    <chr>
##  1 N14228
##  2 N24211
##  3 N619AA
##  4 N804JB
##  5 N668DN
##  6 N39463
##  7 N516JB
##  8 N829AS
##  9 N593JB
## 10 N3ALAA
## # ... with 4,034 more rows
```

```
distinct(flights, origin, dest)
```

```
## # A tibble: 224 x 2
##    origin dest
##    <chr>  <chr>
##  1 EWR    IAH
##  2 LGA    IAH
##  3 JFK    MIA
##  4 JFK    BQN
##  5 LGA    ATL
##  6 EWR    ORD
##  7 EWR    FLL
##  8 LGA    IAD
##  9 JFK    MCO
## 10 LGA    ORD
## # ... with 214 more rows
```

(This is very similar to `base::unique()` but should be much faster.)

8.11.4 Add new columns with mutate()

Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns. This is the job of `mutate()`:

```
mutate(flights,
  gain = arr_delay - dep_delay,
  speed = distance / air_time * 60)
```

```
## # A tibble: 336,776 x 21
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 336,766 more rows, and 13 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
## #   gain <dbl>, speed <dbl>
```

`dplyr::mutate()` works to `base::transform()`. The key difference between `mutate()` and `transform()` is that mutate allows you to refer to columns that you've just created:

```
mutate(flights,
  gain = arr_delay - dep_delay,
  gain_per_hour = gain / (air_time / 60)
)
```

```
## # A tibble: 336,776 x 21
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 336,766 more rows, and 13 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
## #   gain <dbl>, gain_per_hour <dbl>
```

But:

```
transform(flights,
  gain = arr_delay - delay,
  gain_per_hour = gain / (air_time / 60)
)
#> Error: object 'gain' not found
```

If you only want to keep the new variables, use transmute():

```
transmute(flights,
  gain = arr_delay - dep_delay,
  gain_per_hour = gain / (air_time / 60)
)
```

```
## # A tibble: 336,776 x 2
##     gain gain_per_hour
##    <dbl>         <dbl>
##  1     9          2.38
##  2    16          4.23
##  3    31         11.6
##  4   -17         -5.57
##  5   -19         -9.83
##  6    16          6.4
##  7    24          9.11
##  8   -11        -12.5
##  9    -5         -2.14
## 10    10          4.35
## # ... with 336,766 more rows
```

8.11.5 Summarise values with `summarise()`

The last verb is `summarise()`. It collapses a data frame to a single row:

```
summarise(flights,
  delay = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   delay
##   <dbl>
## 1  12.6
```

Below, we'll see how this verb can be very useful.

8.11.6 Randomly sample rows with sample_n() and sample_frac()

You can use `sample_n()` and `sample_frac()` to take a random sample of rows: use `sample_n()` for a fixed number and `sample_frac()` for a fixed fraction.

```
sample_n(flights, 10)
```

```
## # A tibble: 10 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     2     9       NA           1225        NA       NA           1355
##  2  2013     7     9      656            700        -4      916            940
##  3  2013     6     7      636            634         2      838            845
##  4  2013     7    23     1826           1829        -3     2101           2035
##  5  2013     5    30     1457           1455         2     1735           1752
##  6  2013     1     5     1746           1750        -4     2037           2045
```

```
##  7  2013     3     8     2031           1929        62     2221           2137
##  8  2013     7    20     1550           1552        -2     1805           1840
##  9  2013     5    23       NA           1603        NA       NA           1730
## 10  2013     6    12     2028           2029        -1       14             25
## # ... with 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

```
sample_frac(flights, 0.01)
```

```
## # A tibble: 3,368 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     5    14     1345           1355       -10     1539           1554
##  2  2013     9    28     1659           1700        -1     1917           1955
##  3  2013     7     1     2049           2034        15       37             30
##  4  2013     3    23     1905           1905         0     2206           2139
##  5  2013     3    15      604            600         4      813            815
##  6  2013    12     5     1435           1345        50     1753           1705
##  7  2013    11    10     2047           2025        22        7           2340
##  8  2013     1    26     1939           1940        -1     2138           2125
##  9  2013     2     8      639            635         4      937            940
## 10  2013     9    14     1000           1004        -4     1114           1120
## # ... with 3,358 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Use `replace = TRUE` to perform a bootstrap sample. If needed, you can weight the sample with the weight argument.

8.11.7 Commonalities

You may have noticed that the syntax and function of all these verbs are very similar:

- The first argument is a data frame.
- The subsequent arguments describe what to do with the data frame. Notice that you can refer to columns in the data frame directly without using $.
- The result is a new data frame
- Together these properties make it easy to chain together multiple simple steps to achieve a complex result.

These five functions provide the basis of a language of data manipulation. At the most basic level, you can only alter a tidy data frame in five useful ways: you can reorder the rows (`arrange()`), pick observations and variables of interest (`filter()` and `select()`), add new variables that are functions of existing variables (`mutate()`), or collapse many values to a summary (`summarise()`). The remainder of the language comes from applying the five functions to different types of data. For example, I'll discuss how these functions work with grouped data.

8.12 Grouped operations

These verbs are useful on their own, but they become really powerful when you apply them to groups of observations within a dataset. In `dplyr`, you do this by with the `group_by()` function. It breaks down a dataset into specified groups of rows. When you then apply the verbs above on the resulting object they'll be automatically applied "by group". Most importantly, all this is achieved by using the same exact syntax you'd use with an ungrouped object.

Grouping affects the verbs as follows:

grouped `select()` is the same as ungrouped `select()`, except that grouping variables are always retained.

grouped `arrange()` orders first by the grouping variables

`mutate()` and `filter()` are most useful in conjunction with window functions (like `rank()`, or `min(x) == x`). They are described in detail in `vignette("window-functions")`.

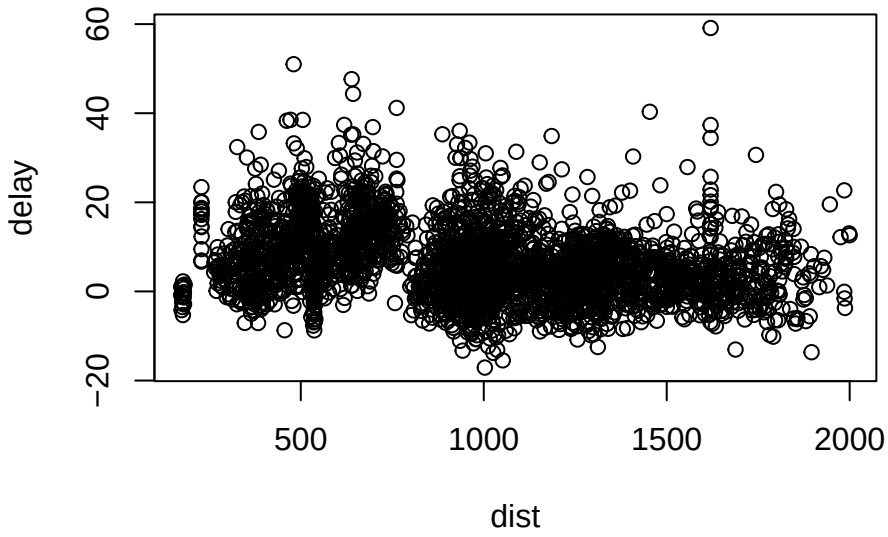
`sample_n()` and `sample_frac()` sample the specified number/fraction of rows in each group.

`slice()` extracts rows within each group.

`summarise()` is powerful and easy to understand, as described in more detail below.

In the following example, we split the complete dataset into individual planes and then summarise each plane by counting the number of flights (`count = n()`) and computing the average distance (`dist = mean(Distance, na.rm = TRUE)`) and arrival delay (`delay = mean(ArrDelay, na.rm = TRUE)`).

```
by_tailnum <- group_by(flights, tailnum)
delay <- summarise(by_tailnum,
  count = n(),
  dist = mean(distance, na.rm = TRUE),
  delay = mean(arr_delay, na.rm = TRUE))
delay <- filter(delay, count > 20, dist < 2000)
# Interestingly, the average delay is only slightly related to the
# average distance flown by a plane.
plot( delay~dist, data=delay)
```



You use `summarise()` with aggregate functions, which take a vector of values and return a single number. There are many useful examples of such functions in base R like `min()`, `max()`, `mean()`, `sum()`, `sd()`, `median()`, and `IQR()`. `dplyr` provides a handful of others:

- `n()`: the number of observations in the current group
- `n_distinct(x)`: the number of unique values in x.
- `first(x)`, `last(x)` and `nth(x, n)` - these work similarly to `x[1]`, `x[length(x)]`, and `x[n]` but give you more control over the result if the value is missing.

For example, we could use these to find the number of planes and the number of flights that go to each possible destination:

```
destinations <- group_by(flights, dest)
summarise(destinations,
  planes = n_distinct(tailnum),
  flights = n()
)
```

```
## # A tibble: 105 x 3
##    dest  planes flights
##    <chr>  <int>   <int>
##  1 ABQ      108     254
##  2 ACK       58     265
##  3 ALB      172     439
##  4 ANC        6       8
##  5 ATL     1180   17215
##  6 AUS      993    2439
##  7 AVL      159     275
##  8 BDL      186     443
```

```
##  9 BGR       46     375
## 10 BHM       45     297
## # ... with 95 more rows
```

You can also use any function that you write yourself. For performance, `dplyr` provides optimised `C++` versions of many of these functions. If you want to provide your own `C++` function, see the hybrid-evaluation vignette for more details.

When you group by multiple variables, each summary peels off one level of the grouping. That makes it easy to progressively roll-up a dataset:

```
daily <- group_by(flights, year, month, day)
(per_day   <- summarise(daily, flights = n()))
```

```
## `summarise()` has grouped output by 'year', 'month'. You can override using the `.groups` argument.
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day flights
##    <int> <int> <int>   <int>
##  1  2013     1     1     842
##  2  2013     1     2     943
##  3  2013     1     3     914
##  4  2013     1     4     915
##  5  2013     1     5     720
##  6  2013     1     6     832
##  7  2013     1     7     933
##  8  2013     1     8     899
##  9  2013     1     9     902
## 10  2013     1    10     932
## # ... with 355 more rows
```

```
(per_month <- summarise(per_day, flights = sum(flights)))
```

```
## `summarise()` has grouped output by 'year'. You can override using the `.groups` argument.
```

```
## # A tibble: 12 x 3
## # Groups:   year [1]
##     year month flights
##    <int> <int>   <int>
##  1  2013     1   27004
##  2  2013     2   24951
##  3  2013     3   28834
##  4  2013     4   28330
##  5  2013     5   28796
##  6  2013     6   28243
##  7  2013     7   29425
##  8  2013     8   29327
##  9  2013     9   27574
## 10  2013    10   28889
## 11  2013    11   27268
## 12  2013    12   28135
```

```
(per_year  <- summarise(per_month, flights = sum(flights)))
```

```
## # A tibble: 1 x 2
##    year flights
##   <int>   <int>
## 1  2013  336776
```

However you need to be careful when progressively rolling up summaries like this: it's ok for sums and counts, but you need to think about weighting for means and variances (it's not possible to do this exactly for medians).

8.13 Chaining

The `dplyr` API is functional in the sense that function calls don't have side-effects. You must always save their results. This doesn't lead to particularly elegant code, especially if you want to do many operations at once. You either have to do it step-by-step:

```
a1 <- group_by(flights, year, month, day)
a2 <- dplyr::select(a1, arr_delay, dep_delay)
```

```
## Adding missing grouping variables: `year`, `month`, `day`
```

```
a3 <- summarise(a2,
  arr = mean(arr_delay, na.rm = TRUE),
  dep = mean(dep_delay, na.rm = TRUE))
```

```
## `summarise()` has grouped output by 'year', 'month'. You can override using the `.groups` argument.
```

```
a4 <- filter(a3, arr > 30 | dep > 30)
```

Or if you don't want to save the intermediate results, you need to wrap the function calls inside each other:

```
filter(
  summarise(
    dplyr::select(
      group_by(flights, year, month, day),
      arr_delay, dep_delay
    ),
    arr = mean(arr_delay, na.rm = TRUE),
    dep = mean(dep_delay, na.rm = TRUE)
  ),
  arr > 30 | dep > 30
)
```

```
## Adding missing grouping variables: `year`, `month`, `day`
```

```
## `summarise()` has grouped output by 'year', 'month'. You can override using the `.groups` argument.
```

```
## # A tibble: 49 x 5
## # Groups:   year, month [11]
##     year month   day   arr   dep
##    <int> <int> <int> <dbl> <dbl>
##  1  2013     1    16  34.2  24.6
##  2  2013     1    31  32.6  28.7
##  3  2013     2    11  36.3  39.1
##  4  2013     2    27  31.3  37.8
##  5  2013     3     8  85.9  83.5
##  6  2013     3    18  41.3  30.1
##  7  2013     4    10  38.4  33.0
##  8  2013     4    12  36.0  34.8
##  9  2013     4    18  36.0  34.9
## 10  2013     4    19  47.9  46.1
## # ... with 39 more rows
```

This is difficult to read because the order of the operations is from inside to out. Thus, the arguments are a long way away from the function. To get around this problem, `dplyr` provides the `%>%` operator. `x %>% f(y)` turns into `f(x, y)` so you can use it to rewrite multiple operations that you can read left-to-right, top-to-bottom:

```
flights %>%
  group_by(year, month, day) %>%
  dplyr::select(arr_delay, dep_delay) %>%
  summarise(
    arr = mean(arr_delay, na.rm = TRUE),
    dep = mean(dep_delay, na.rm = TRUE)
  ) %>%
  filter(arr > 30 | dep > 30)
```

```
## Adding missing grouping variables: `year`, `month`, `day`
```

```
## `summarise()` has grouped output by 'year', 'month'. You can override using the `.groups` argument.
```

```
## # A tibble: 49 x 5
## # Groups:   year, month [11]
##     year month   day   arr   dep
##    <int> <int> <int> <dbl> <dbl>
##  1  2013     1    16  34.2  24.6
##  2  2013     1    31  32.6  28.7
##  3  2013     2    11  36.3  39.1
##  4  2013     2    27  31.3  37.8
##  5  2013     3     8  85.9  83.5
##  6  2013     3    18  41.3  30.1
##  7  2013     4    10  38.4  33.0
##  8  2013     4    12  36.0  34.8
##  9  2013     4    18  36.0  34.9
## 10  2013     4    19  47.9  46.1
## # ... with 39 more rows
```

8.14 A quick Summary

I will use the mammals sleep data to illustrate the verbs of `dplyr`

```
packages(downloader)
```

```
## Loading required package: downloader
```

```
## Warning: package 'downloader' was built under R version 4.1.2
```

```
url <- "https://raw.githubusercontent.com/genomicsclass/dagdata/master/inst/extdata/msleep_ggplot2.csv"
filename <- "msleep_ggplot2.csv"
if (!file.exists(filename)) download(url,filename)
msleep <- read.csv("msleep_ggplot2.csv")
head(msleep)
```

```
##                         name      genus  vore        order conservation
## 1                    Cheetah   Acinonyx carni    Carnivora           lc
## 2                 Owl monkey      Aotus  omni     Primates         <NA>
## 3            Mountain beaver Aplodontia herbi     Rodentia           nt
## 4 Greater short-tailed shrew    Blarina  omni Soricomorpha           lc
## 5                        Cow        Bos herbi Artiodactyla domesticated
## 6           Three-toed sloth   Bradypus herbi       Pilosa         <NA>
##   sleep_total sleep_rem sleep_cycle awake brainwt  bodywt
## 1        12.1        NA          NA  11.9      NA  50.000
## 2        17.0       1.8          NA   7.0 0.01550   0.480
## 3        14.4       2.4          NA   9.6      NA   1.350
## 4        14.9       2.3   0.1333333   9.1 0.00029   0.019
## 5         4.0       0.7   0.6666667  20.0 0.42300 600.000
## 6        14.4       2.2   0.7666667   9.6      NA   3.850
```

The columns (in order) correspond to the following:

|————|————| |column name | Description| |name | common name| |genus | taxonomic rank| |vore | carnivore, omnivore or herbivore?| |order | taxonomic rank| |conservation |the conservation status of the mammal| |sleep_total | total amount of sleep, in hours| |sleep_rem |rem sleep, in hours| |sleep_cycle | length of sleep cycle, in hours| |awake | amount of time spent awake, in hours| |brainwt | brain weight in kilograms| |bodywt | body weight in kilograms| ———|————————-|

Important dplyr verbs to remember

|————|————-| |dplyr verbs| Description | |select() | select columns| |filter() | filter rows| |arrange() | re-order or arrange rows| |mutate() | create new columns| |summarise()| summarise values| |group_by() | allows for group operations in the "split-apply-combine" concept | ————|————|

8.14.1 dplyr verbs in action

The two most basic functions are `select()` and `filter()` which selects columns and filters rows, respectively.

Selecting columns using `select()`

```
## Select a set of columns: the name and the sleep_total columns.
sleepData <- dplyr::select(msleep, name, sleep_total)
head(sleepData)
```

```
##                         name sleep_total
## 1                    Cheetah        12.1
## 2                 Owl monkey        17.0
## 3             Mountain beaver        14.4
## 4 Greater short-tailed shrew        14.9
## 5                        Cow         4.0
## 6            Three-toed sloth        14.4
```

To select all the columns except a specific column, use the "-" (subtraction) operator (also known as negative indexing)

```
head(dplyr::select(msleep, -name))
```

```
##        genus  vore        order conservation sleep_total sleep_rem sleep_cycle
## 1   Acinonyx carni    Carnivora           lc        12.1        NA          NA
## 2      Aotus  omni     Primates         <NA>        17.0       1.8          NA
## 3 Aplodontia herbi     Rodentia           nt        14.4       2.4          NA
## 4    Blarina  omni Soricomorpha           lc        14.9       2.3   0.1333333
## 5        Bos herbi Artiodactyla domesticated         4.0       0.7   0.6666667
## 6   Bradypus herbi       Pilosa         <NA>        14.4       2.2   0.7666667
##   awake brainwt  bodywt
## 1  11.9      NA  50.000
## 2   7.0 0.01550   0.480
## 3   9.6      NA   1.350
## 4   9.1 0.00029   0.019
## 5  20.0 0.42300 600.000
## 6   9.6      NA   3.850
```

To select a range of columns by name, use the ":" (colon) operator

```
head(dplyr::select(msleep, name:order))
```

```
##                         name      genus  vore        order
## 1                    Cheetah   Acinonyx carni    Carnivora
## 2                 Owl monkey      Aotus  omni     Primates
## 3             Mountain beaver Aplodontia herbi     Rodentia
## 4 Greater short-tailed shrew    Blarina  omni Soricomorpha
## 5                        Cow        Bos herbi Artiodactyla
## 6            Three-toed sloth   Bradypus herbi       Pilosa
```

To select all columns that start with the character string "sl", use the function `starts_with()`

```
head(dplyr::select(msleep, starts_with("sl")))
```

```
##   sleep_total sleep_rem sleep_cycle
## 1        12.1        NA          NA
## 2        17.0       1.8          NA
## 3        14.4       2.4          NA
## 4        14.9       2.3   0.1333333
## 5         4.0       0.7   0.6666667
## 6        14.4       2.2   0.7666667
```

Some additional options to select columns based on a specific criteria include

`ends_with()` = Select columns that end with a character string

`contains()` = Select columns that contain a character string

`matches()` = Select columns that match a regular expression

`one_of()` = Select columns names that are from a group of names

Selecting rows using `filter()`

```
## Filter the rows for mammals that sleep a total of more than 16 hours.
filter(msleep, sleep_total >= 16)
```

```
##                     name        genus    vore           order conservation
## 1             Owl monkey        Aotus    omni        Primates         <NA>
## 2   Long-nosed armadillo      Dasypus   carni       Cingulata           lc
## 3 North American Opossum    Didelphis    omni Didelphimorphia           lc
## 4          Big brown bat    Eptesicus insecti      Chiroptera           lc
## 5   Thick-tailed opposum   Lutreolina   carni Didelphimorphia           lc
## 6       Little brown bat       Myotis insecti      Chiroptera         <NA>
## 7        Giant armadillo   Priodontes insecti       Cingulata           en
## 8 Arctic ground squirrel Spermophilus   herbi        Rodentia           lc
##   sleep_total sleep_rem sleep_cycle awake brainwt bodywt
## 1        17.0       1.8          NA   7.0 0.01550  0.480
## 2        17.4       3.1   0.3833333   6.6 0.01080  3.500
## 3        18.0       4.9   0.3333333   6.0 0.00630  1.700
## 4        19.7       3.9   0.1166667   4.3 0.00030  0.023
## 5        19.4       6.6          NA   4.6      NA  0.370
## 6        19.9       2.0   0.2000000   4.1 0.00025  0.010
## 7        18.1       6.1          NA   5.9 0.08100 60.000
## 8        16.6        NA          NA   7.4 0.00570  0.920
```

Filter the rows for mammals that sleep a total of more than 16 hours and have a body weight of greater than 1 kilogram.

```
filter(msleep, sleep_total >= 16, bodywt >= 1)
```

```
##                     name      genus    vore           order conservation
## 1   Long-nosed armadillo    Dasypus   carni       Cingulata           lc
## 2 North American Opossum  Didelphis    omni Didelphimorphia           lc
## 3        Giant armadillo Priodontes insecti       Cingulata           en
##   sleep_total sleep_rem sleep_cycle awake brainwt bodywt
## 1        17.4       3.1   0.3833333   6.6  0.0108    3.5
## 2        18.0       4.9   0.3333333   6.0  0.0063    1.7
## 3        18.1       6.1          NA   5.9  0.0810   60.0
```

Filter the rows for mammals in the *Perissodactyla* and *Primates* taxonomic order

```
filter(msleep, order %in% c("Perissodactyla", "Primates"))
```

```
##              name         genus  vore          order conservation sleep_total
## 1       Owl monkey         Aotus  omni       Primates         <NA>        17.0
```

```
## 2           Grivet Cercopithecus  omni       Primates           lc        10.0
## 3            Horse         Equus herbi Perissodactyla domesticated         2.9
## 4           Donkey         Equus herbi Perissodactyla domesticated         3.1
## 5     Patas monkey  Erythrocebus  omni       Primates           lc        10.9
## 6           Galago        Galago  omni       Primates         <NA>         9.8
## 7            Human          Homo  omni       Primates         <NA>         8.0
## 8   Mongoose lemur         Lemur herbi       Primates           vu         9.5
## 9          Macaque        Macaca  omni       Primates         <NA>        10.1
## 10      Slow loris     Nyctibeus carni       Primates         <NA>        11.0
## 11      Chimpanzee           Pan  omni       Primates         <NA>         9.7
## 12          Baboon         Papio  omni       Primates         <NA>         9.4
## 13           Potto  Perodicticus  omni       Primates           lc        11.0
## 14 Squirrel monkey       Saimiri  omni       Primates         <NA>         9.6
## 15 Brazilian tapir       Tapirus herbi Perissodactyla           vu         4.4
##    sleep_rem sleep_cycle awake brainwt  bodywt
## 1        1.8          NA   7.0  0.0155   0.480
## 2        0.7          NA  14.0      NA   4.750
## 3        0.6   1.0000000  21.1  0.6550 521.000
## 4        0.4          NA  20.9  0.4190 187.000
## 5        1.1          NA  13.1  0.1150  10.000
## 6        1.1   0.5500000  14.2  0.0050   0.200
## 7        1.9   1.5000000  16.0  1.3200  62.000
## 8        0.9          NA  14.5      NA   1.670
## 9        1.2   0.7500000  13.9  0.1790   6.800
## 10        NA          NA  13.0  0.0125   1.400
## 11       1.4   1.4166667  14.3  0.4400  52.200
## 12       1.0   0.6666667  14.6  0.1800  25.235
## 13        NA          NA  13.0      NA   1.100
## 14       1.4          NA  14.4  0.0200   0.743
## 15       1.0   0.9000000  19.6  0.1690 207.501
```

You can use the boolean operators (e.g. `>, <, >=, <=, !=, %in%`) to create the logical tests.

Pipe operator: `%>%`

Before we go any futher, let's introduce the pipe operator: `%>%`. `dplyr` imports this operator from another package (`magrittr`). This operator allows you to pipe the output from one function to the input of another function. Instead of nesting functions (reading from the inside to the outside), the idea of of piping is to read the functions from left to right.

Here's an example you have seen:

```
head(dplyr::select(msleep, name, sleep_total))
```

```
##                         name sleep_total
## 1                    Cheetah        12.1
## 2                 Owl monkey        17.0
## 3             Mountain beaver        14.4
## 4 Greater short-tailed shrew        14.9
## 5                        Cow         4.0
## 6           Three-toed sloth        14.4
```

Now in this case, we will pipe the msleep data frame to the function that will select two columns (name and sleep_total) and then pipe the new data frame to the function head() which will return the head of the new data frame.

```
msleep %>%
    dplyr::select(name, sleep_total) %>%
    head
```

```
##                          name sleep_total
## 1                     Cheetah        12.1
## 2                 Owl monkey        17.0
## 3             Mountain beaver        14.4
## 4 Greater short-tailed shrew        14.9
## 5                         Cow         4.0
## 6           Three-toed sloth        14.4
```

You will soon see how useful the pipe operator is when we start to combine many functions.

Back to `dplyr` verbs in action

Now that you know about the pipe operator (`%>%`), we will use it throughout the rest of the summary.

Arrange or re-order rows using `arrange()`

```
##To arrange (or re-order) rows by a particular column such as the taxonomic order, list the name of th
 msleep %>% arrange(order) %>% head
```

```
##       name     genus  vore        order conservation sleep_total sleep_rem
## 1   Tenrec    Tenrec  omni Afrosoricida         <NA>        15.6       2.3
## 2      Cow       Bos herbi Artiodactyla domesticated         4.0       0.7
## 3 Roe deer Capreolus herbi Artiodactyla           lc         3.0        NA
## 4     Goat     Capri herbi Artiodactyla           lc         5.3       0.6
## 5  Giraffe   Giraffa herbi Artiodactyla           cd         1.9       0.4
## 6    Sheep      Ovis herbi Artiodactyla domesticated         3.8       0.6
##   sleep_cycle awake brainwt  bodywt
## 1          NA   8.4  0.0026   0.900
## 2   0.6666667  20.0  0.4230 600.000
## 3          NA  21.0  0.0982  14.800
## 4          NA  18.7  0.1150  33.500
## 5          NA  22.1      NA 899.995
## 6          NA  20.2  0.1750  55.500
```

Now, we will select three columns from `msleep`, arrange the rows by the taxonomic order and then arrange the rows by `sleep_total`. Finally show the head of the final data frame

```
msleep %>%
    dplyr::select(name, order, sleep_total) %>%
    arrange(order, sleep_total) %>%
    head
```

```
##       name        order sleep_total
## 1   Tenrec Afrosoricida        15.6
## 2  Giraffe Artiodactyla         1.9
## 3 Roe deer Artiodactyla         3.0
## 4    Sheep Artiodactyla         3.8
## 5      Cow Artiodactyla         4.0
## 6     Goat Artiodactyla         5.3
```

Same as above, except here we filter the rows for mammals that sleep for 16 or more hours instead of showing the head of the final data frame

```
msleep %>%
    dplyr::select(name, order, sleep_total) %>%
    arrange(order, sleep_total) %>%
    filter(sleep_total >= 16)
```

```
##                     name           order sleep_total
## 1          Big brown bat      Chiroptera        19.7
## 2       Little brown bat      Chiroptera        19.9
## 3   Long-nosed armadillo       Cingulata        17.4
## 4        Giant armadillo       Cingulata        18.1
## 5 North American Opossum Didelphimorphia        18.0
## 6   Thick-tailed opposum Didelphimorphia        19.4
## 7             Owl monkey        Primates        17.0
## 8 Arctic ground squirrel        Rodentia        16.6
```

Something slightly more complicated: same as above, except arrange the rows in the `sleep_total` column in a descending order. For this, use the function `desc()`

```
msleep %>%
    dplyr::select(name, order, sleep_total) %>%
    arrange(order, desc(sleep_total)) %>%
    filter(sleep_total >= 16)
```

```
##                     name           order sleep_total
## 1       Little brown bat      Chiroptera        19.9
## 2          Big brown bat      Chiroptera        19.7
## 3        Giant armadillo       Cingulata        18.1
## 4   Long-nosed armadillo       Cingulata        17.4
## 5   Thick-tailed opposum Didelphimorphia        19.4
## 6 North American Opossum Didelphimorphia        18.0
## 7             Owl monkey        Primates        17.0
## 8 Arctic ground squirrel        Rodentia        16.6
```

Create new columns using `mutate()`

The `mutate()` function will add new columns to the data frame. Create a new column called `rem_proportion` which is the ratio of rem sleep to total amount of sleep.

```
msleep %>%
    mutate(rem_proportion = sleep_rem / sleep_total) %>%
    head
```

```
##                        name      genus  vore        order conservation
## 1                   Cheetah   Acinonyx carni    Carnivora           lc
## 2                 Owl monkey      Aotus  omni     Primates         <NA>
## 3            Mountain beaver Aplodontia herbi     Rodentia           nt
## 4 Greater short-tailed shrew    Blarina  omni Soricomorpha           lc
## 5                       Cow        Bos herbi Artiodactyla domesticated
## 6           Three-toed sloth   Bradypus herbi       Pilosa         <NA>
```

```
##   sleep_total sleep_rem sleep_cycle awake brainwt  bodywt rem_proportion
## 1        12.1        NA          NA  11.9      NA  50.000             NA
## 2        17.0       1.8          NA   7.0 0.01550   0.480      0.1058824
## 3        14.4       2.4          NA   9.6      NA   1.350      0.1666667
## 4        14.9       2.3   0.1333333   9.1 0.00029   0.019      0.1543624
## 5         4.0       0.7   0.6666667  20.0 0.42300 600.000      0.1750000
## 6        14.4       2.2   0.7666667   9.6      NA   3.850      0.1527778
```

You can many new columns using `mutate` (separated by commas). Here we add a second column called `bodywt_grams` which is the `bodywt` column in grams.

```
msleep %>%
    mutate(rem_proportion = sleep_rem / sleep_total,
           bodywt_grams = bodywt * 1000) %>%
    head
```

```
##                         name      genus  vore        order conservation
## 1                    Cheetah   Acinonyx carni    Carnivora           lc
## 2                 Owl monkey      Aotus  omni     Primates         <NA>
## 3            Mountain beaver Aplodontia herbi     Rodentia           nt
## 4 Greater short-tailed shrew    Blarina  omni Soricomorpha           lc
## 5                        Cow        Bos herbi Artiodactyla domesticated
## 6           Three-toed sloth   Bradypus herbi       Pilosa         <NA>
##   sleep_total sleep_rem sleep_cycle awake brainwt  bodywt rem_proportion
## 1        12.1        NA          NA  11.9      NA  50.000             NA
## 2        17.0       1.8          NA   7.0 0.01550   0.480      0.1058824
## 3        14.4       2.4          NA   9.6      NA   1.350      0.1666667
## 4        14.9       2.3   0.1333333   9.1 0.00029   0.019      0.1543624
## 5         4.0       0.7   0.6666667  20.0 0.42300 600.000      0.1750000
## 6        14.4       2.2   0.7666667   9.6      NA   3.850      0.1527778
##   bodywt_grams
## 1        50000
## 2          480
## 3         1350
## 4           19
## 5       600000
## 6         3850
```

Create summaries of the data frame using `summarise()`

The `summarise()` function will create summary statistics for a given column in the data frame such as finding the mean. For example, to compute the average number of hours of sleep, apply the mean() function to the column sleep_total and call the summary value avg_sleep.

```
msleep %>%
    summarise(avg_sleep = mean(sleep_total))
```

```
##   avg_sleep
## 1  10.43373
```

There are many other summary statistics you could consider such `sd()`, `min()`, `max()`, `median()`, `sum()`, `n()` (returns the length of vector), `first()` (returns first value in vector), `last()` (returns last value in vector) and `n_distinct()` (number of distinct values in vector).

```
msleep %>%
    summarise(avg_sleep = mean(sleep_total),
              min_sleep = min(sleep_total),
              max_sleep = max(sleep_total),
              total = n())
```

```
##   avg_sleep min_sleep max_sleep total
## 1  10.43373       1.9      19.9    83
```

Group operations using `group_by()`

The `group_by()` verb is an important function in `dplyr`. As we mentioned before it's related to concept of "split-apply-combine". We literally want to split the data frame by some variable (e.g. taxonomic order), apply a function to the individual data frames and then combine the output.

Let's do that: split the `msleep` data frame by the taxonomic order, then ask for the same summary statistics as above. We expect a set of summary statistics for each taxonomic order.

```
msleep %>%
    group_by(order) %>%
    summarise(avg_sleep = mean(sleep_total),
              min_sleep = min(sleep_total),
              max_sleep = max(sleep_total),
              total = n())
```

```
## # A tibble: 19 x 5
##    order           avg_sleep min_sleep max_sleep total
##    <chr>               <dbl>     <dbl>     <dbl> <int>
##  1 Afrosoricida        15.6       15.6      15.6     1
##  2 Artiodactyla         4.52       1.9       9.1     6
##  3 Carnivora           10.1        3.5      15.8    12
##  4 Cetacea              4.5        2.7       5.6     3
##  5 Chiroptera          19.8       19.7      19.9     2
##  6 Cingulata           17.8       17.4      18.1     2
##  7 Didelphimorphia     18.7       18        19.4     2
##  8 Diprotodontia       12.4       11.1      13.7     2
##  9 Erinaceomorpha      10.2       10.1      10.3     2
## 10 Hyracoidea           5.67       5.3       6.3     3
## 11 Lagomorpha           8.4        8.4       8.4     1
## 12 Monotremata          8.6        8.6       8.6     1
## 13 Perissodactyla       3.47       2.9       4.4     3
## 14 Pilosa              14.4       14.4      14.4     1
## 15 Primates            10.5        8         17      12
## 16 Proboscidea          3.6        3.3       3.9     2
## 17 Rodentia            12.5        7         16.6    22
## 18 Scandentia           8.9        8.9       8.9     1
## 19 Soricomorpha        11.1        8.4      14.9     5
```

9 Week 11

A list of commonly used latice functions: `-xyplot` – scatter plots `-dotplot` – Cleveland dot plots `-histogram` – histograms `-barchart` – bar and column charts `-splom` – scatter plot matrix `-levelplot` – image plots `-contourplot -- contours -qq-- quantile-quantile plots -qqmath-- quantile-quantile plots,`

`comparing data distribution to a standard probability dsitribution -cloud‘` – perspective (3-D) plots

9.1 Costom Panel Functions

The panel function allows costomizing a plot, such as superimposing points, lines, text, and other objects. For example, to add a diagonal line to the two panel xyplot, we use:

```
xyplot(y~x|z, data=d,
       panel=function(x, y, ...){
         panel.abline(0, 1, col="red")
         panel.xyplot(x, y, ...)
       })
```

9.2 The Example That Started Trellis

The `barley` dataset – > These data are yields in bushels per acre, of 10 varieties of barley grown in 1/40 acre plots at University Farm, St. Paul, and at the five branch experiment stations located at Waseca, Morris, Crookston, Grand Rapids, and Duluth (all in Minnesota). The varieties were grown in three randomized blocks at each of the six stations during 1931 and 1932, different land being used each year of the test.

> Immer et al. (1934) present the data for each Year*Site*Variety*Block. The data here is the average yield across the three blocks.
>
> Immer et al. (1934) refer (once) to the experiment as being conducted in 1930 and 1931, then later refer to it (repeatedly) as being conducted in 1931 and 1932. Later authors have continued the confusion.

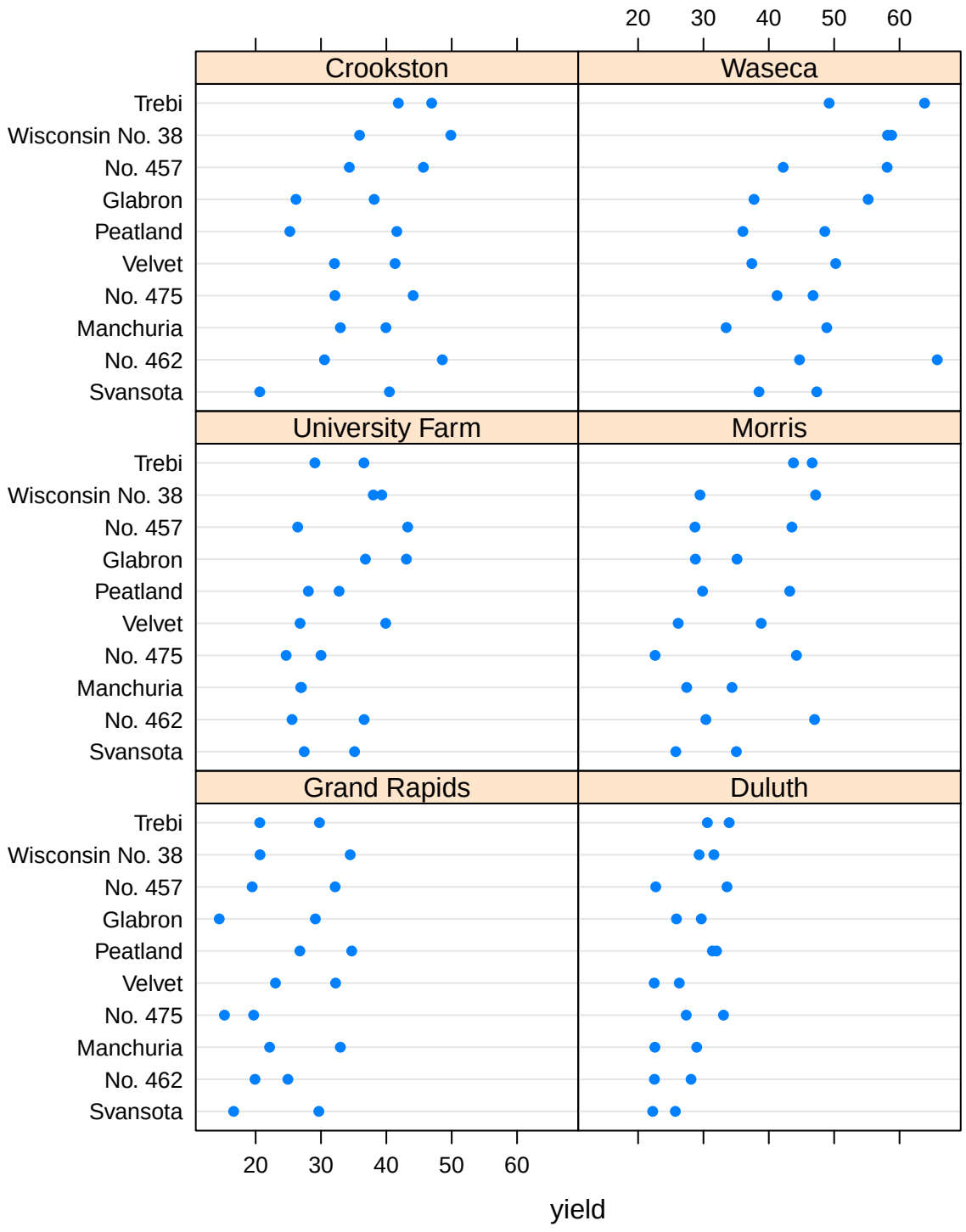
```
dim(barley)
```

```
## [1] 120   4
```

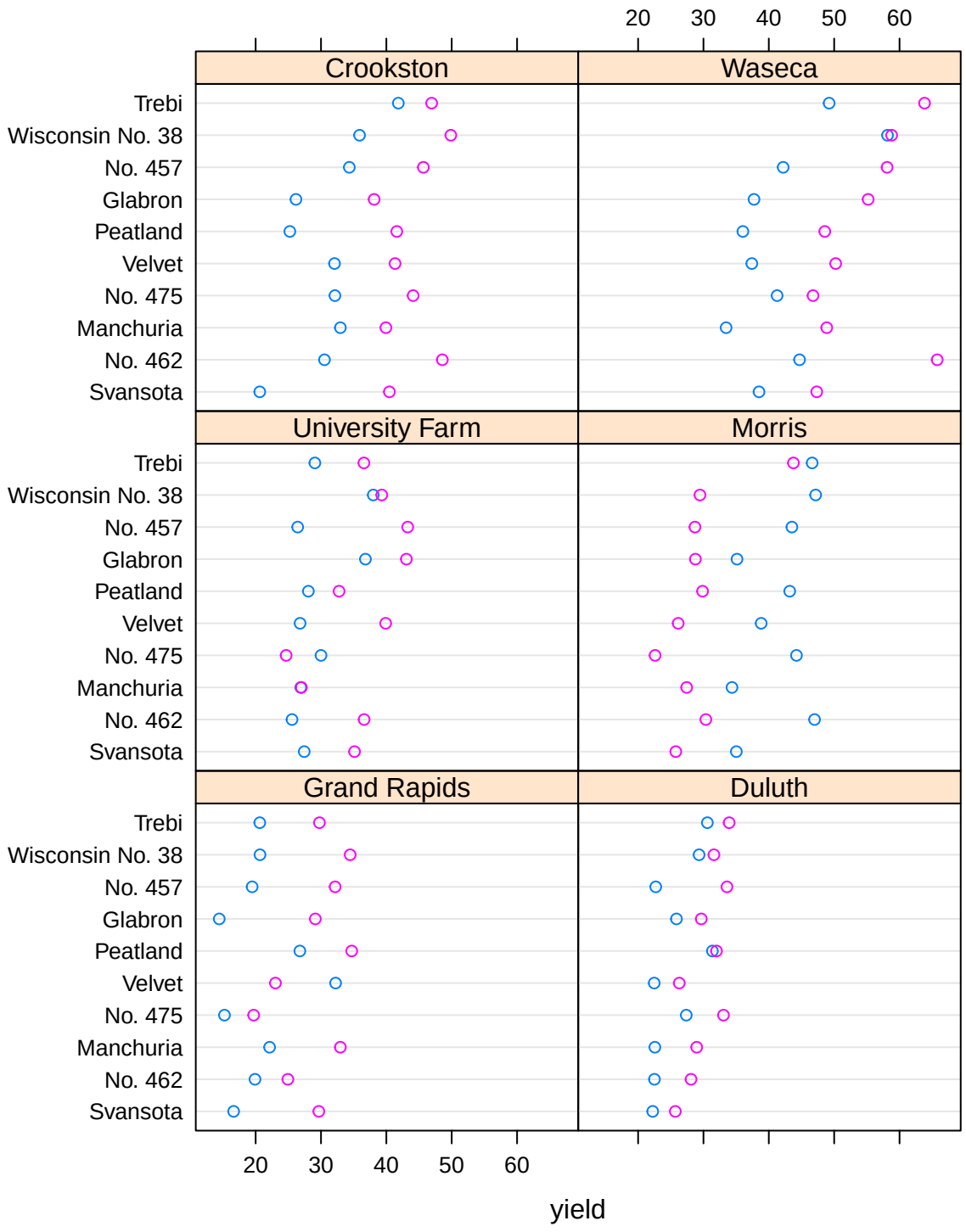
```
head(barley)
```

```
##      yield   variety year           site
## 1 27.00000 Manchuria 1931 University Farm
## 2 48.86667 Manchuria 1931          Waseca
## 3 27.43334 Manchuria 1931          Morris
## 4 39.93333 Manchuria 1931       Crookston
## 5 32.96667 Manchuria 1931    Grand Rapids
## 6 28.96667 Manchuria 1931          Duluth
```

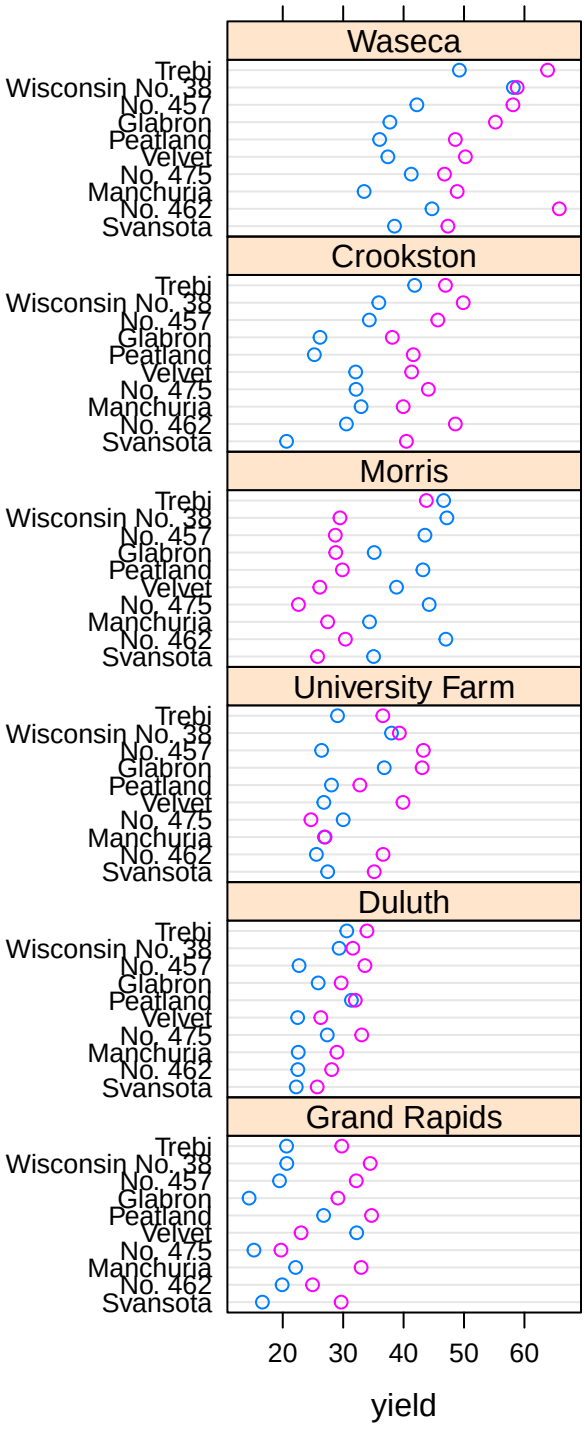
```
    dotplot(variety ~ yield | site, data = barley)
```




```
dotplot(variety ~ yield | site, data = barley, groups=year)
```

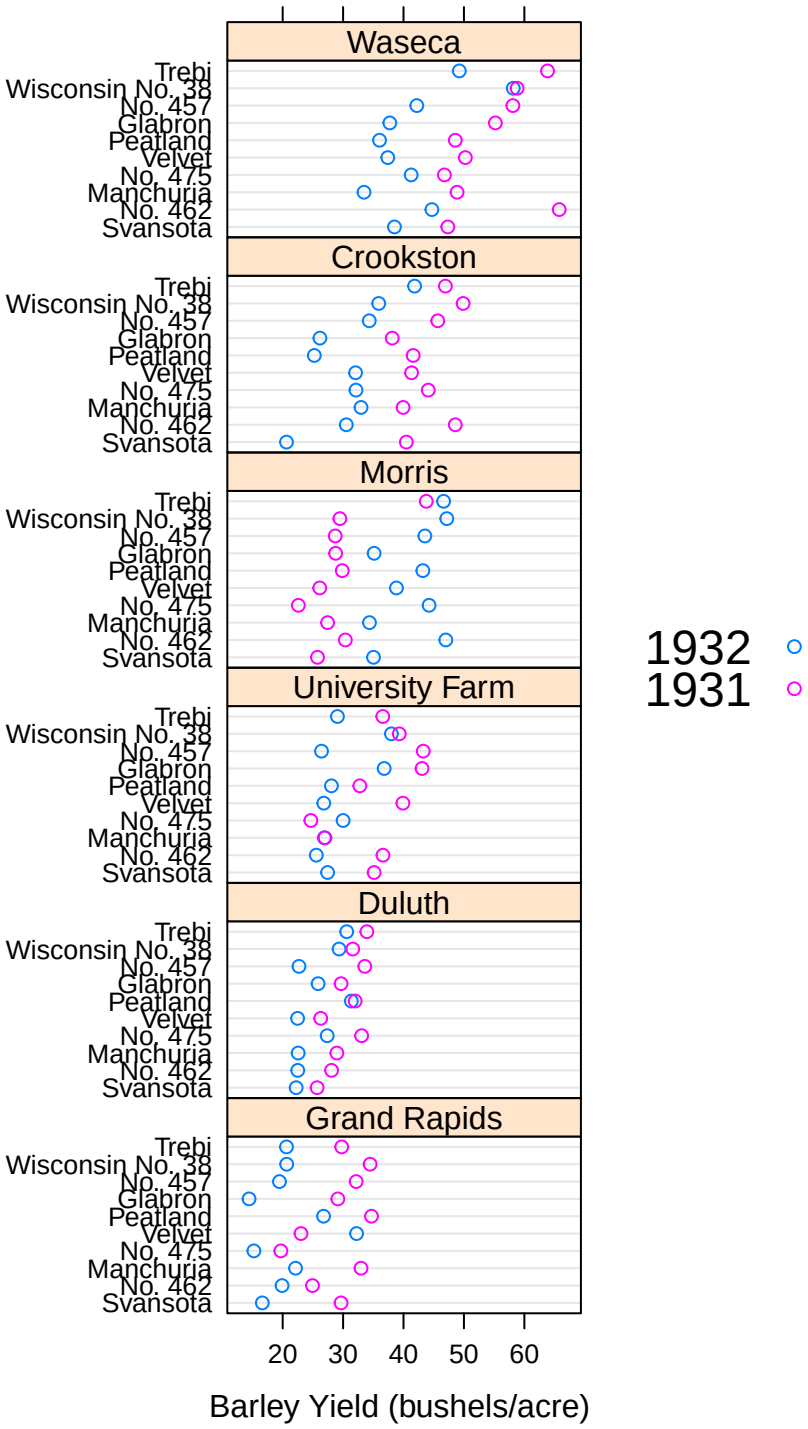



```
dotplot(variety ~ yield | site, data = barley, groups=year,
        aspect=0.5, layout=c(1,6))
```




```
key <- simpleKey(levels(barley$year), space = "right")
key$text$cex <- 1.5
dotplot(variety ~ yield | site, data = barley, groups = year,
        key = key,
        xlab = "Barley Yield (bushels/acre) ",
```

```
        aspect=0.5, layout = c(1,6), ylab=NULL)
```



Now we want to save it into a PDF file:

```
pdf(file=paste(plotDIR, "barley.pdf", sep="/"), height=8, width=5)
trellis.par.set(theme = canonical.theme("postscript", col=FALSE))
trellis.par.set(list(fontsize=list(text=6),
                par.xlab.text=list(cex=1.5),
                    add.text=list(cex=1.5),
                    superpose.symbol=list(cex=.5)))
     key <- simpleKey(levels(barley$year), space = "right")
     key$text$cex <- 1.5
     dotplot(variety ~ yield | site, data = barley, groups = year,
             key = key,
             xlab = "Barley Yield (bushels/acre) ",
             aspect=0.5, layout = c(1,6), ylab=NULL)
dev.off()
```

```
## pdf
##   2
```

9.3 Types of Data

Data used here are from Cleveland's *Visualizing Data*.

```
## R data and code for each plot in the book
packages(foreign)
#data.restore ("visualizing.data")
```

9.3.1 Univariate data

- Show and compare distributions
  - Dot-plot

```
book.1.1 <-
    function()
{
    n <- length(levels(barley$year))
    obj<-dotplot(variety ~ yield | site,
            data = barley,
            groups = year,
            layout = c(1, 6),
            aspect = .5,
            sub = list("Figure 1.1",cex=.8),
            xlab = "Barley Yield (bushels/acre)",
            key = list(points =
                        Rows(trellis.par.get("superpose.symbol"), 1:n),
                       text = list(levels(barley$year)),
                       columns = n))
    print(obj)
}
book.1.1()
```

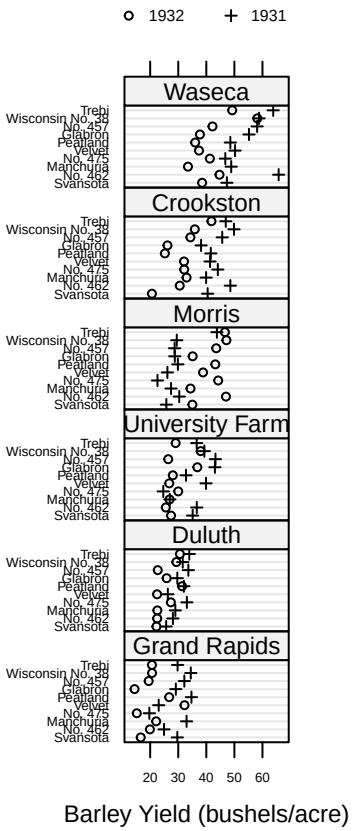
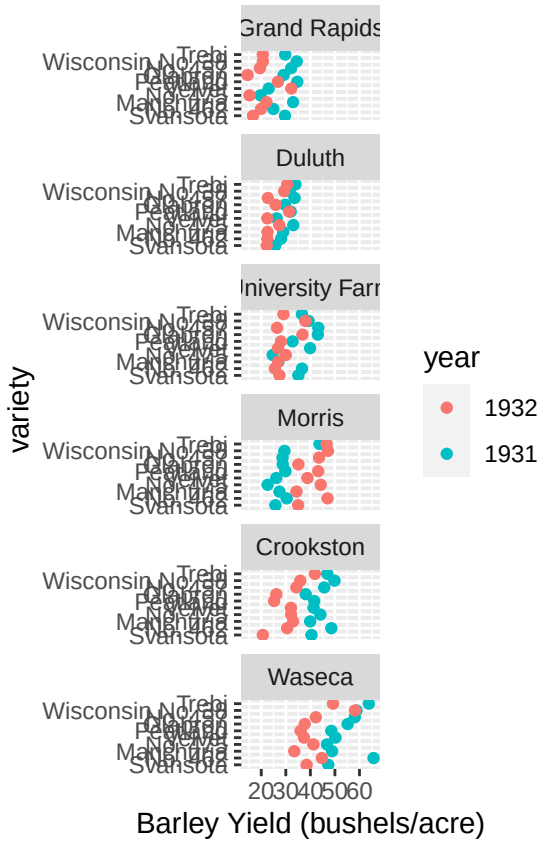


Figure 1.1

```
ggVD_1.1 <- function(asp_rt=0.5){
    p<-ggplot(barley, aes(x=yield, y=variety, color=year))+
        xlab( "Barley Yield (bushels/acre)") +
        geom_point() + facet_wrap(site~., ncol=1)
print(p + theme(aspect.ratio = asp_rt))
}

ggVD_1.1()
```



- Histogram Figure 1.2

```
book.1.2 <-
    function()
    histogram(~ height | voice.part,
          data = singer,
          nint = 17,
          endpoints = c(59.5, 76.5),
          layout = c(2, 4),
          aspect = 1,
          sub = list("Figure 1.2",cex=.8),
          xlab = "Height (inches)")

book.1.2()
```

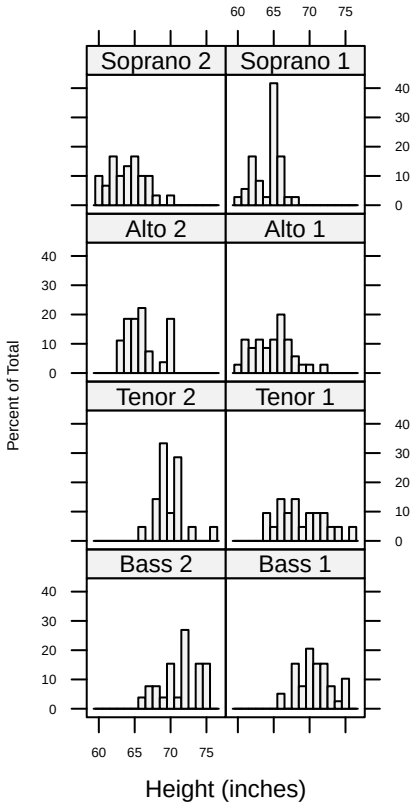
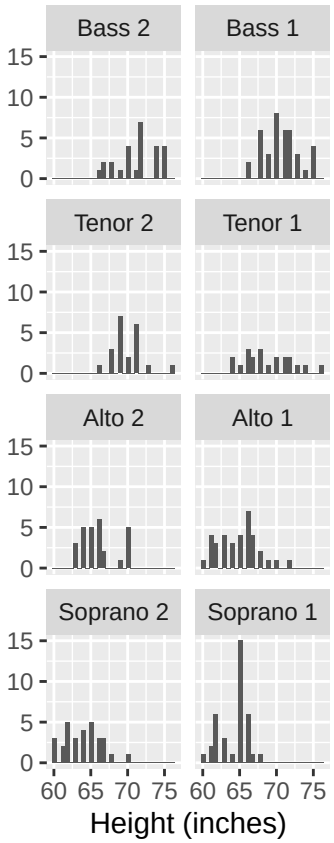


Figure 1.2

```
ggVD_1.2 <- function(){
    p <- qplot(height, data = singer, xlab = "Height (inches)") +
        geom_histogram() + facet_wrap( ~ voice.part, ncol = 2)
    print(p+theme(aspect.ratio = 1))
}
    ggVD_1.2()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



- Quantile plot

```
book.2.2 <-
    function()
    qqmath(~ sort(singer$height[singer$voice.part=="Tenor 1"]),
               distribution = qunif,
               panel = function(x, ...) {
                   panel.qqmath(x, type = "b", ...)
                   ##panel.qqmath(x, col = 0, pch = 16)
                   ##panel.qqmath(x, ...)
               },
               aspect = 1,
               sub = list("Figure 2.2",cex=.8),
               xlab = "f-value",
               ylab = "Tenor 1 Height (inches)")

book.2.2()
```

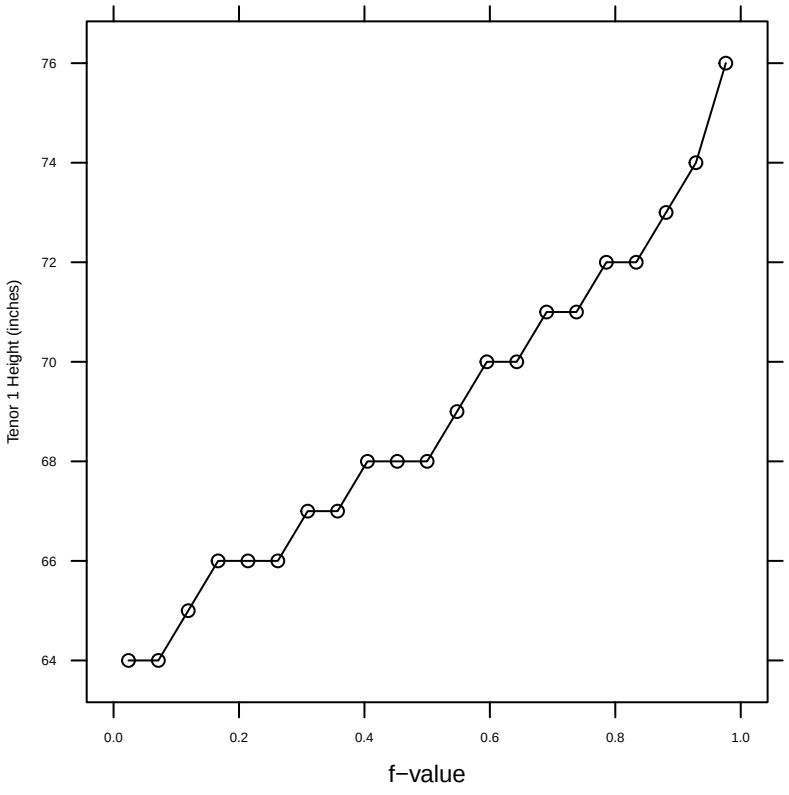
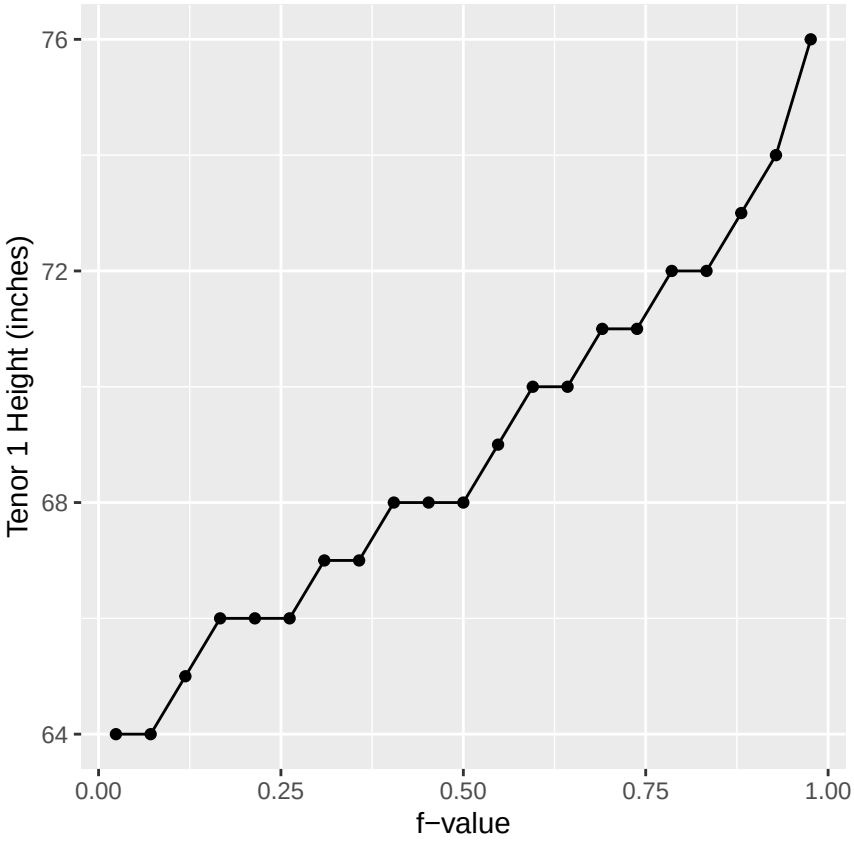


Figure 2.2

```
ggVD_2.2 <- function(){
    fval.df <- function(x){
        oo <- order(x)
        n <- length(x)
        f <- ((1:n)-0.5)/n
        return(data.frame(value=x[oo], q=f))
    }
    Tenor1 <- fval.df(singer$height[singer$voice.part=="Tenor 1"])
    ggplot(Tenor1, aes(q, value)) + geom_point() + geom_path() +
        labs(x = "f-value",
             y = "Tenor 1 Height (inches)") +
        theme(aspect.ratio=1)
}

ggVD_2.2()
```




```
book.2.1 <-
    function()
    qqmath(~ height | voice.part,
               distribution=qunif,
               data=singer,
               panel = function(x, ...) {
                   panel.grid()
                   panel.qqmath(x, ...)
               },
               layout=c(2,4),
               aspect=1,
               sub = list("Figure 2.1",cex=.8),
               xlab = "f-value",
               ylab="Height (inches)")

book.2.1()
```

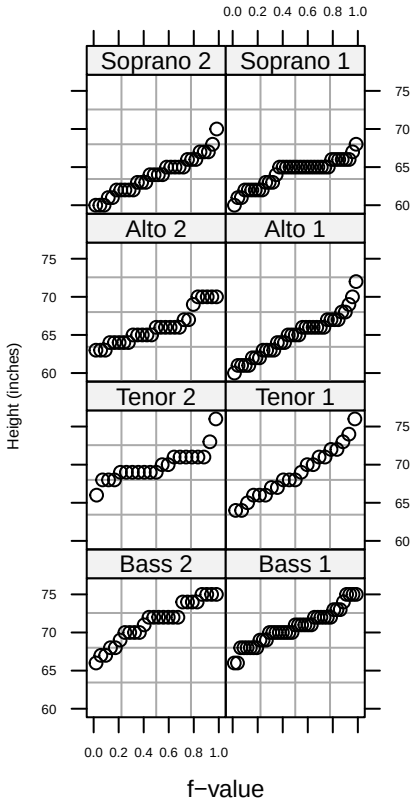
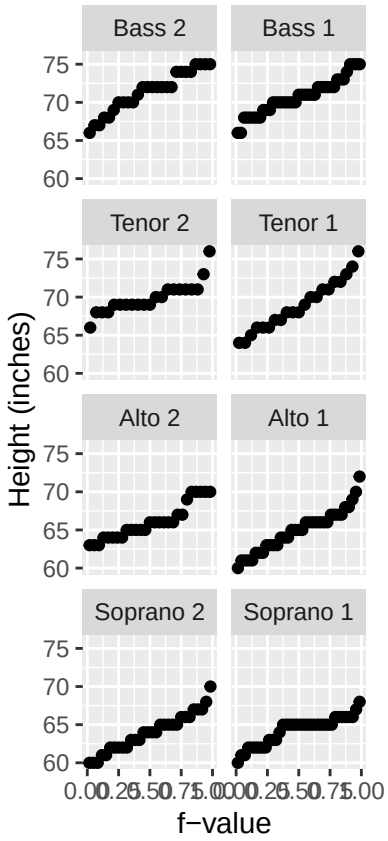
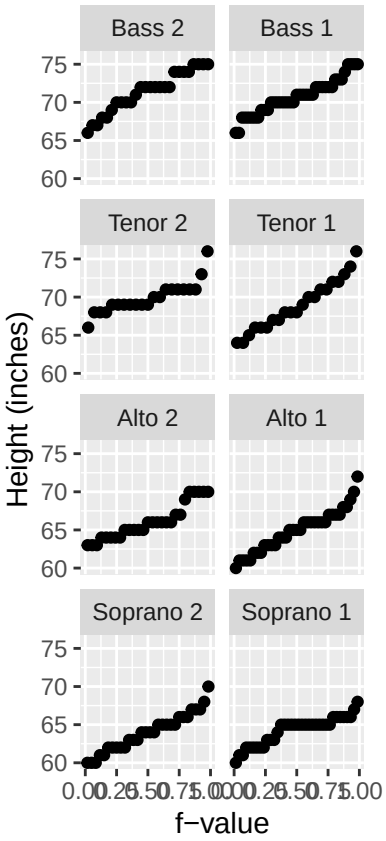


Figure 2.1

```
ggVD_2.1 <- function(){
ggplot(singer, aes(sample=height))+
    stat_qq(distribution=qunif)+
    facet_wrap(~voice.part, ncol=2)+
    labs(x = "f-value",
         y = "Height (inches)") + theme(aspect.ratio=1)
}

ggVD_2.1()
```




```
ggVD_2.1alt <- function(){
fval.df <- function(x){
        oo <- order(x)
        n <- length(x)
        f <- ((1:n)-0.5)/n
        return(data.frame(value=x[oo], q=f))
    }
    singer.q <- plyr::ddply(singer, "voice.part",
                        function(df) fval.df(df$height))
    ggplot(singer.q, aes(q, value))+geom_point()+
        facet_wrap(.~voice.part, ncol=2) +
        labs(  x = "f-value",
               y="Height (inches)") + theme(aspect.ratio=1)
}

ggVD_2.1alt()
```



- Quantile-quantile plot

```
book.2.3 <-
function()
{
    voice.part <- ordered(singer$voice.part,
        c("Soprano 1", "Soprano 2", "Alto 1", "Alto 2",
            "Tenor 1", "Tenor 2", "Bass 1", "Bass 2"))
    obj<- qq(voice.part ~ singer$height,
        subset=voice.part=="Bass 2" | voice.part=="Tenor 1",
        aspect=1,
        sub = list("Figure 2.3",cex=.8),
        xlab = "Tenor 1 Height (inches)",
        ylab = "Base 2 Height (inches)")
    print(obj)
}
book.2.3()
```

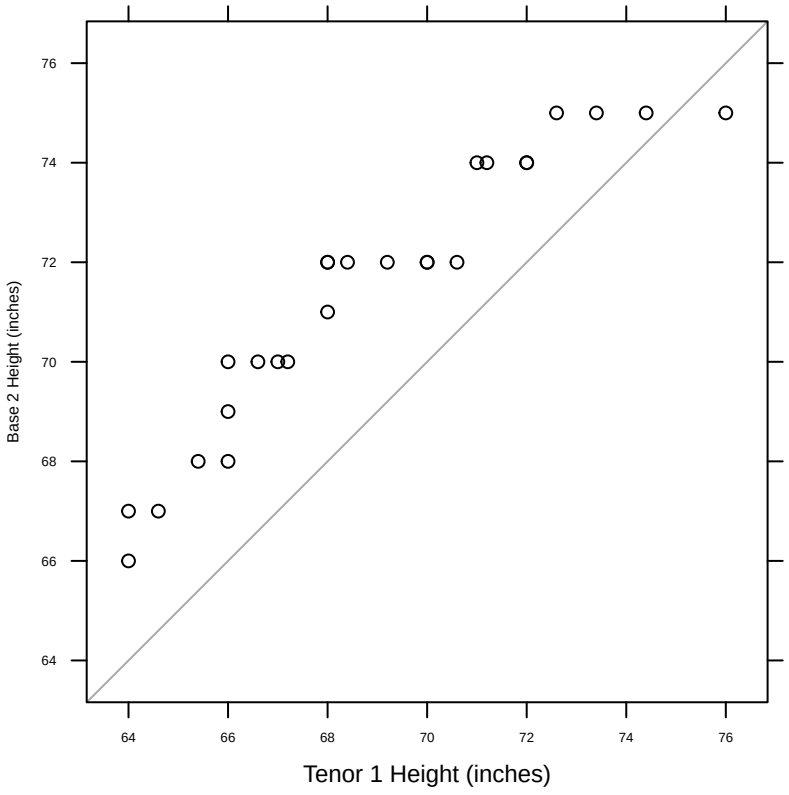
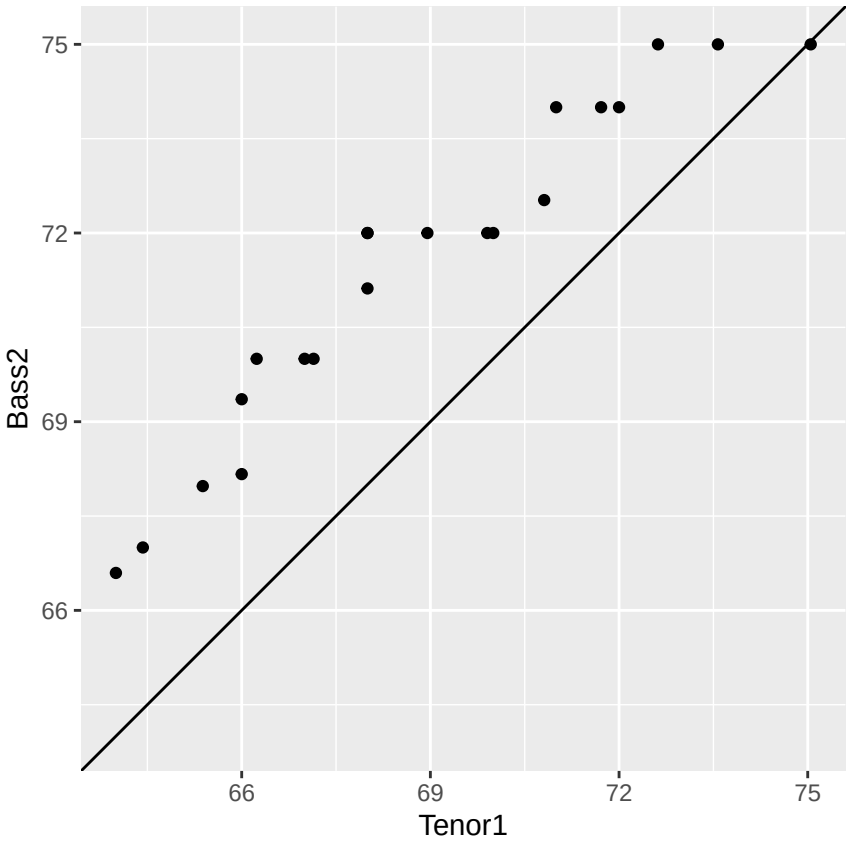


Figure 2.3

```
ggVD_2.3 <- function(){
    q <- function(x, probs = ppoints(100)) {
        data.frame(q = probs, value = quantile(x, probs))
    }
    temp <- singer[singer$voice.part=="Bass 2" |
                   singer$voice.part=="Tenor 1", ]
    n <- min(c(sum(singer$voice.part=="Bass 2"),
               sum(singer$voice.part=="Tenor 1")))
    temp.q <- plyr::ddply(temp, "voice.part",
                      function(df) q(df$height, ppoints(n)))
    temp.df <- reshape2::recast(temp.q, q ~ voice.part, id.var = c(2,1))
    names(temp.df) <- c("q", "Bass2", "Tenor1")
    pg <- ggplot(temp.df, aes(Tenor1, Bass2)) +
        geom_point() + geom_abline() +
        coord_equal(xlim=range(temp.q$value),
                    ylim=range(temp.q$value))
    print(pg+theme(aspect.ratio=1))
}
ggVD_2.3()
```




```
book.2.4 <-
function()
{
    voice.part <- ordered(singer$voice.part,
        c("Soprano 1", "Soprano 2", "Alto 1", "Alto 2",
            "Tenor 1", "Tenor 2", "Bass 1", "Bass 2"))
    bass.tenor.qq <- qq(voice.part ~ singer$height,
        subset=voice.part=="Bass 2" | voice.part=="Tenor 1")
    tmd(bass.tenor.qq,
        aspect=1,
        ylab = "Difference (inches)",
        sub = list("Figure 2.4",cex=.8),
        xlab = "Mean (inches)")
}
book.2.4()
```

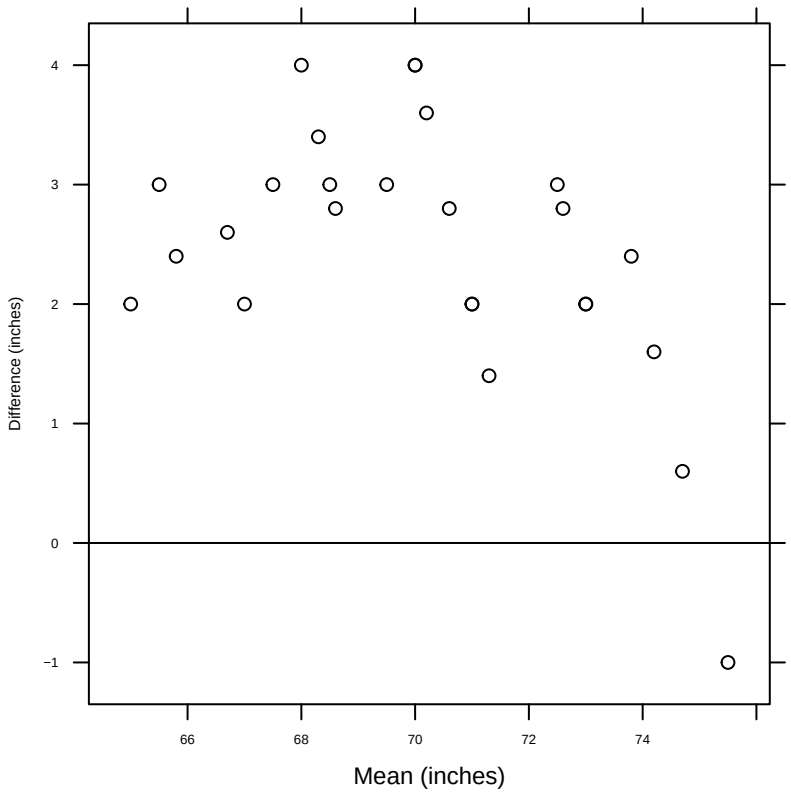
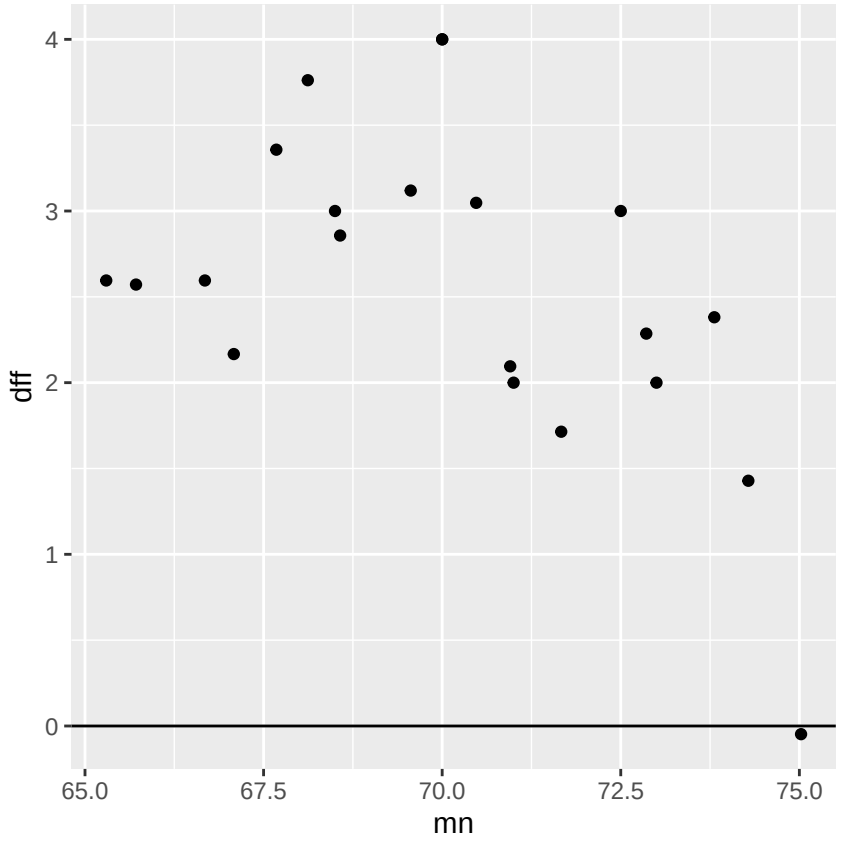


Figure 2.4

```
ggVD_2.4 <- function(){
    q <- function(x, probs = ppoints(100)) {
        data.frame(q = probs, value = quantile(x, probs))
    }
    temp <- singer[singer$voice.part=="Bass 2" |
                   singer$voice.part=="Tenor 1", ]
    n <- min(c(sum(singer$voice.part=="Bass 2"),
               sum(singer$voice.part=="Tenor 1")))
    temp.q <- plyr::ddply(temp, "voice.part",
                    function(df) q(df$height, ppoints(n)))
    temp.df <- reshape2::recast(temp.q, q ~ voice.part, id.var = c(2,1))
    names(temp.df) <- c("q", "Bass2", "Tenor1")
    temp.df$mn <- (temp.df$Bass2+temp.df$Tenor1)/2
    temp.df$dff <- temp.df$Bass2-temp.df$Tenor1
    pg <- ggplot(temp.df, aes(mn,dff)) +
        geom_point() + geom_hline(yintercept=0)
    print(pg+theme(aspect.ratio=1))
}
ggVD_2.4()
```



- Box plot

```
book.2.6 <-
function(){
    oldpty <- par("pty")
    par(pty = "s")
    data <-
    c(0.9, 1.6, 2.26305, 2.55052, 2.61059, 2.69284, 2.78511, 2.80955,
        2.94647, 2.96043, 3.05728, 3.15748, 3.18033, 3.20021,
        3.20156, 3.24435, 3.33231, 3.34176, 3.3762, 3.39578, 3.4925,
        3.55195, 3.56207, 3.65149, 3.72746, 3.73338, 3.73869,
        3.80469, 3.85224, 3.91386, 3.93034, 4.02351, 4.03947,
            4.05481, 4.10111, 4.26249, 4.28782, 4.37586, 4.48811,
        4.6001, 4.65677, 4.66167, 4.73211, 4.80803, 4.9812, 5.17246,
        5.3156, 5.35086, 5.36848, 5.48167, 5.68, 5.98848, 6.2, 7.1,
        7.4)
    boxplot(data, rep(NA, length(data)), ylab = "Data")
    usr <- par("usr")
    x <- usr[1] + (usr[2] - usr[1]) * 0.5
    at <- c(0.9, 1.6, 3.2, 3.8, 4.65, 6.2, 7.2)
    arrows(rep(x * 1.15, 7), at, rep(x, 7), at)
    mtext("Figure 2.6",1,1,cex=.8)
    text(rep(x * 1.2, 7), at, adj = 0,
        labels = c("outside value", "lower adjacent value",
            "lower quartile", "median", "upper quartile",
            "upper adjacent value", "outside values"))
    par(pty = oldpty)
```

```
    invisible()
}

book.2.6()
```

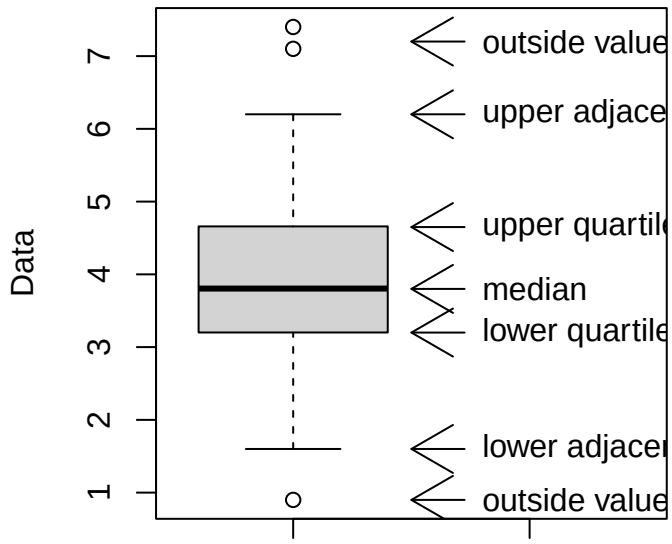
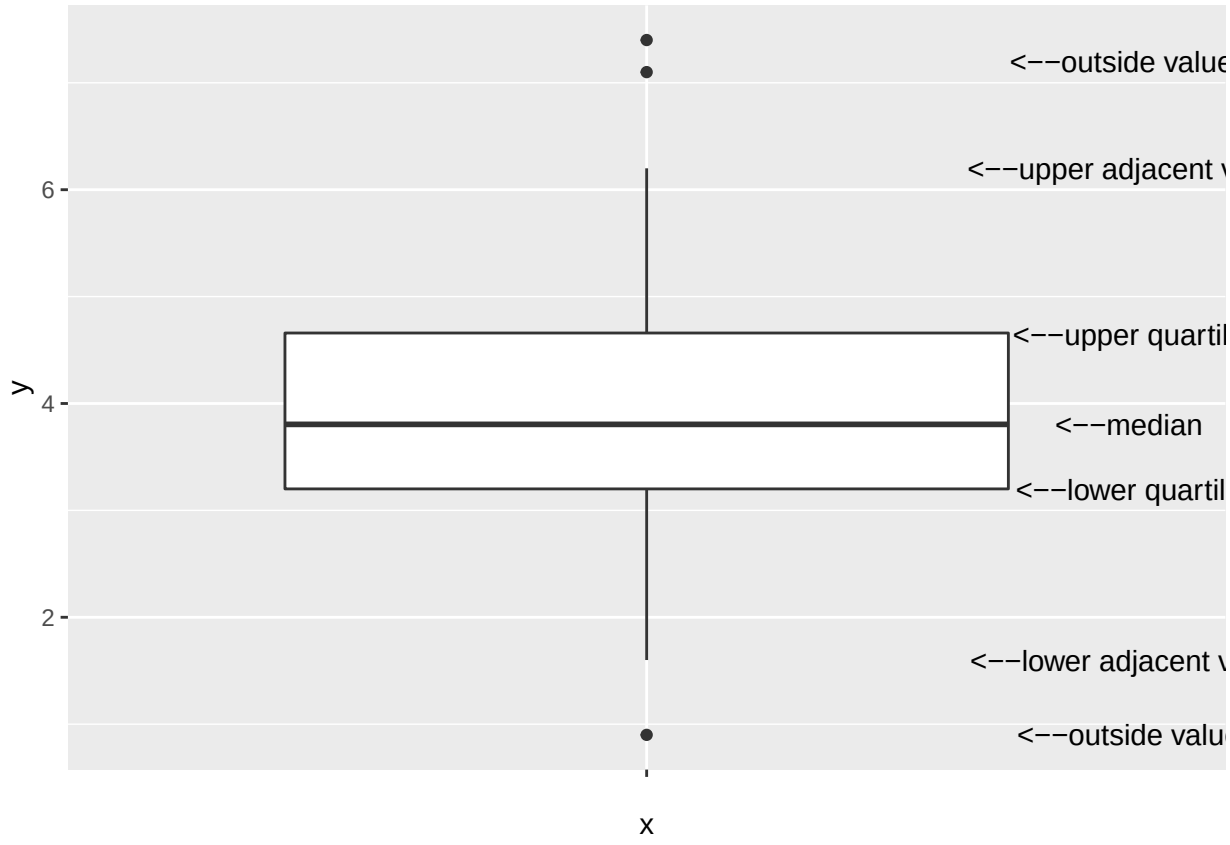


Figure 2.6

```
ggVD_2.6 <- function(){
    ydata <- c(0.9, 1.6, 2.26305, 2.55052,
               2.61059, 2.69284, 2.78511, 2.80955, 2.94647, 2.96043,
               3.05728, 3.15748, 3.18033, 3.20021, 3.20156, 3.24435,
               3.33231, 3.34176, 3.37620, 3.39578, 3.49250, 3.55195,
               3.56207, 3.65149, 3.72746, 3.73338, 3.73869, 3.80469,
               3.85224, 3.91386, 3.93034, 4.02351, 4.03947, 4.05481,
               4.10111, 4.26249, 4.28782, 4.37586, 4.48811, 4.6001,
               4.65677, 4.66167, 4.73211, 4.80803, 4.9812, 5.17246,
               5.3156, 5.35086, 5.36848, 5.48167, 5.68, 5.98848, 6.2,
               7.1, 7.4)
    text <- c("outside value", "lower adjacent value",
              "lower quartile", "median", "upper quartile",
              "upper adjacent value", "outside values")
    at <- c(0.9, 1.6, 3.2,3.8, 4.65, 6.2, 7.2)

    data = data.frame(y = ydata, x = 1)

    p <- ggplot(data, aes(x ="", y)) + stat_boxplot() +
    annotate("text", x = 1.5, y = at, label = paste("<--", text, sep = ""))
```

```
    print(p)
}

ggVD_2.6()
```




```
book.2.7 <-
function()
{
    data <- round(c(0.9, 1.6, 2.263047,
        2.550518, 2.610592, 2.69284, 2.785113,
        2.809547, 2.946467, 2.96044, 3.057283,
        3.15748, 3.180327, 3.200206,
        3.20156, 3.244347, 3.332312,
        3.341763, 3.3762, 3.395778, 3.492497,
        3.551945, 3.562066, 3.65149,
        3.7274632, 3.73338, 3.738686, 3.80469,
        3.85224, 3.91386, 3.93034,
        4.02351, 4.039466, 4.05481, 4.101108, 4.262486,
        4.28782, 4.375864, 4.48811, 4.6001,
        4.656775, 4.661673, 4.73211,
        4.80803, 4.9812, 5.172464,
        5.3156, 5.35086, 5.36848,
        5.48167, 5.68, 5.98848, 6.2,
        7.1, 7.4),5)
    uq <- quantile(data,.75)
```

```
    lq <- quantile(data,.25)
    r <- 1.5*(uq-lq)
    h <- c(lq-r,1.6,lq,uq,6.2,uq+r)
    writing <- c("lower quartile - 1.5 r",
        "lower adjacent value",
        "lower quartile",
        "upper quartile",
         "upper adjacent value",
         "upper quartile + 1.5 r")
    qqmath(~ data,
        distribution = qunif,
        panel = substitute(function(x,...) {
            reference.line <- trellis.par.get("reference.line")
            panel.abline(h = h, lwd = reference.line$lwd, lty = reference.line$lty, col = reference.lin
            panel.qqmath(x, type="b", ...)
            panel.text(rep(0,3), h[4:6], writing[4:6], adj=0)
            panel.text(rep(1,3), h[1:3], writing[1:3], adj=1)
        }),
        aspect = 1,
        sub = list("Figure 2.7",cex=.8),
        xlab = "f-value",
        ylab = "Data")
}

book.2.7()
```

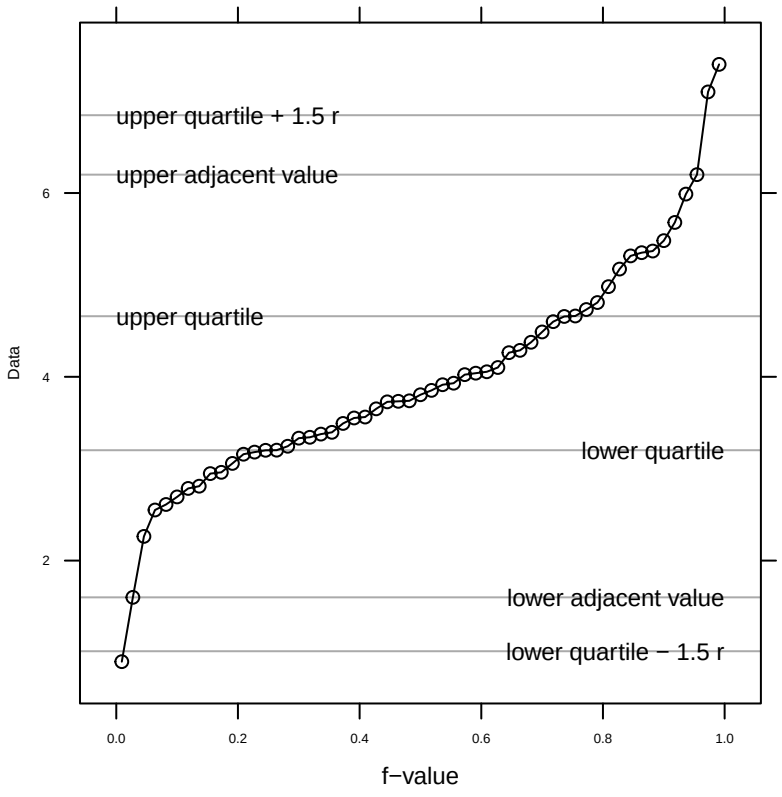
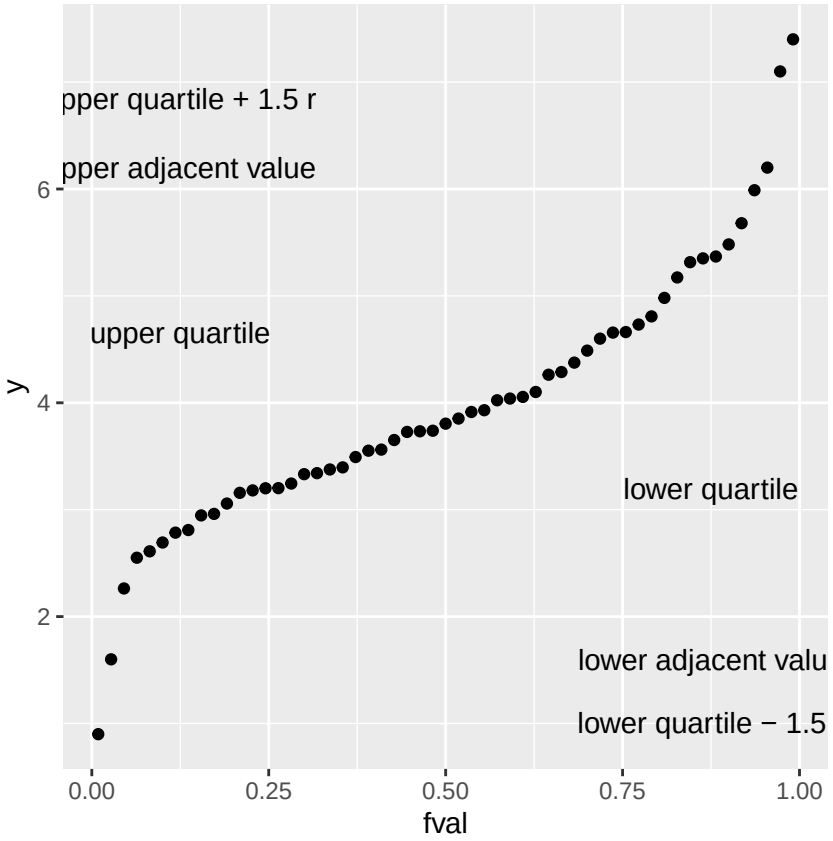


Figure 2.7

```
ggVD_2.7 <- function(){
    data <- round(c(0.9, 1.6, 2.263047,
                    2.550518, 2.610592, 2.69284, 2.785113,
                    2.809547, 2.946467, 2.96044, 3.057283,
                    3.15748, 3.180327, 3.200206,
                    3.20156, 3.244347, 3.332312,
                    3.341763, 3.3762, 3.395778, 3.492497,
                    3.551945, 3.562066, 3.65149,
                    3.7274632, 3.73338, 3.738686, 3.80469,
                    3.85224, 3.91386, 3.93034,
                    4.02351, 4.039466, 4.05481, 4.101108, 4.262486,
                    4.28782, 4.375864, 4.48811, 4.6001,
                    4.656775, 4.661673, 4.73211,
                    4.80803, 4.9812, 5.172464,
                    5.3156, 5.35086, 5.36848,
                    5.48167, 5.68, 5.98848, 6.2,
                    7.1, 7.4),5)
    n <- length(data)
    uq <- quantile(data,.75)
    lq <- quantile(data,.25)
    r <- 1.5*(uq-lq)
    h <- c(lq-r,1.6,lq,uq,6.2,uq+r)
    writing <- c("lower quartile - 1.5 r",
                 "lower adjacent value",
                 "lower quartile",
                 "upper quartile",
         "upper adjacent value",
         "upper quartile + 1.5 r")
    data = data.frame(y = data, fval=((1:n)-0.5)/n)

    p<- ggplot(data, aes(fval, y)) + geom_point() +
        annotate("text", x = 0.125, y = h[4:6], label = writing[4:6]) +
        annotate("text", x = 0.875, y = h[1:3], label = writing[1:3])
    print(p + theme(aspect.ratio=1))
}

ggVD_2.7()
```




```
book.2.8 <-
    function()
    bwplot(voice.part ~ height,
                data=singer,
                aspect=1,
                sub = list("Figure 2.8",cex=.8),
                xlab="Height (inches)")

book.2.8()
```

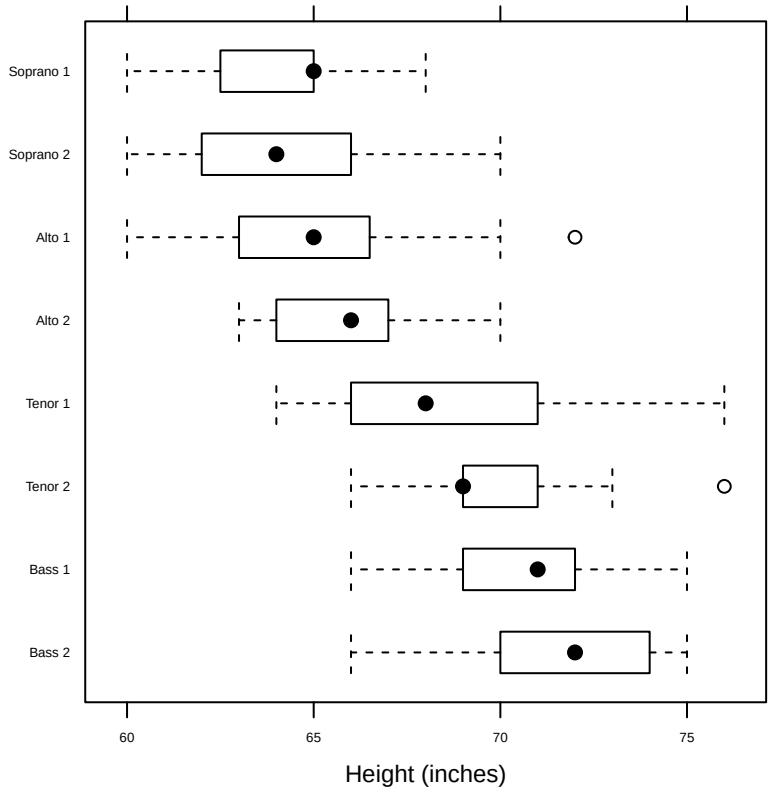
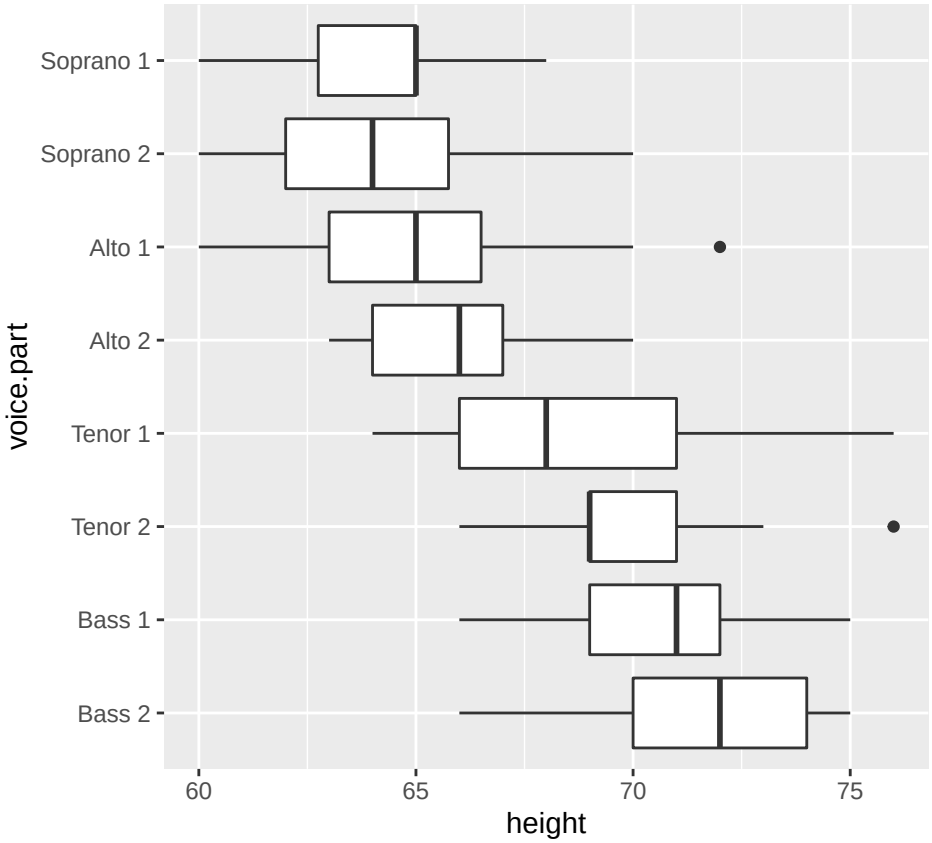


Figure 2.8

```
ggVD_2.8 <- function(){
    ggplot(singer, aes(voice.part, height)) + geom_boxplot() +
        coord_flip()+labs(xlab = " Height (inches)")+theme(aspect.ratio=1)

}
ggVD_2.8()
```



- Normal Q-Q plot First, let's take a lok of the basic Q-Q plot

```
book.2.9 <- function(){
    data <- sort(singer$height[singer$voice.part=="Alto 1"])
    qqmath(~ data, #qqmath computes the fvalue
           distribution = qunif,
           panel = function(x, ...) {
               panel.grid()
               panel.qqmath(x, type="b", ...)
           },
           aspect = 1,
           ylim = range(data, qnorm(ppoints(data), mean(data),
                                    sqrt(var(data)))),
           sub = list("Figure 2.9",cex=.8),
           xlab = "f-value",
           ylab = "Alto 1 Height (inches)")
}

book.2.9()
```

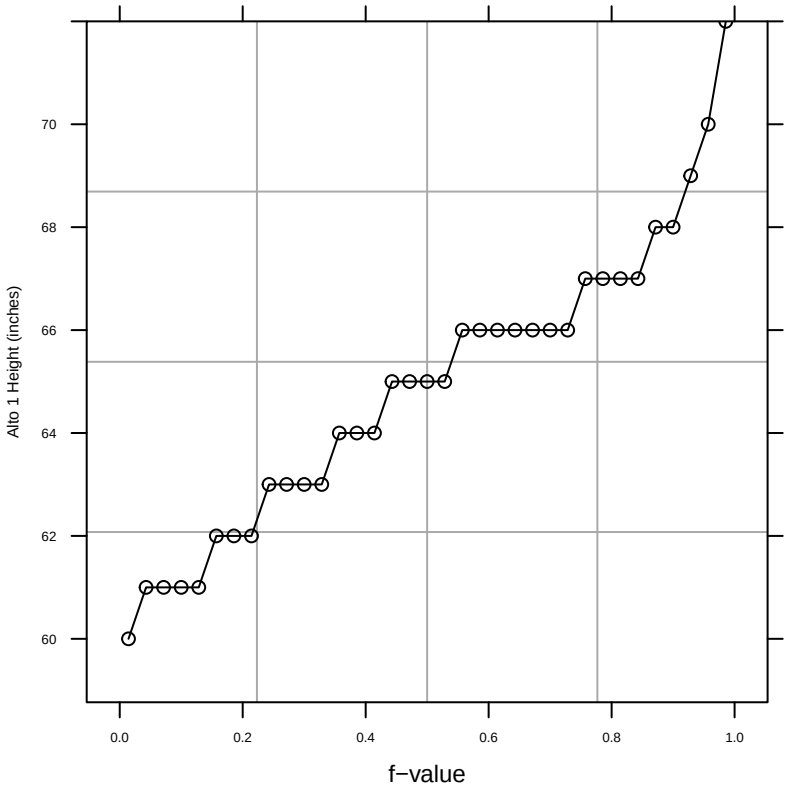
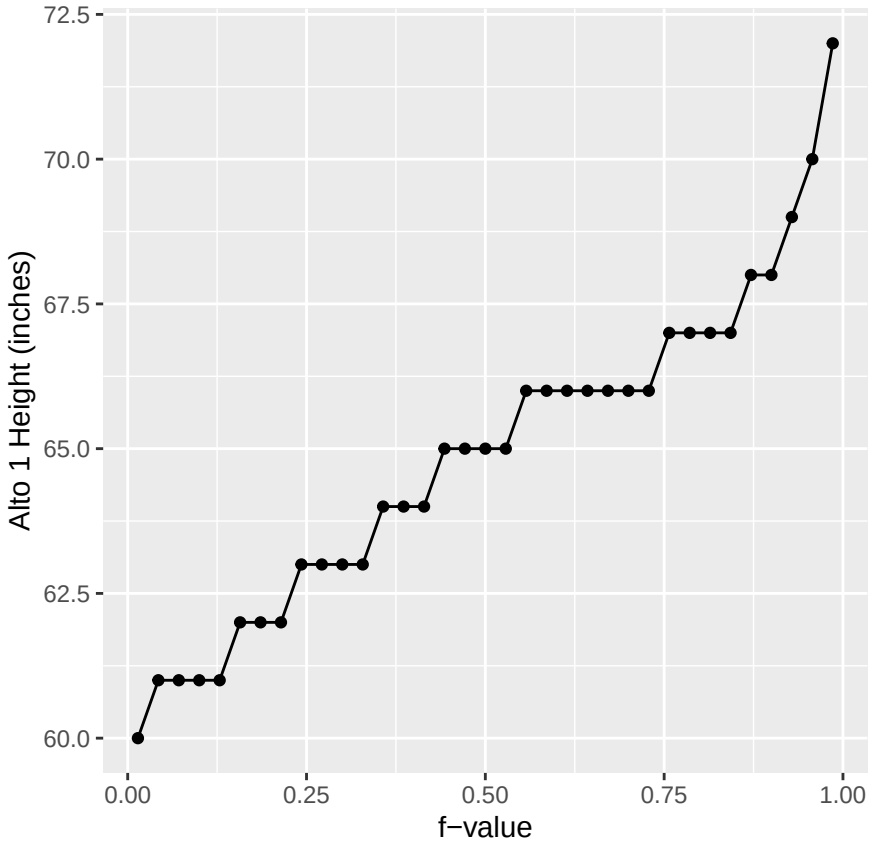


Figure 2.9

```
ggVD_2.9 <- function(){
    fval <- function(x){
        oo <- order(x)
        return((((1:max(oo))-0.5)/max(oo))[oo])
    }
    data <- (sort(singer$height[singer$voice.part=="Alto 1"]))
    alto1 <- data.frame(height= data)
    alto1$f <- fval(data)

    ggplot(alto1, aes(f, height)) +geom_point() +
        labs(y = "Alto 1 Height (inches)", x = "f-value") +
        geom_path() + theme(aspect.ratio=1)
}
ggVD_2.9()
```




```
book.2.10 <- function(){
    data <- sort(singer$height[singer$voice.part=="Alto 1"])
    x <- ppoints(data)
    y <- qnorm(x, mean(data), sqrt(var(data)))
    xyplot(y ~ x,
           panel = function(x, y){
               panel.grid()
               panel.xyplot(x, y, type = "l")
           },
           ylim = range(data, y),
           aspect = 1,
           sub = list("Figure 2.10",cex=.8),
           xlab = "f-value",
           ylab = "Normal Quantile Function")
}

book.2.10()
```

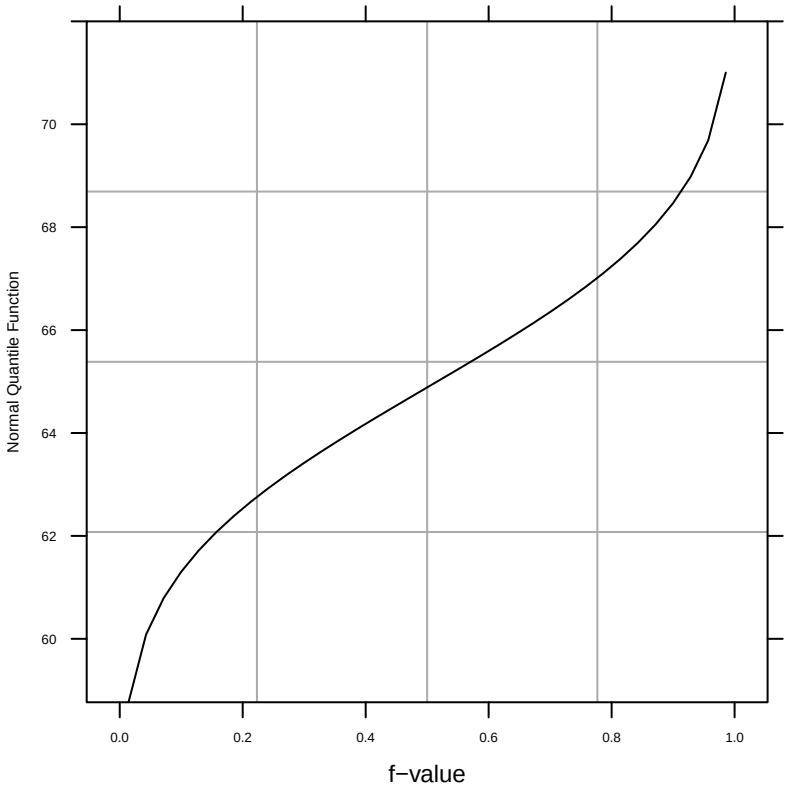
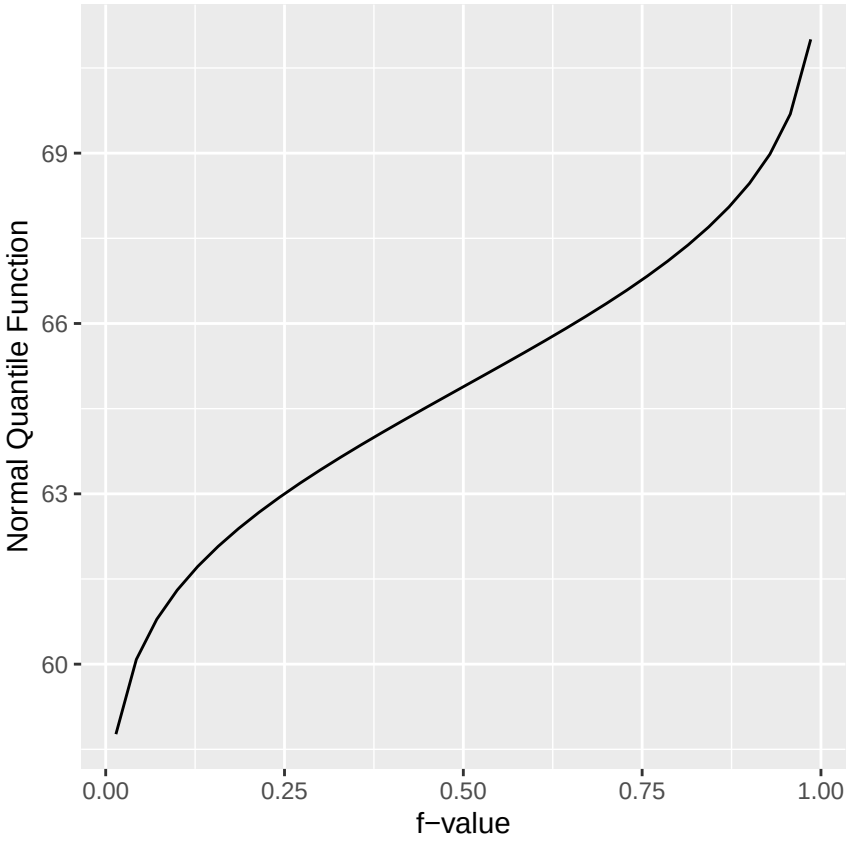


Figure 2.10

```
ggVD_2.10 <- function(){
    data <- sort(singer$height[singer$voice.part=="Alto 1"])
    x <- ppoints(data)
    y <- qnorm(x, mean(data), sqrt(var(data)))

    ggplot(data.frame(x, y), (aes(x, y))) + geom_line() +
        labs(y = "Normal Quantile Function", x = "f-value")+
        theme(aspect.ratio=1)
}
ggVD_2.10()
```



Now, the relationship between the f-values of the two plots.

$$q_{\mu,\sigma}(f) = \mu + \sigma q_{0,1}(f)$$

```
book.2.11 <- function(){
    qqmath(~ height | voice.part,
           data=singer,
           prepanel = prepanel.qqmathline,
           panel = function(x, ...) {
               panel.grid()
               panel.qqmathline(x, distribution = qnorm)
               panel.qqmath(x, ...)
           },
           layout=c(2,4),
           aspect=1,
           sub = list("Figure 2.11",cex=.8),
           xlab = "Unit Normal Quantile",
           ylab="Height (inches)")

}
book.2.11()
```

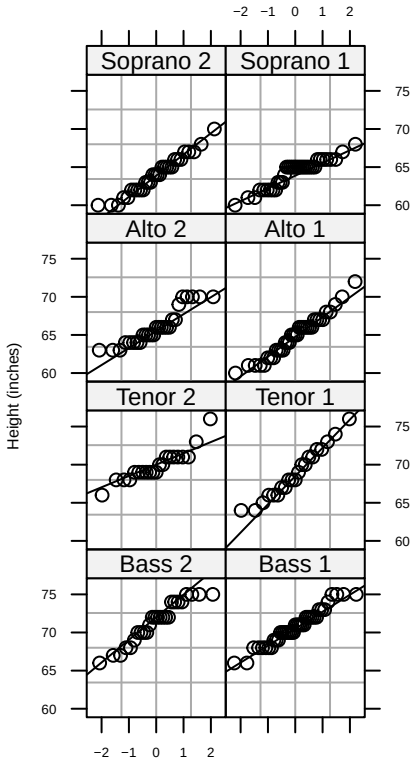
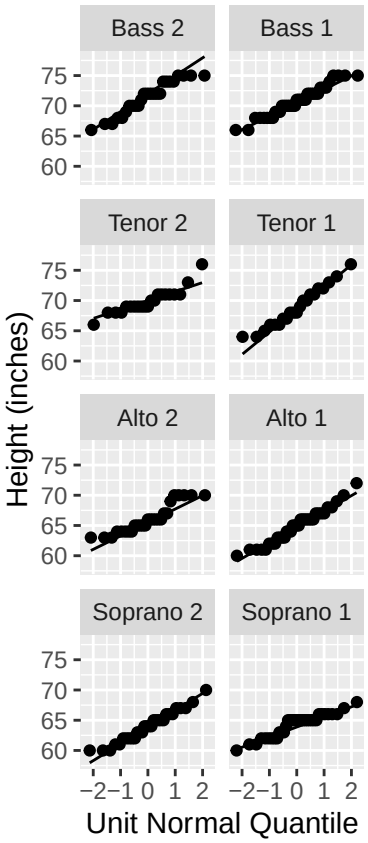


Figure 2.11

```
ggVD_2.11 <- function()
    ggplot(singer, aes(sample = height)) + stat_qq(distribution = qnorm)+
        geom_qq_line()+ facet_wrap(. ~ voice.part, ncol = 2) +
        labs(x = "Unit Normal Quantile", y = "Height (inches)") +
        theme(aspect.ratio=1)
ggVD_2.11()
```



9.3.2 Fits and Residuals

```
book.2.12 <- function(){
    dotplot(tapply(singer$height,singer$voice.part,mean),
            aspect=1,
            sub = list("Figure 2.12",cex=.8),
            xlab="Mean Height (inches)")
}

book.2.12()
```

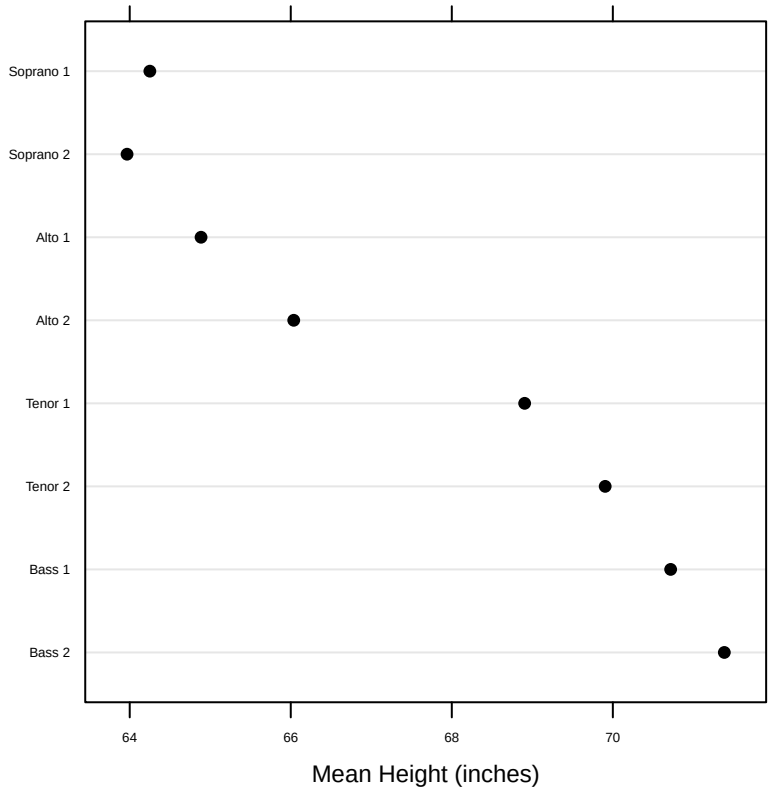
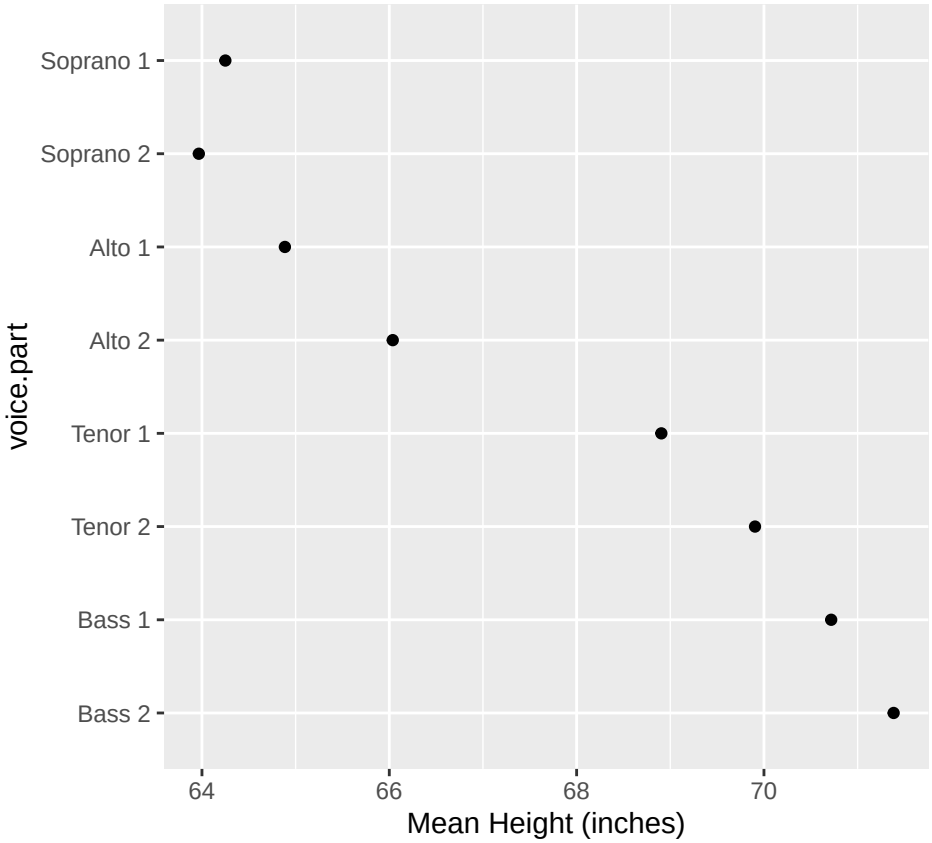


Figure 2.12

```
ggVD_2.12 <- function(){
    meandot <- data.frame(tapply(singer$height, singer$voice.part, mean))
    singer.means = data.frame(
        voice.part=ordered(rownames(meandot),
                           levels=rev(c("Soprano 1", "Soprano 2",
                                        "Alto 1", "Alto 2" ,
                                        "Tenor 1", "Tenor 2",
                                        "Bass 1", "Bass 2"))),
        height=meandot[,1])

    ggplot(singer.means, aes(height,voice.part)) + geom_point() +
        labs(x = "Mean Height (inches)")  + theme(aspect.ratio=1)
}

ggVD_2.12()
```



Fitted values versus residuals

$$h_{pi} = \hat{h}_{pi} + \hat{\varepsilon}_{pi}$$

```
book.2.13 <- function(){
    bwplot(voice.part ~ oneway(height~voice.part, spread = 1)$residuals,
           data = singer,
           aspect=0.75,
           panel = function(x,y){
               panel.bwplot(x,y)
               panel.abline(v=0)
           },
           sub = list("Figure 2.13",cex=.8),
           xlab = "Residual Height (inches)")
}

book.2.13()
```

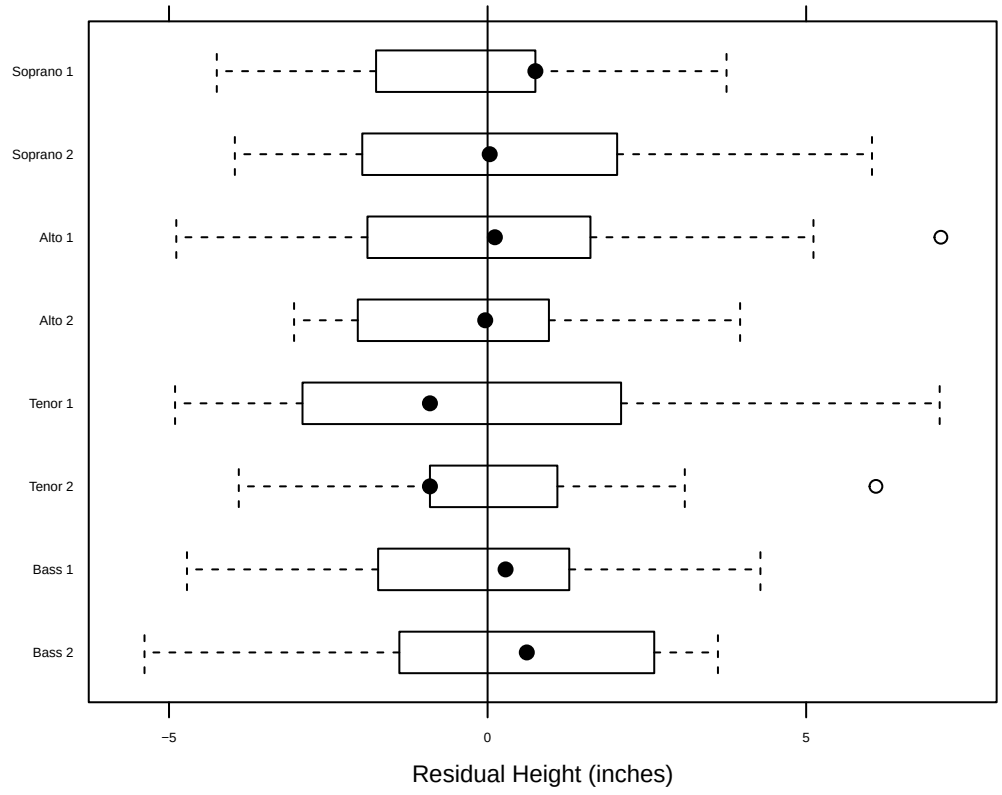
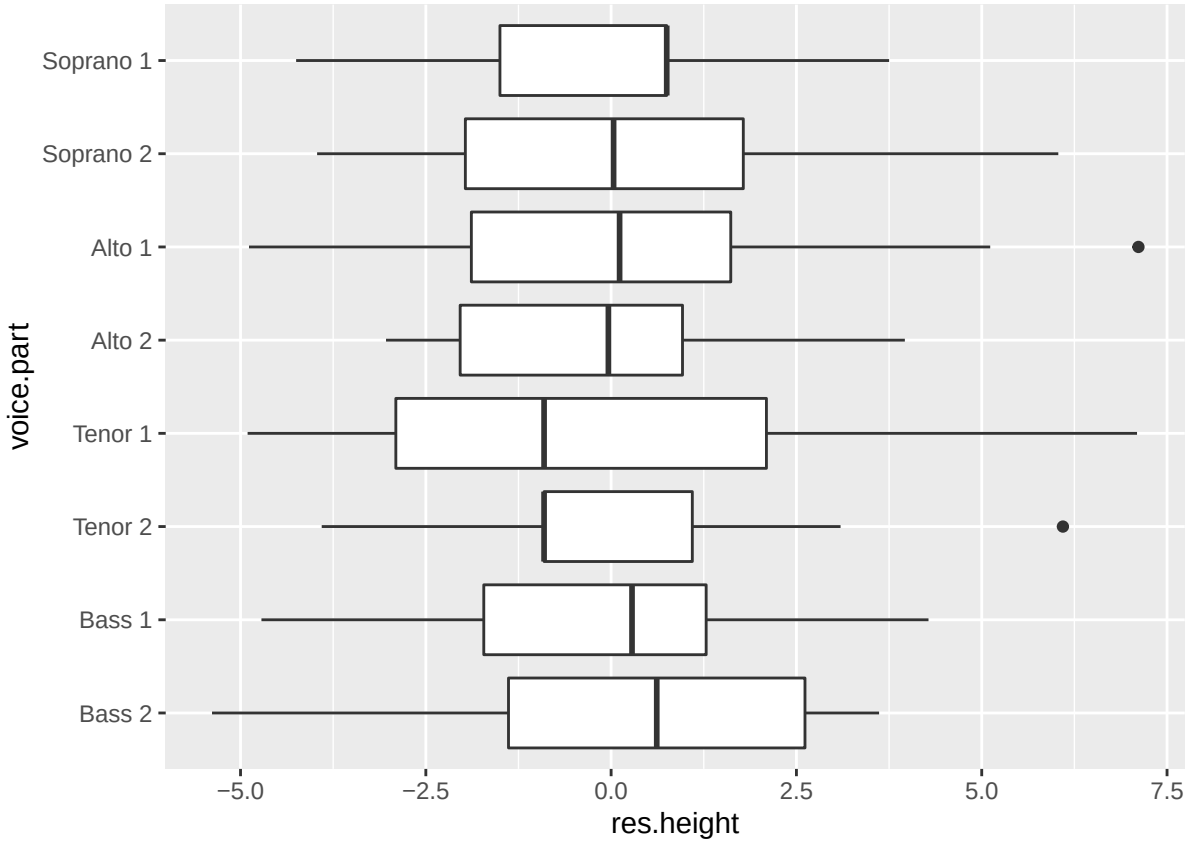


Figure 2.13

```
ggVD_2.13 <- function(){
    res.height <- oneway(height ~ voice.part, data = singer,
                         spread = 1)$residuals

    ggplot(singer, aes(voice.part, res.height)) + geom_boxplot() +
        coord_flip( )+ labs(xlab = "Residual Height (inches)") +
        theme(aspect.ratio=0.75)
}
ggVD_2.13()
```



9.3.3 Homogeneity and Pooling

```
book.2.14 <- function()
{
    res.height <- oneway(height ~ voice.part, data = singer,
                         spread = 1)$residuals
    qqmath(~ res.height | singer$voice.part,
           distribution = substitute(function(p) quantile(res.height, p)),
           panel=function(x){
               panel.grid()
               panel.qqmathline(x)
               panel.qqmath(x)
           },
           aspect=1,
           layout=c(2,4),
           sub = list("Figure 2.14",cex=.8),
           xlab = "Pooled Residual Height (inches)",
           ylab = "Residual Height (inches)")
}
book.2.14()
```

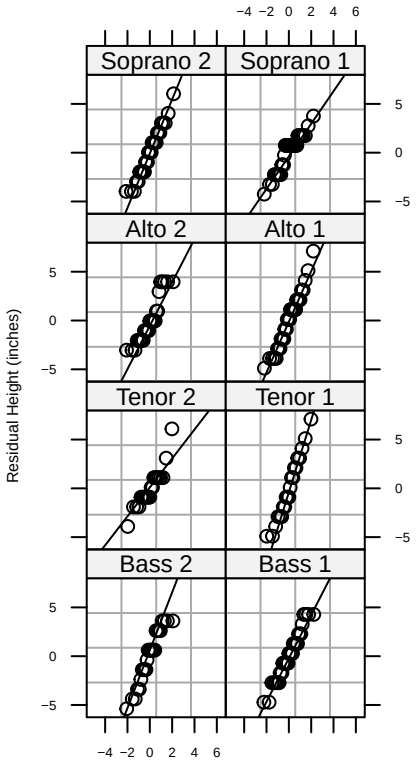
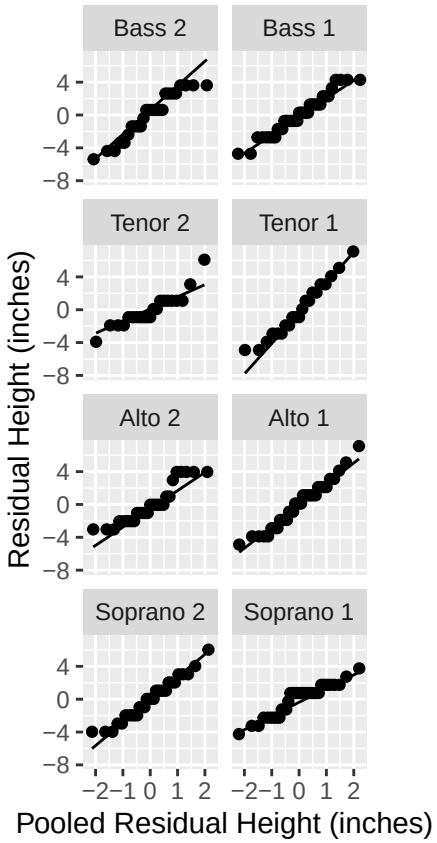


Figure 2.14

```
ggVD_2.14 <- function(){
    res.height <- oneway(height ~ voice.part, data = singer,
                         spread = 1)$residuals

    ggplot(singer, aes(sample =res.height )) +
        stat_qq(distribution = qnorm) +
        geom_qq_line() +
        facet_wrap(~voice.part, ncol = 2) +
        labs(x = "Pooled Residual Height (inches)",
             y = "Residual Height (inches)") +
        theme(aspect.ratio=1)
}
ggVD_2.14()
```



9.3.4 Checking for normality of the pooled residuals

```
book.2.15 <- function()
{
    qqmath(~ oneway(height ~ voice.part, spread = 1)$residuals,
           data = singer,
           distribution = qunif,
           aspect = 1,
           sub = list("Figure 2.15",cex=.8),
           xlab = "f-value",
           ylab = "Residual Height (inches)")
}

book.2.15()
```

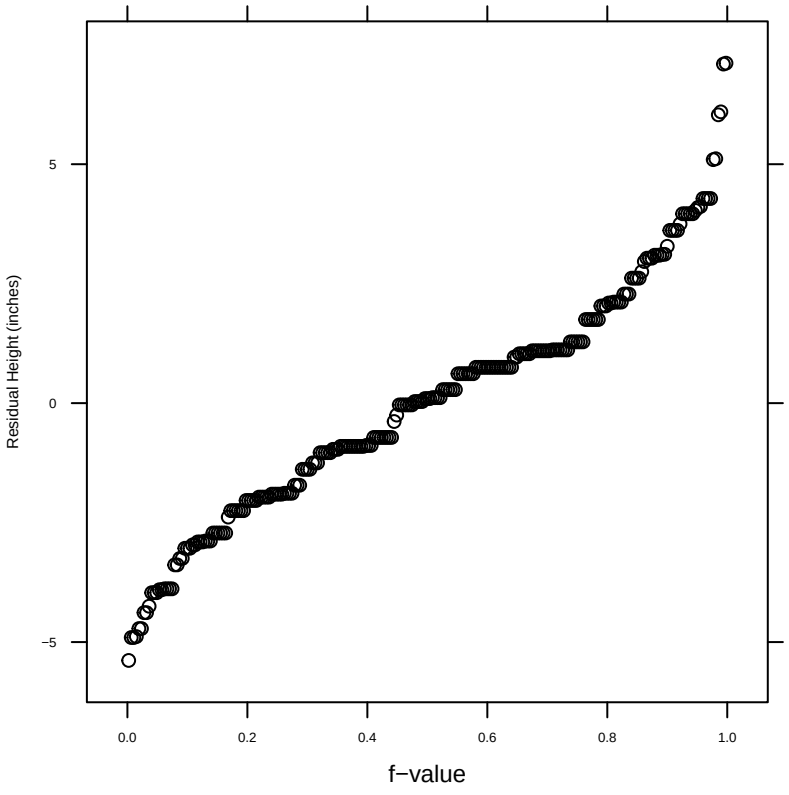
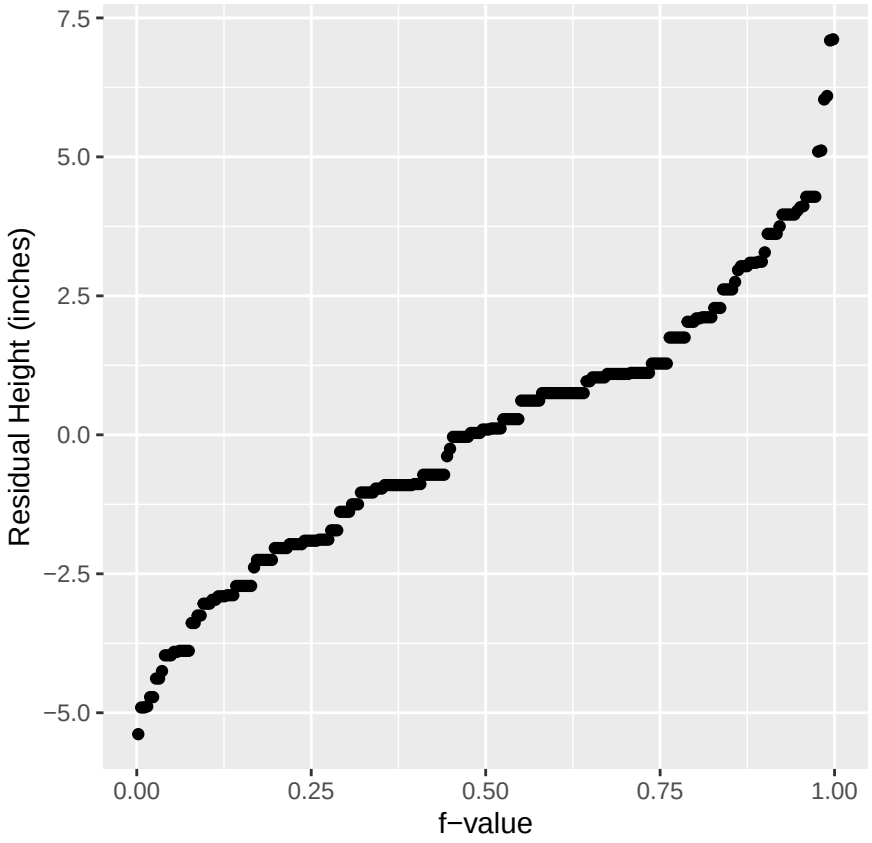


Figure 2.15

```
ggVD_2.15 <- function()
{
    res.height.f <- ((1:dim(singer)[1]) -.5)/(dim(singer)[1])
    singer$res.height.f <- res.height.f
    singer$res.height <- oneway(height ~ voice.part,
                                data = singer, spread = 1)$residuals

    ggplot(singer, aes(sample = res.height)) +
        stat_qq(distribution = qunif) +
        labs(x = "f-value", y = "Residual Height (inches)")+
        theme(aspect.ratio=1)
}

ggVD_2.15()
```




```
book.2.16 <- function(){
    qqmath(~ oneway(height~voice.part, spread = 1)$residuals,
           data = singer,
           prepanel = prepanel.qqmathline,
           panel = function(x, ...) {
               panel.grid()
               panel.qqmathline(x, distribution = qnorm)
               panel.qqmath(x,...)
           },
           aspect=1,
           sub = list("Figure 2.16",cex=.8),
           xlab = "Unit Normal Quantile",
           ylab="Residual Height (inches)")
}

book.2.16()
```

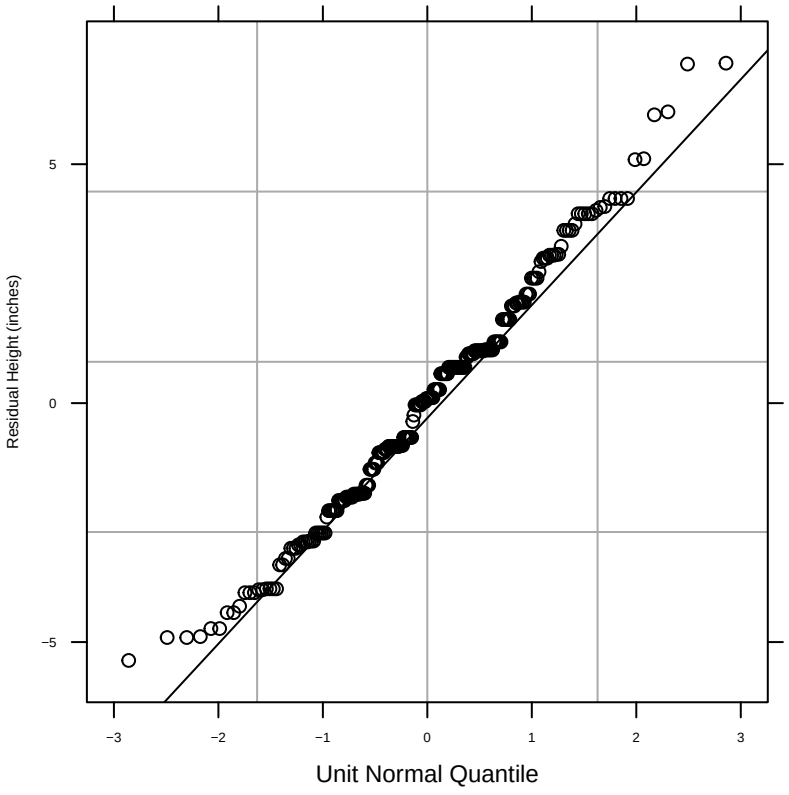
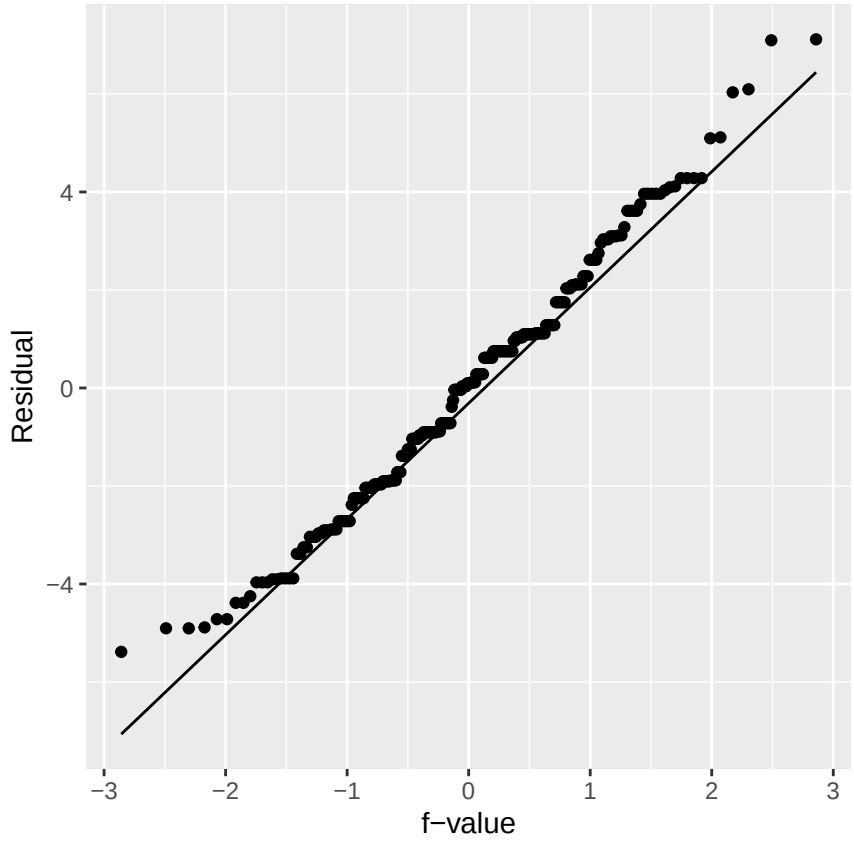


Figure 2.16

```
ggVD_2.16 <- function(){
    data <- singer
    data$res.height <- oneway(height~voice.part,
                              spread = 1, data=singer)$residuals
    ggplot(data, aes(sample = res.height)) +
    stat_qq(distribution = qnorm) +
    geom_qq_line()+labs(x = "f-value", y = "Residual")+
        theme(aspect.ratio=1)
}

ggVD_2.16()
```



9.3.5 The rfs plot

```
book.2.17 <- function(){
    rfs(oneway(height~voice.part, data = singer, spread = 1),
        aspect=1,
        sub = list("Figure 2.17",cex=.8),
        ylab = "Height (inches)")
}
book.2.17()
```

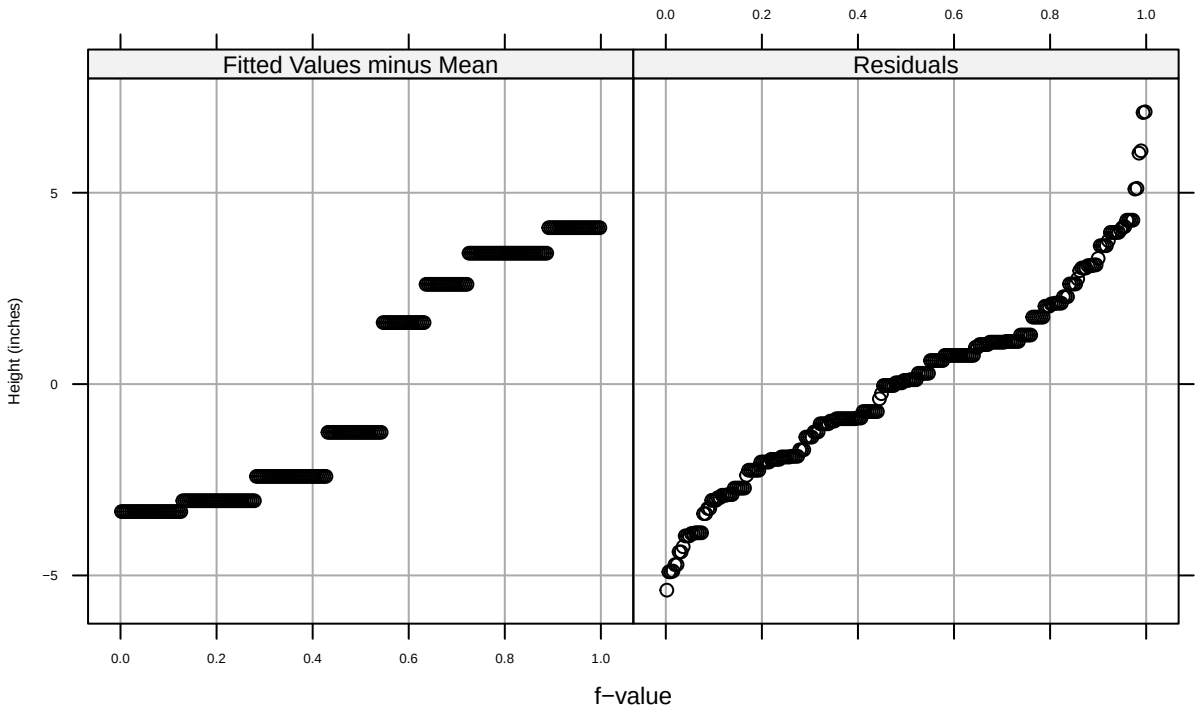
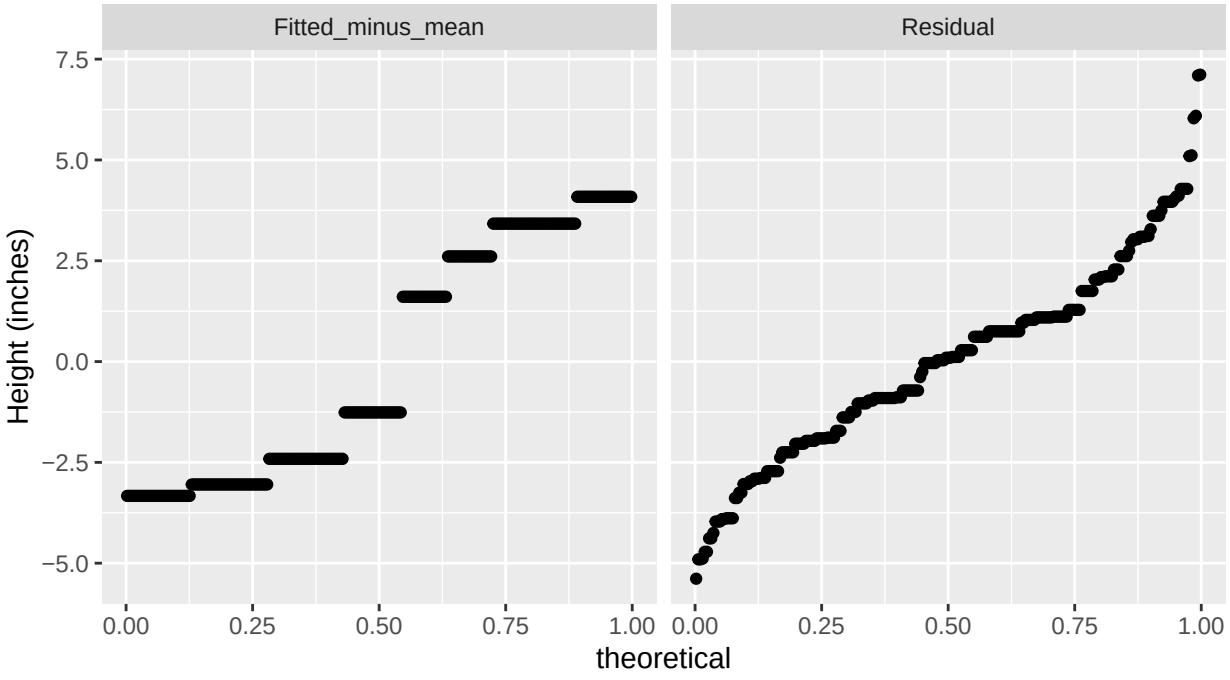


Figure 2.17

```
ggVD_2.17 <- function(){
    Fitoneway <- oneway(height~voice.part, data = singer, spread = 1)
    fitmean <- Fitoneway$fitted-mean(Fitoneway$fitted)
    singer.rfs <- data.frame(Fitted_minus_mean=fitmean,
                             Residual=Fitoneway$residuals)
    singer.m <-reshape2::melt(singer.rfs)
    ggplot(singer.m)+stat_qq(aes(sample = value),
                             distribution = qunif) +
        facet_wrap(.~variable) +
        labs(y = "Height (inches)")+
        theme(aspect.ratio=1)
}

ggVD_2.17()
```

```
## No id variables; using all as measure variables
```



9.3.6 Bivariate data

- Displaying functional relationship between two variables
  - Scatter plot Figure 1.3
  - Smoothing

9.3.7 Trivariate and hypervariate data

- Exploring interactions and conditional relationships
  - Scatter-plot matrix Figures 1.4 & 1.5
  - Conditional plots

9.3.8 Multiway data

Figures 1.1 & 1.6

- Multiway dotplot
  - The barley data
  - The lizard data

9.4 Univariate Data – Displaying and Comparing Distributions

9.4.1 Displaying distribution

Figure 2.1 - Quantile plot – Figure 2.2 Figure 2.3 versus Figure 2.4 - Boxplot Figure 2.6 versus Figure 2.7 - Histogram

9.4.2 Comparing distributions

- Boxplot Figure 2.8
- Normal quantile-quantile (Q-Q) plot Figures 2.10-11
- Q-Q plot Figure 2.9
- Multiplicative versus additive shift Figures 2.19-33

9.4.3 An Example of Graphical Exploratory Data Analysis

The foodweb for the animal species in an ecosystem is a description of who eats whom. A chain is a path through the web. It begins with a species that is eaten by no other, moves to a species that the first species eats, moves next to a species that the second species eats, and so forth until the chain ends at a species that preys on no other. If there are 7 species in the chain then there are 6 links between species, and the length of the chain is 6. The mean chain length of a web is the mean of the lengths of all chains in the web.

Ecosystems can be categorized based on their spatial dimensions: two-, three-, or mixed-dimensional ecosystems. The following plots analyze the mean chain length data from 113 ecosystems.

```
library(foreign)
library(lattice)
library(survival)  # using this library for date object
data.restore (paste(dataDIR, "visualizing.data", sep="/"))
```

```
## [1] "C:/Users/emman/OneDrive - University of Toledo/Documents/UToledo (Masters)/Courses/Fall 2021/En
```

```
## quantile plot
book.2.34 <-
function()
    qqmath(~ mean.length | dimension,
        distribution = qunif,
        data=food.web,
            panel = function(x, ...) {
            panel.grid()
                    panel.qqmath(x, ...)
            },
        layout=c(1,3),
        aspect=1,
        sub = list("Figure 2.34",cex=.8),
        xlab = "f-value",
        ylab="Chain Length")

book.2.35 <-
function()
{
    foo.m <- oneway(mean.length~dimension, data = food.web, location = median, spread=1)
```

```
    set.seed(19)
    xyplot(sqrt(abs(residuals(foo.m))) ~ jitter(fitted.values(foo.m),factor=0.3),
        aspect=1,
            panel = substitute(function(x,y){
            panel.xyplot(x,y)
            srmads <- tapply(abs(residuals(foo.m)),
                food.web$dimension, median)
            panel.lines(foo.m$location,srmads)
        }),
        sub = list("Figure 2.35",cex=.8),
        xlab="Jittered Median Chain Length",
        ylab="Square Root Absolute Residual Chain Length")
}


book.2.36 <-
function()
    qqmath(~ mean.length | dimension,
        data=food.web,
        prepanel = prepanel.qqmathline,
            panel = function(x, ...) {
            panel.grid()
                    panel.qqmathline(x, distribution = qnorm)
                    panel.qqmath(x, ...)
            },
        layout=c(1,3),
        aspect=1,
        sub = list("Figure 2.36",cex=.8),
        xlab = "Unit Normal Quantile",
        ylab="Chain Length")
book.2.37 <-
function()
{
    foo.m <- oneway(log(mean.length, 2) ~ dimension, data = food.web, location = median, spread = 1)
    set.seed(19)
    xyplot(sqrt(abs(residuals(foo.m))) ~ jitter(fitted.values(foo.m), factor = 0.3),
            panel = substitute(function(x, y) {
            panel.xyplot(x, y)
            add.line <- trellis.par.get("add.line")
            panel.lines(foo.m$location, tapply(y, food.web$dimension, median),
                lty = add.line$lty, lwd = add.line$lwd, col = add.line$col)
        }),
        aspect = 1,
        sub = list("Figure 2.37",cex=.8),
        xlab = "Jittered Median Log 2 Chain Length",
        ylab = "Square Root Absolute Residual Log 2 Chain Length")
}

book.2.38 <-
function()
    qqmath(~ log(mean.length,2) | dimension,
        data=food.web,
        prepanel = prepanel.qqmathline,
```

```
            panel = function(x, ...) {
            panel.grid()
                    panel.qqmathline(x, distribution = qnorm)
                    panel.qqmath(x, ...)
            },
        layout=c(1,3),
        aspect=1,
        sub = list("Figure 2.38",cex=.8),
        xlab = "Unit Normal Quantile",
        ylab="Log 2 Chain Length")
book.2.39 <-
function()
{
    foo.m <- oneway(1/mean.length ~ dimension, data = food.web, location = median, spread = 1)
    set.seed(19)
    xyplot(sqrt(abs(residuals(foo.m))) ~ jitter(fitted.values(foo.m), factor = 0.3),
            panel = substitute(function(x,y) {
            panel.xyplot(x,y)
                    add.line <- trellis.par.get("add.line")
            panel.lines(foo.m$location, tapply(y, food.web$dimension, median),
            lty = add.line$lty, lwd = add.line$lwd, col = add.line$col)
        }),
        aspect = 1,
        sub = list("Figure 2.39",cex=.8),
        xlab = "Jittered Median Link Fraction",
        ylab = "Square Root Absolute Residual Link Fraction")
}
book.2.4 <-
function()
{
    voice.part <- ordered(singer$voice.part,
        c("Soprano 1", "Soprano 2", "Alto 1", "Alto 2",
            "Tenor 1", "Tenor 2", "Bass 1", "Bass 2"))
    bass.tenor.qq <- qq(voice.part ~ singer$height,
        subset=voice.part=="Bass 2" | voice.part=="Tenor 1")
    tmd(bass.tenor.qq,
        aspect=1,
        ylab = "Difference (inches)",
        sub = list("Figure 2.4",cex=.8),
        xlab = "Mean (inches)")
}
book.2.40 <-
function()
    qqmath(~ (1/mean.length) | dimension,
           data = food.web,
           panel = function(x, ...){
            panel.grid()
            panel.qqmath(x, ...)
            panel.qqmathline(x, distribution = qnorm)
           },
           layout = c(1, 3),
           aspect = 1,
        sub = list("Figure 2.40",cex=.8),
```

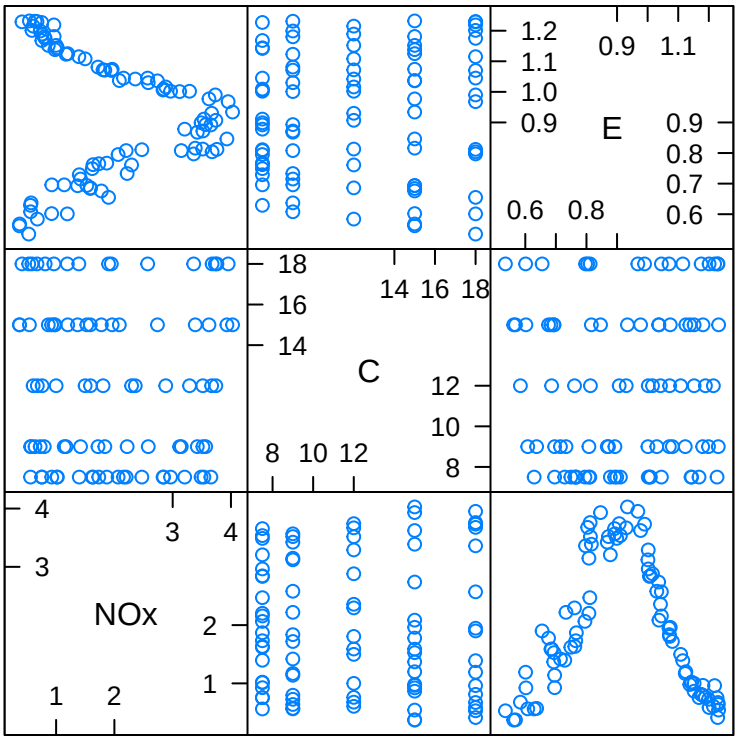
```
           xlab = "Unit Normal Quantile",
           ylab = "Link Fraction")
book.2.41 <-
function()
{
    res <- oneway((1/mean.length)~dimension, data = food.web, spread = 1)$residuals
    qqmath(~ res | food.web$dimension,
        distribution = substitute(function(p) quantile(res, p)),
        panel=function(x,...){
            panel.grid()
            panel.abline(0, 1)
            panel.qqmath(x, ...)
        },
        layout=c(1,3),
        aspect=1,
        sub = list("Figure 2.41",cex=.8),
        xlab = "Pooled Residual Link Fraction",
        ylab = "Residual Link Fraction")
}

book.2.42 <-
function()
    rfs(oneway((1/mean.length)~dimension, data = food.web, spread = 1),
        sub = list("Figure 2.42",cex=.8),
        aspect=1,
        ylab = "Link Fraction")
```

10 Week 12: Conditioning

10.1 The Ethanol Data

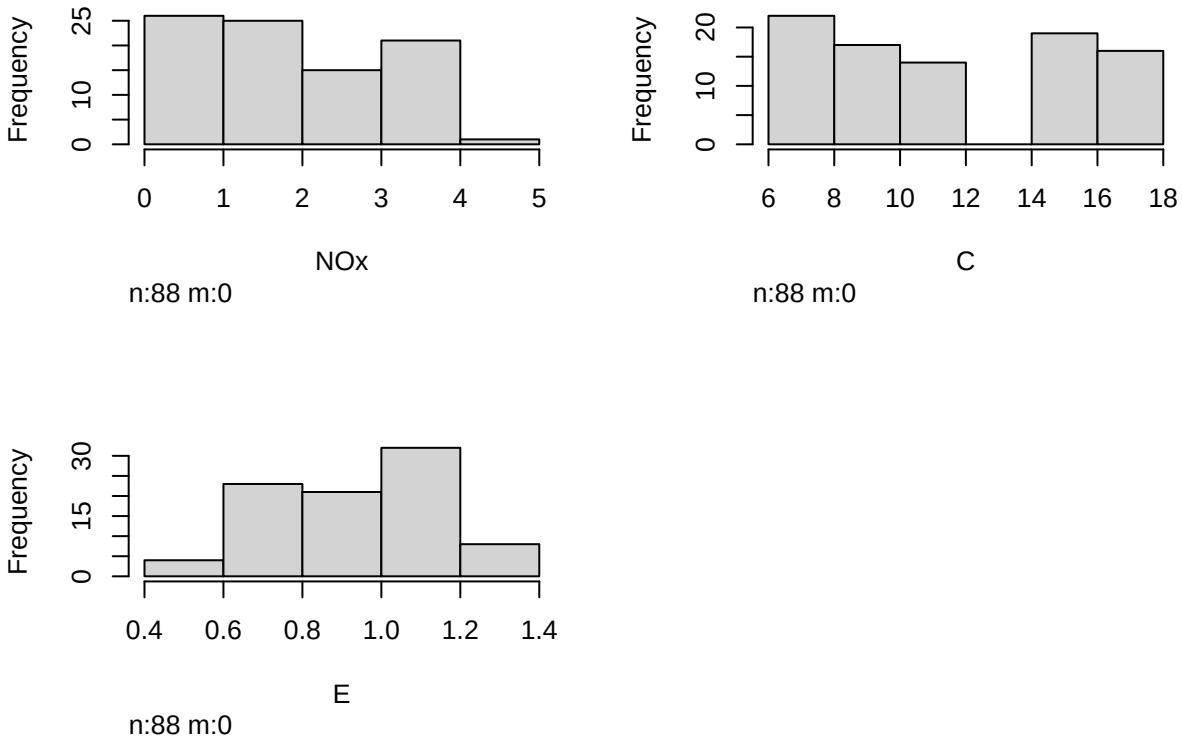
Brinkman (1981) published a dataset documenting the NOx (NO and NO2) emission from a single-cylinder engine using ethanol as fuel. The experiment imeasured the NOx emission under different combinations of two operation conditions: compression ratio of the engine (C, the volume inside the cylinder when the piston is retracted divided by the volume when the piston is at its maximum point of penetration) and the equivalence ratio (E, a measure of the richness of the air and ethanol fuel mixture). Cleveland used this dataset in his book **Visualizing Data** to demonstrate the conditional plots.

```
## using lattice
eth <- ethanol
splom(eth)
```



Scatter Plot Matrix

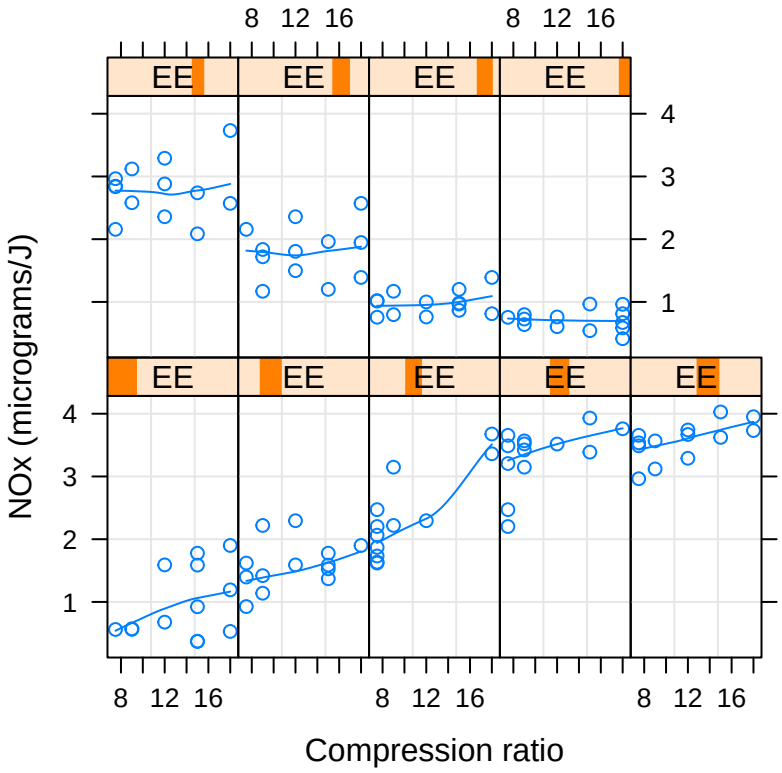
```
hist(eth)
```




```
## using gglopt2 (actually GGally)
#ggpairs(ethanol, upper="blank")
```

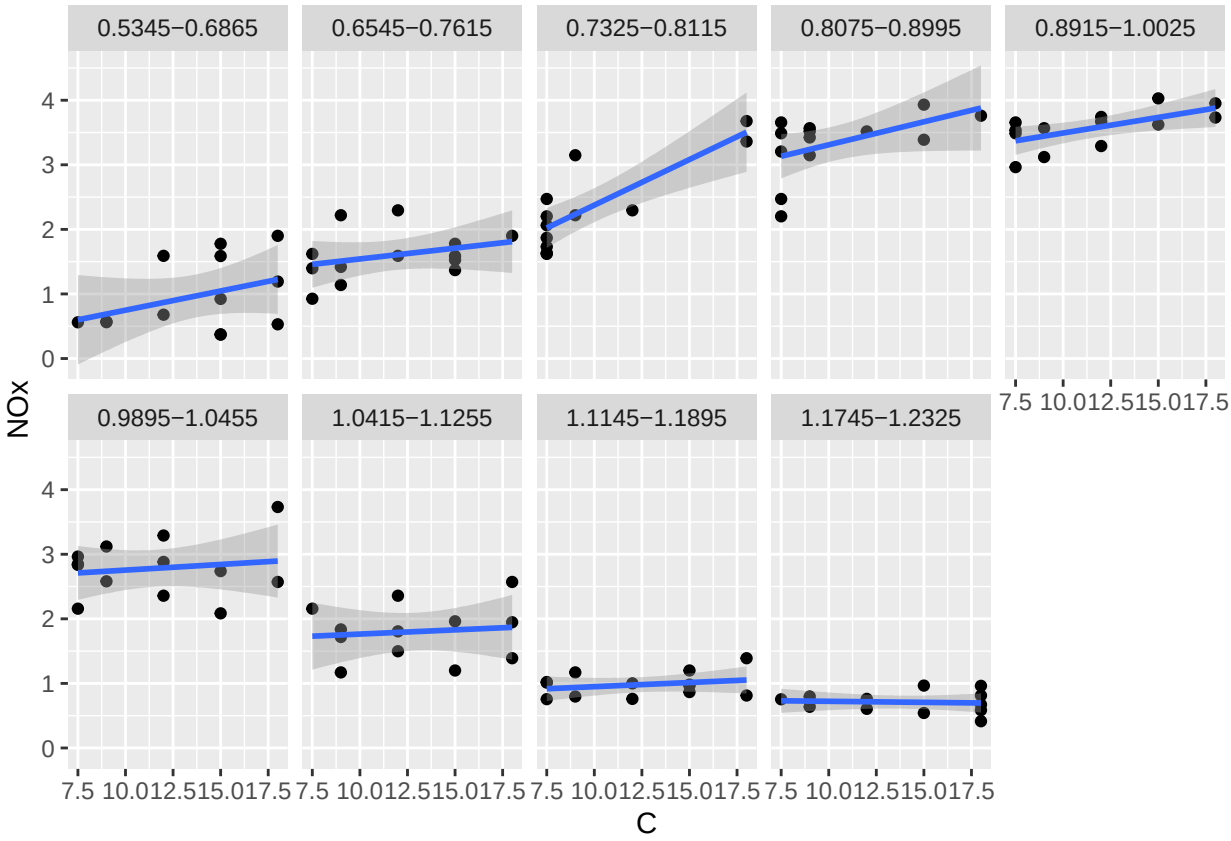
There are 88 data points. From the scatter plot matrix, we are tempted to infer that NOx emission is largely determined by the equivalence ratio. The engine compress ratio is not a factor. However, the scatter plot of NOx against C does not take E into consideration and the scatter plot of NOx against E does not take C into consideration. An important concept in data analysis of more than three dimensions is interaction – the effect of C on NOx may vary when C changes. In environmental sciences, we find that interaction is common. Furthermore, the strong effect of E can mask the effect of C. Cleveland showed a conditional plot (co-plot):

```
## Using lattice
eth <- ethanol
eth$EE <- equal.count(ethanol$E, number=9, overlap=1/4)
xyplot(NOx ~ C | EE, data = eth,
       xlab = "Compression ratio", ylab = "NOx (micrograms/J)",
       panel = function(x, y) {
           panel.grid(h=-1, v= 2)
           panel.xyplot(x, y)
           panel.loess(x,y, span=1)
       },
       layout=c(5,2),
       aspect = 2)
```



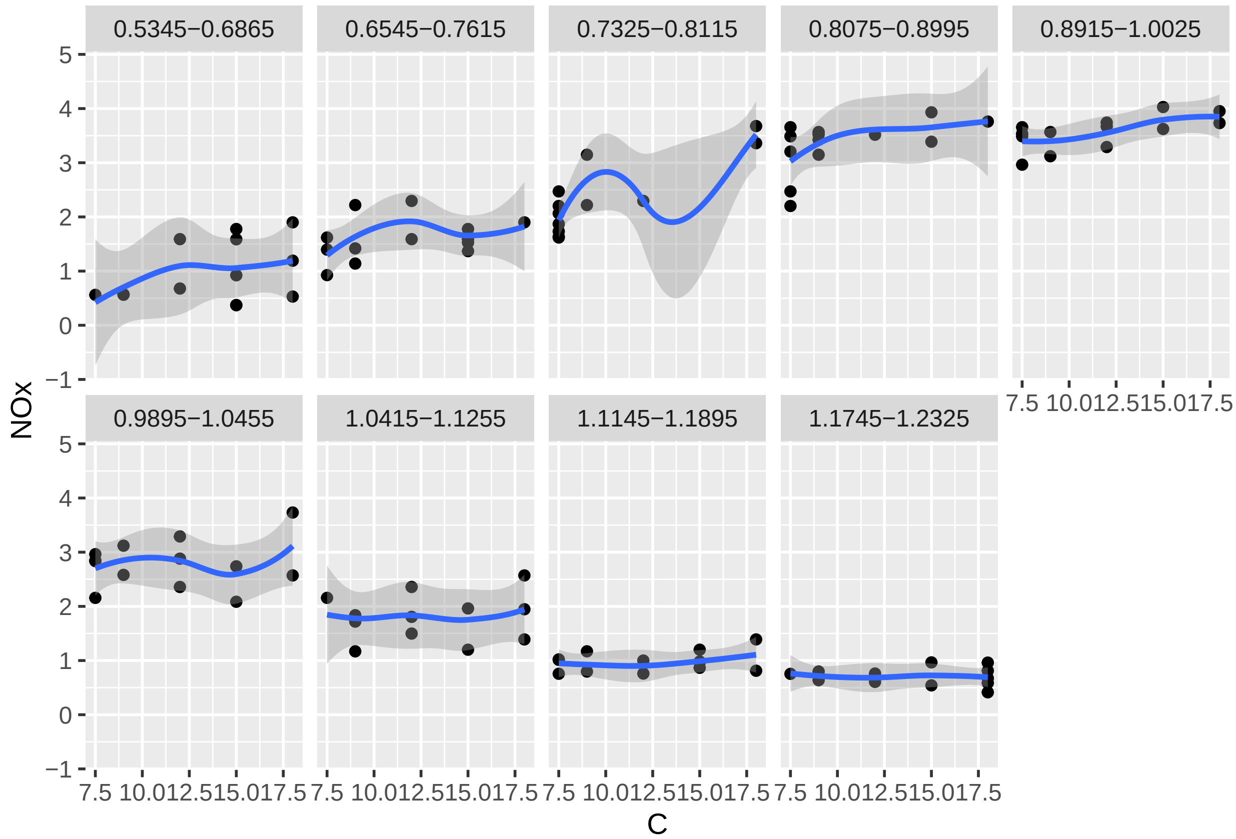

```
## using ggplot2
eth <- ethanol
fn <- function(data = eth$E, number = 4, ...) {
     intrv <<- as.data.frame(co.intervals(data, number,
         ...))
     mg_y <- sort(unique(data))
     intervals <- plyr::ldply(mg_y, function(x) {
         t(as.numeric(x < intrv$V2 & x > intrv$V1))
     })
     tmp <- reshape2::melt(cbind(mg_y, intervals), id.var = 1)
     tmp[tmp$value > 0, 1:2]
 }
 eth.ordered <- merge(eth, fn(number = 9, overlap = 0.25), by.x="E", by.y="mg_y")

intrv <- with(intrv, paste(V1, V2, sep = "-"))

 eth.ordered <- rename(eth.ordered, c(variable = "EE"))
 eth.ordered$EE <- factor(eth.ordered$EE,
     labels = intrv)

p <- ggplot(eth.ordered, aes(x=C, y=NOx)) +
     geom_point() + facet_wrap(~EE, nrow = 2)
 print(p+geom_smooth(method=lm))
```

```
## `geom_smooth()` using formula 'y ~ x'
```




```
print(p+geom_smooth(method="loess", span=1))
```

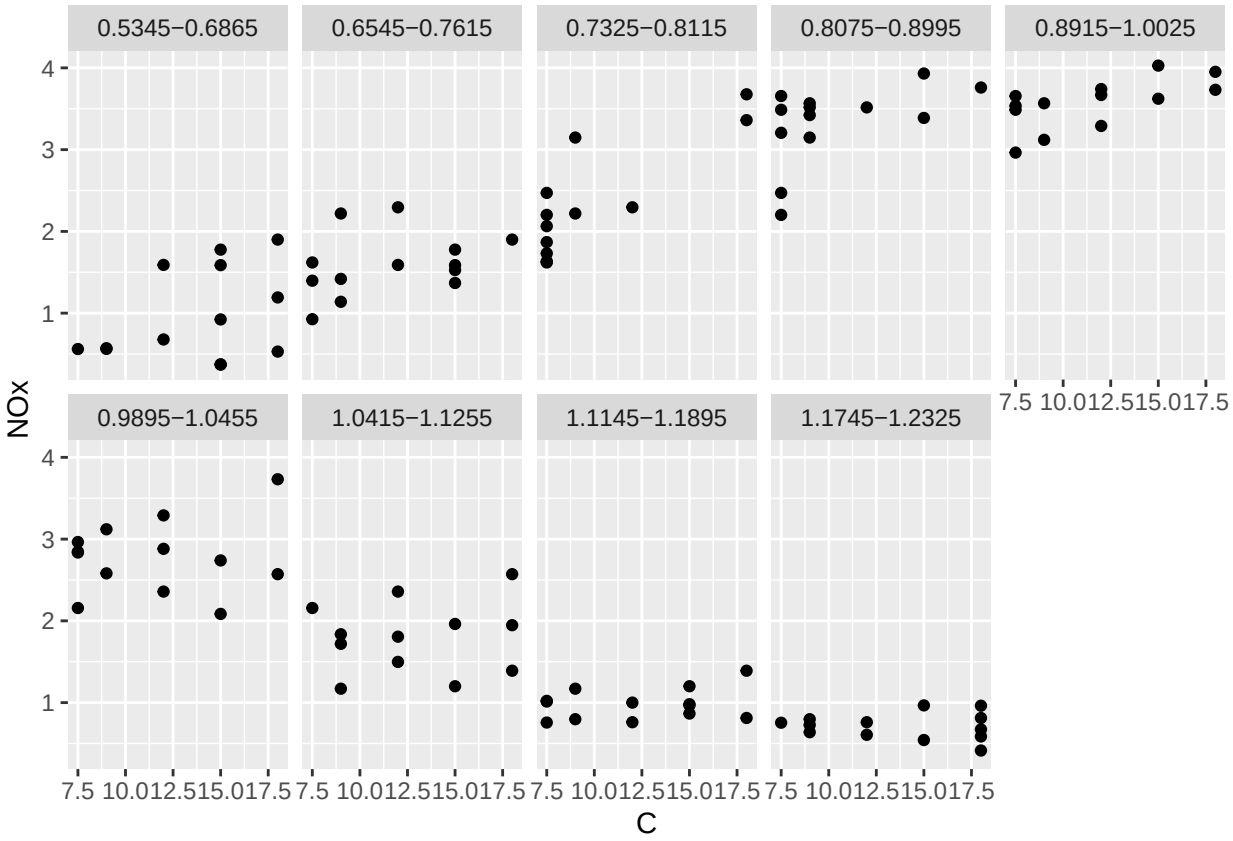
```
## `geom_smooth()` using formula 'y ~ x'
```




```
print(p+geom_smooth(method="gam"))
```

```
## `geom_smooth()` using formula 'y ~ s(x, bs = "cs")'

## Warning: Computation failed in `stat_smooth()`:
## x has insufficient unique values to support 10 knots: reduce k.

## Warning: Computation failed in `stat_smooth()`:
## x has insufficient unique values to support 10 knots: reduce k.

## Warning: Computation failed in `stat_smooth()`:
## x has insufficient unique values to support 10 knots: reduce k.

## Warning: Computation failed in `stat_smooth()`:
## x has insufficient unique values to support 10 knots: reduce k.

## Warning: Computation failed in `stat_smooth()`:
## x has insufficient unique values to support 10 knots: reduce k.

## Warning: Computation failed in `stat_smooth()`:
## x has insufficient unique values to support 10 knots: reduce k.

## Warning: Computation failed in `stat_smooth()`:
## x has insufficient unique values to support 10 knots: reduce k.
```

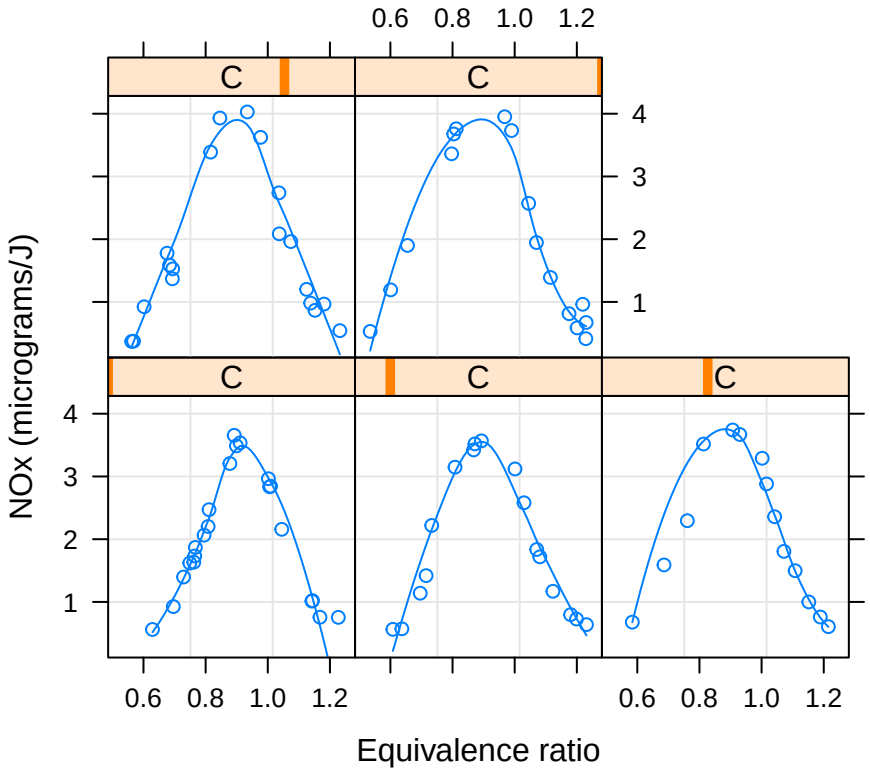
```
## Warning: Computation failed in `stat_smooth()`:
## x has insufficient unique values to support 10 knots: reduce k.

## Warning: Computation failed in `stat_smooth()`:
## x has insufficient unique values to support 10 knots: reduce k.
```



This coplot is a series of scatter plot of NOx against C. Each plot uses a subset of the data selected based on E. These intervals are selected using the equal count method with overlap of 1/4. The coplot shows that NOx does depend on C; for low values of E, NOx increases with C, and for medium and high values of E, NOx is constant as a function of C. The underlying pattern seems to be linear.

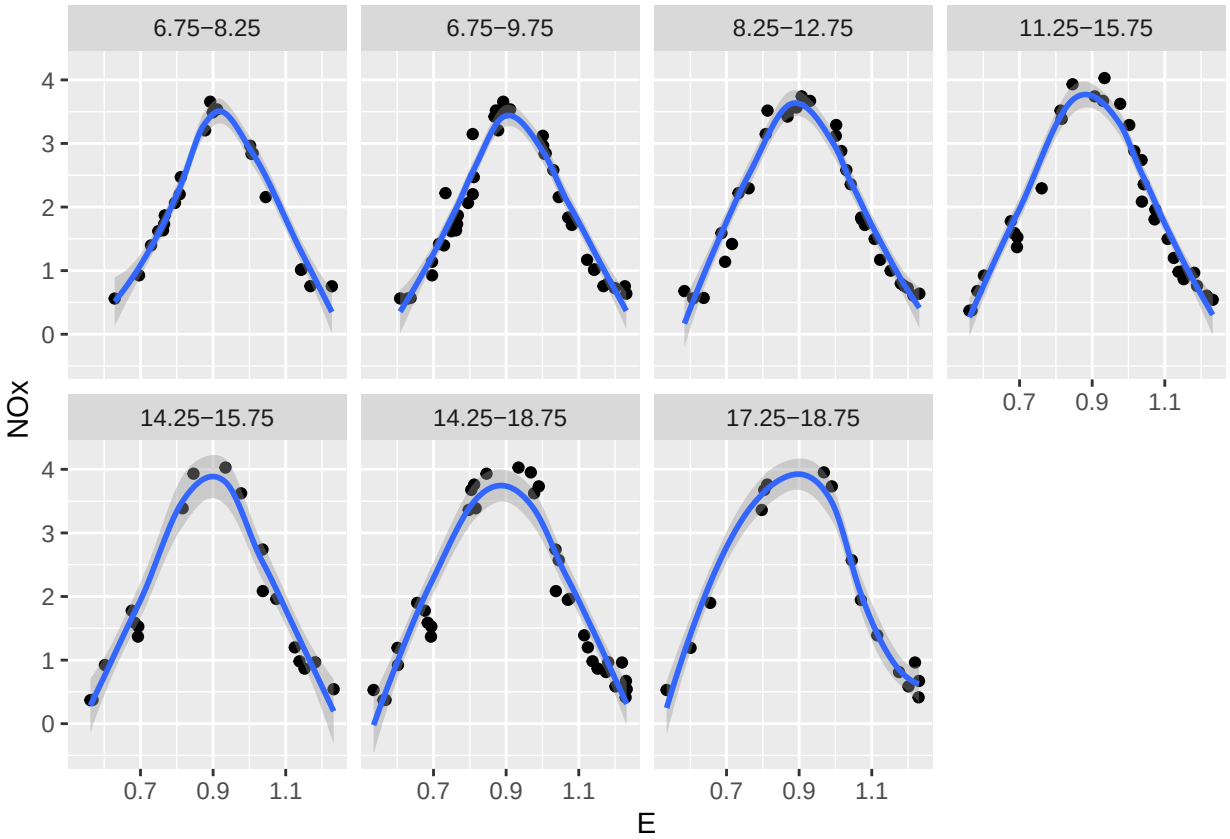
We can also look at the three-D relationship from the other angle:

```
## Using lattice
xyplot(NOx ~ E | C, data = ethanol,
       prepanel = function(x, y) prepanel.loess(x, y, span = 1),
       xlab = "Equivalence ratio", ylab = "NOx (micrograms/J)",
       panel = function(x, y) {
           panel.grid(h=-1, v= 2)
           panel.xyplot(x, y)
           panel.loess(x,y, span=0.75, degree=2)
       },
       aspect = "xy")
```




```
## Using ggplot2
eth <- ethanol
eth.ordered <- merge(eth, fn(data=eth$C, number = 9, overlap = 0.25), by.x="C", by.y="mg_y")

intrv <- with(intrv, paste(V1, V2, sep = "-"))

 eth.ordered <- rename(eth.ordered, c(variable = "CC"))
 eth.ordered$CC <- factor(eth.ordered$CC,
     labels = intrv)

p <- ggplot(eth.ordered, aes(x=E, y=NOx)) +
     geom_point() + facet_wrap(~CC, nrow = 2)
 print(p+geom_smooth(method="loess", span=0.75))
```

```
## `geom_smooth()` using formula 'y ~ x'
```



The coplot shows that the peak concentration of NOx occurs near E=0.9 for all five values of C. But the value of NOx at the peak increases slightly as C increases. These two coplots show that the effect of C on NOx depends on the value of E, and vice versa, so there is an interaction between the factors.
This example shows the value of coplot – it is an easy to use tool for exploring high-dimensional data. Using coplot, we can explore the data easily. The lattice package provides a great deal of flexibility to facilitate this exploration. For example, we can rearrance the layout of the panels to better illustrate the point:

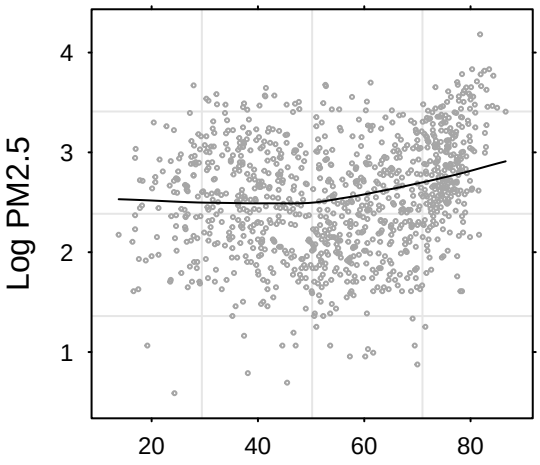
```
xyplot(NOx ~ C | E, data = ethanol,
       prepanel = function(x, y) prepanel.loess(x, y, span = 1),
       xlab = "Compression ratio", ylab = "NOx (micrograms/J)",
       panel = function(x, y) {
           panel.grid(h=-1, v= 2)
           panel.xyplot(x, y)
           panel.loess(x,y, span=1)
       }, layout=c(5,2),
       aspect = 2)

xyplot(NOx ~ E | C, data = ethanol,
       prepanel = function(x, y) prepanel.loess(x, y, span = 1),
       xlab = "Equivalence ratio", ylab = "NOx (micrograms/J)",
       panel = function(x, y) {
           panel.grid(h=-1, v= 2)
           panel.xyplot(x, y)
           panel.loess(x,y, span=0.75, degree=2)
       }, layout=c(3,2),
       aspect = "xy")
```

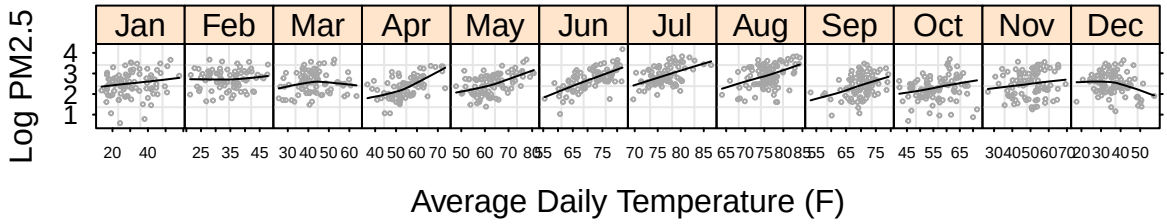
The story of the NOx data was the result of several iterations of selecting the number of intervals and the appropriate layout. The goals are (1) to better understand the structure of the data and (2) to better present the structure.

We have seen several examples already:

The PM2.5 data from Baltimore

```
pmdata<-read.table(paste (dataDIR, "PM-RAW-DATA.txt", sep="/"),header=TRUE)
pmdata$rain<-pmdata$Precip > 0
pmdata$log.wind<-log(pmdata$AvgWind)
pmdata$z.log.wind<-as.vector(scale(pmdata$log.wind))
pmdata$z.temp<-as.vector(scale(pmdata$AvgTemp))
pmdata$log.value<-log(pmdata$value)
## dates were recorded as days since Jan. 1, 2003
pmdata$Dates <-  as.Date("2003-01-01") + pmdata$date-1
pmdata$Weekday <- weekdays(pmdata$Dates, abbreviate=T)
pmdata$Month <- ordered(months(pmdata$Dates, T), levels=month.abb)

## using lattice
obj1<-xyplot(log.value~AvgTemp, panel=function(x,y,...){
  panel.grid()
  panel.xyplot(x,y, col=grey(0.65), cex=0.25, ...)
  panel.loess(x,y,span=1, degree=1,col=1,...)
},    scales=list(x=list(cex=0.75, tck=0.2), y=list(cex=0.75, tck=0.2)),
##               par.settings=trellis.par.temp,
       data=pmdata, xlab="", ylab="Log PM2.5")


obj2 <- xyplot(log.value~AvgTemp|Month, panel=function(x,y,...){
  panel.grid()
  panel.xyplot(x,y, col=grey(0.65), cex=0.25, ...)
  panel.loess(x,y,span=1, degree=1,col=1,...)
  }, layout=c(12, 1),
       scales=list(y=list(tck=0.2),
         x=list(relation="free", cex=0.5,tck=c(0.2,0),
           alternating=c(1,2))),
       ## x-axis relation and font size
  ##par.settings=trellis.par.temp,
  data=pmdata, xlab="Average Daily Temperature (F)", ylab="Log PM2.5")
print(obj1, position=c(1/4, 0.3, 3/4, 1), more=T)
print(obj2, position=c(0, 0, 1,0.35), more=F)
```

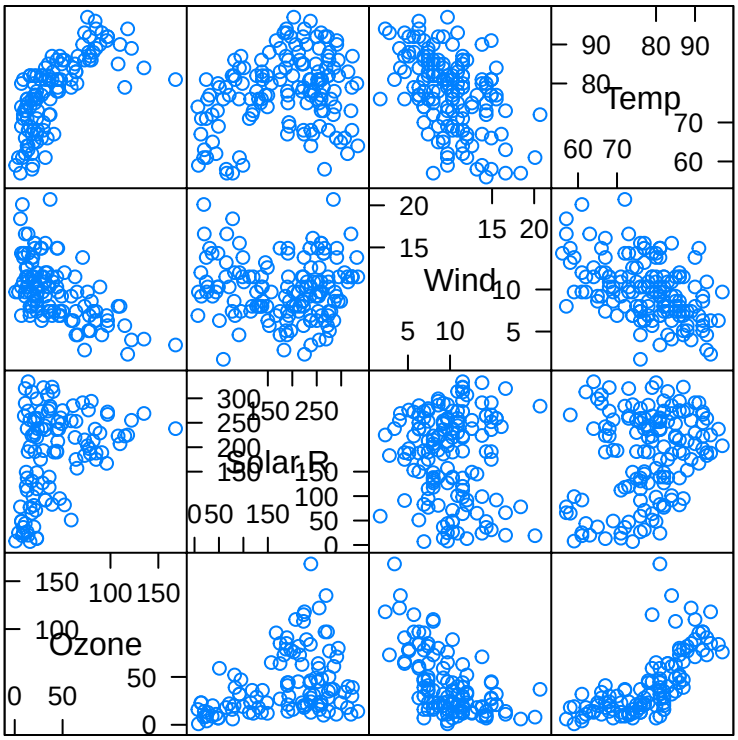





```
## using ggplot2
p1 <- ggplot(pmdata, aes(x=AvgTemp, y=log.value))
obj1 <- p1 + geom_point(size=0.25) + geom_smooth(method="loess", span=1) + theme(aspect.ratio=0.75) + xl
obj2 <- p1 + geom_point(size=0.25) + facet_grid(.~Month, scales="free_x") + geom_smooth(method="loess",

#grid.arrange(obj1, obj2, ncol=1)
```

The Airquality data from R:

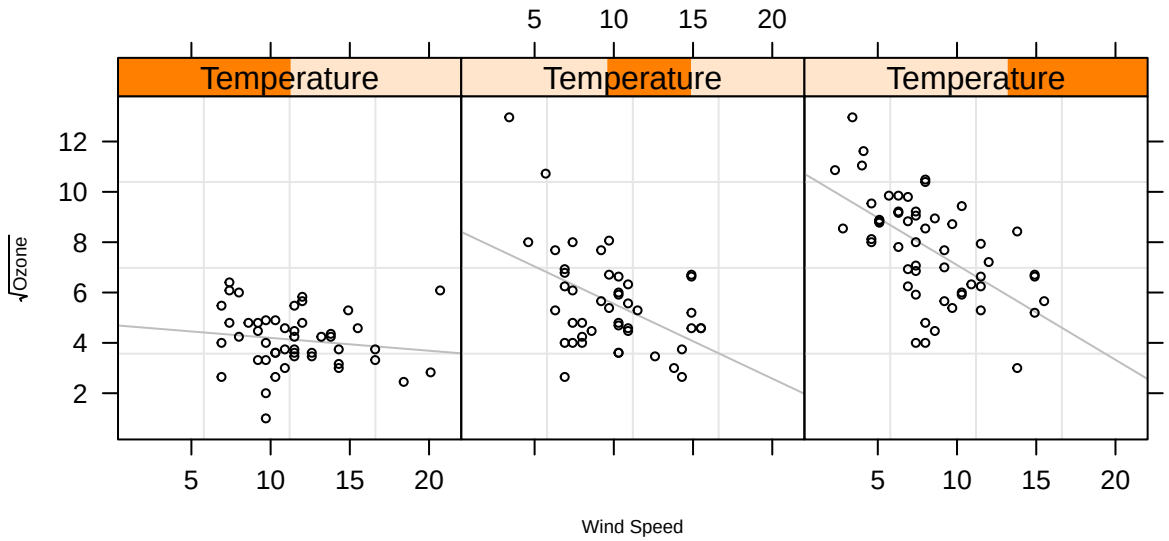
```
data(airquality)
splom(airquality[,1:4])
```



Scatter Plot Matrix

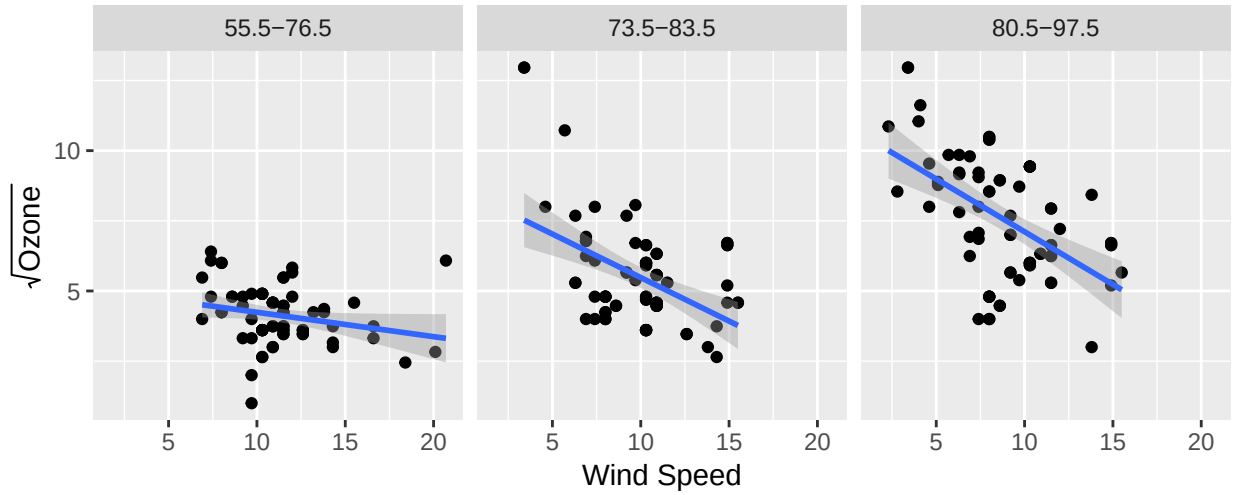
```
## compared to:
## pairs(Ozone~Solar.R+Wind+Temp, data=airquality)

Temperature <- equal.count(airquality$Temp, 3, 0.25)
Wind_Speed <- equal.count(airquality$Wind, 3, 0.25)
Solar_R <- equal.count(airquality$Solar.R, 3, 0.25)

airQ <- airquality

airQ <- merge(airQ, fn(data=airQ$Temp, number = 3, overlap = 0.25), by.x="Temp", by.y="mg_y")
intrv <- with(intrv, paste(V1, V2, sep = "-"))
airQ <- rename(airQ, c(variable = "Temperature"))
airQ$Temperature <- factor(airQ$Temperature, labels = intrv)

airQ <- merge(airQ, fn(data=airQ$Wind, number = 3, overlap = 0.25), by.x="Wind", by.y="mg_y")
intrv <- with(intrv, paste(V1, V2, sep = "-"))
airQ <- rename(airQ, c(variable = "Wind_Speed"))
airQ$Wind_Speed <- factor(airQ$Wind_Speed, labels = intrv)

airQ <- merge(airQ, fn(data=airQ$Solar.R, number = 3, overlap = 0.25), by.x="Solar.R", by.y="mg_y")
intrv <- with(intrv, paste(V1, V2, sep = "-"))
airQ <- rename(airQ, c(variable = "Solar_Radiation"))
airQ$Solar_Radiation <- factor(airQ$Solar_Radiation, labels = intrv)


xyplot(sqrt(Ozone) ~ Wind|Temperature,
```

```
       data=airquality,
       panel=function(x,y,...){
#            panel.loess(x, y, span=1, degree=1, ...)
             panel.grid()
             panel.lmline(x, y, col="grey",...)
             panel.xyplot(x, y, col=1, cex=0.5, ...)
       },
       layout=c(3, 1), aspect=1,
       ylab=list(label=expression(sqrt("Ozone")), cex=0.6),
       xlab=list(label="Wind Speed", cex=0.6)
)
```

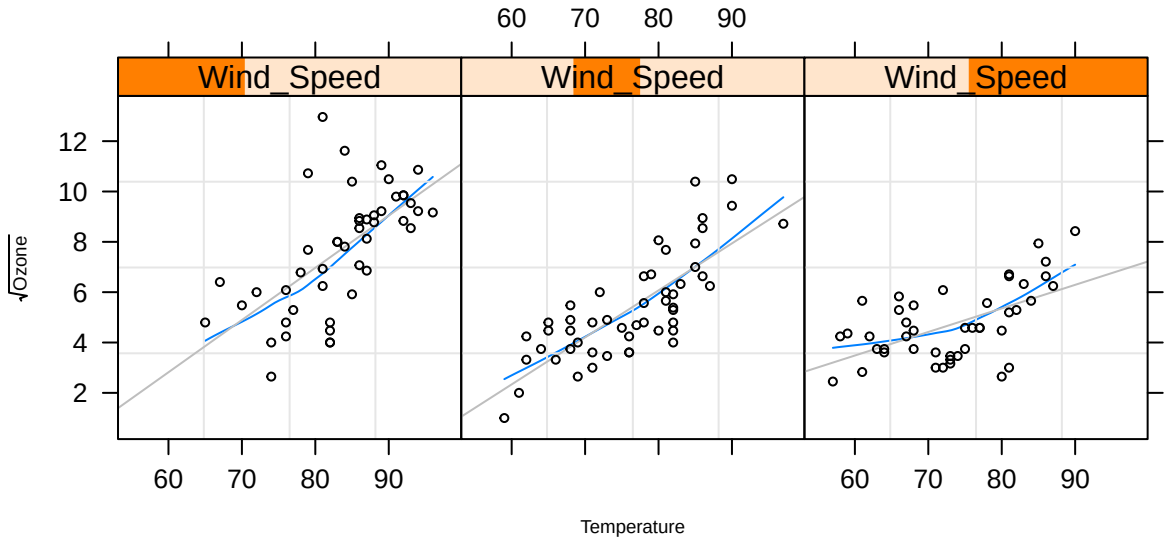



```
p <- ggplot(airQ, aes(x=Wind, y=sqrt(Ozone))) +
     geom_point() + facet_grid(.~Temperature)
 print(p+geom_smooth(method="lm") + scale_y_continuous(expression(sqrt("Ozone")))
       + scale_x_continuous(expression("Wind Speed"))+ theme(aspect.ratio=1))
```

```
## `geom_smooth()` using formula 'y ~ x'

## Warning: Removed 82 rows containing non-finite values (stat_smooth).

## Warning: Removed 82 rows containing missing values (geom_point).
```

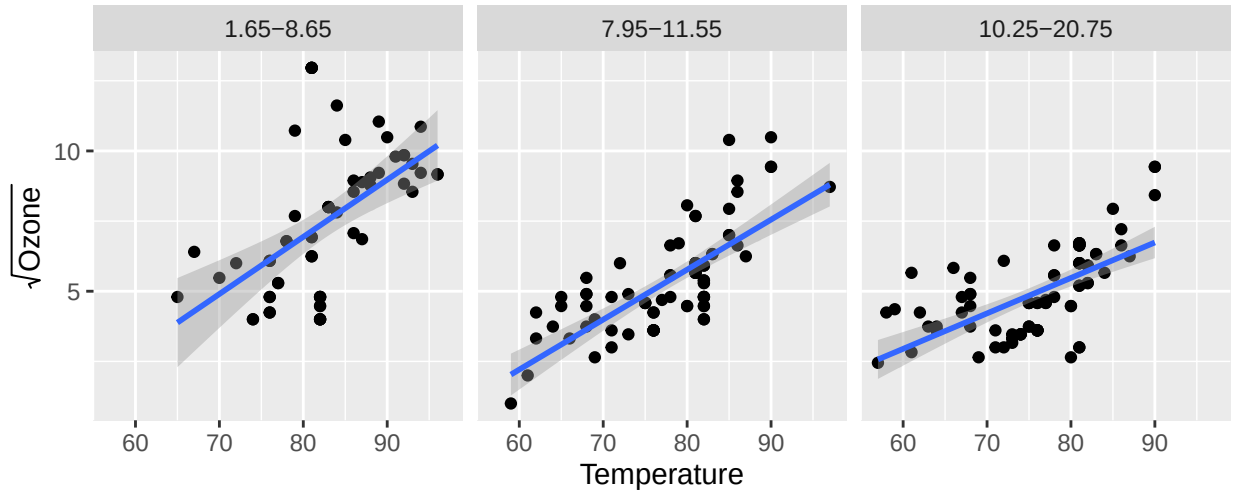



```
xyplot(sqrt(Ozone) ~ Temp|Wind_Speed,
       data=airquality,
       panel=function(x,y,...){
            panel.loess(x, y, span=1, degree=1, ...)
            panel.grid()
            panel.lmline(x, y, col="grey",...)
            panel.xyplot(x, y, col=1, cex=0.5, ...)
       },
       layout=c(3, 1), aspect=1,
       ylab=list(label=expression(sqrt("Ozone")), cex=0.6),
       xlab=list(label="Temperature", cex=0.6)
)
```

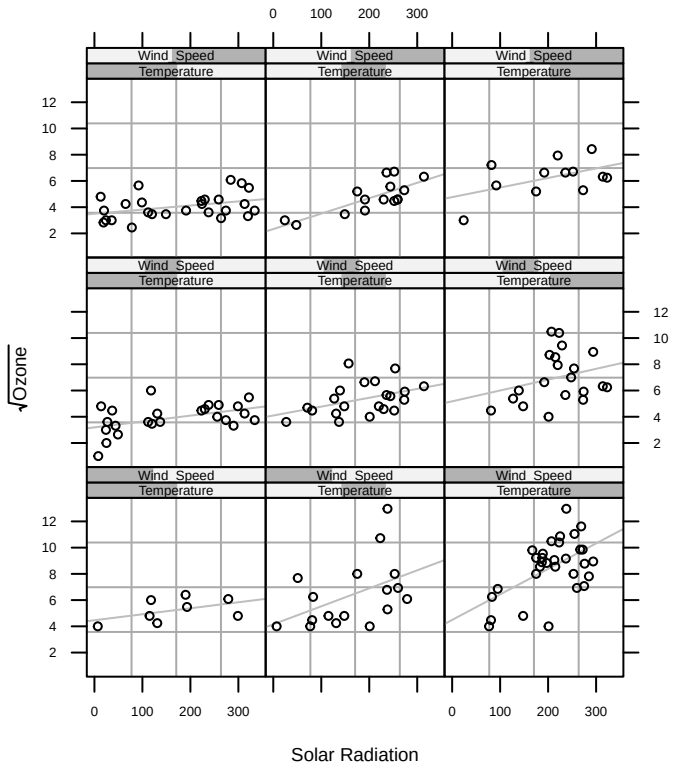



```
p <- ggplot(airQ, aes(x=Temp, y=sqrt(Ozone))) +
     geom_point() + facet_grid(.~Wind_Speed)
print(p+geom_smooth(method="lm") + scale_y_continuous(expression(sqrt("Ozone")))
       + scale_x_continuous(expression("Temperature"))+ theme(aspect.ratio=1))
```

```
## `geom_smooth()` using formula 'y ~ x'

## Warning: Removed 82 rows containing non-finite values (stat_smooth).

## Warning: Removed 82 rows containing missing values (geom_point).
```

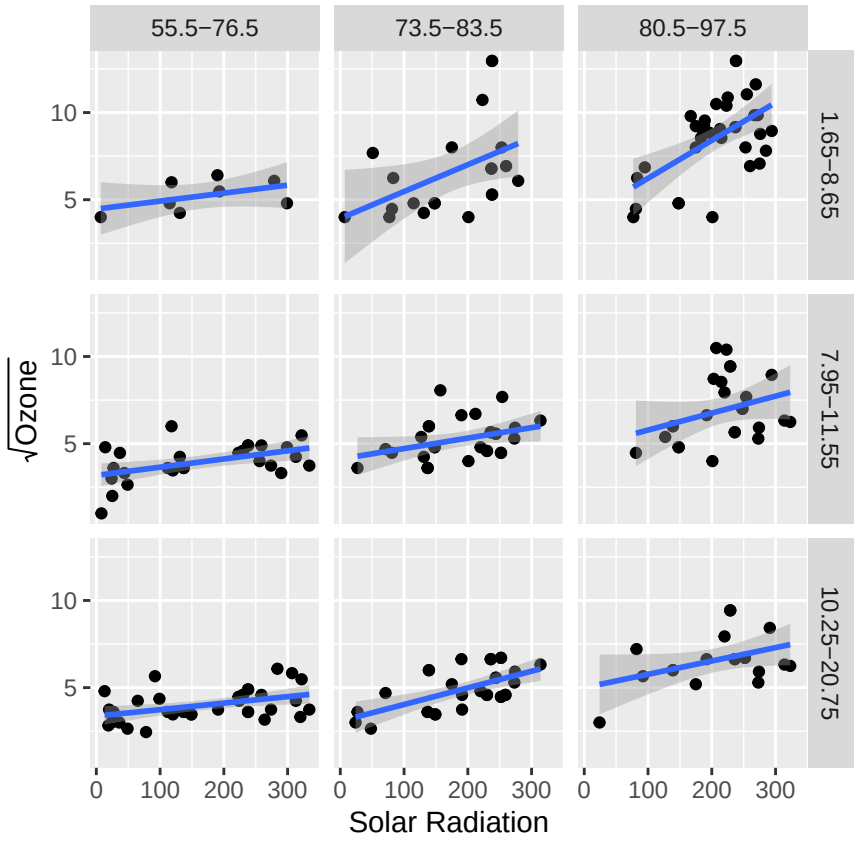



```
trellis.par.set(theme = canonical.theme("postscript", col=FALSE))
trellis.par.set(list(layout.widths=list(left.padding=0, right.padding=0,
                        ylab.axis.padding=0, axis.right=0, key.ylab.padding=0)))

xyplot(sqrt(Ozone) ~ Solar.R|Temperature*Wind_Speed,
       data=airquality,
       panel=function(x,y,...){
#             panel.loess(x, y, span=1, degree=1, ...)
            panel.grid()
            panel.lmline(x, y, col="grey",...)
            panel.xyplot(x, y, col=1, cex=0.5, ...)
       },
       aspect=1,
       ylab=list(label=expression(sqrt("Ozone")), cex=0.6),
       xlab=list(label="Solar Radiation", cex=0.6),
       scales=list(x=list(alternating=c(1, 2, 1))),
#       between=list(y=1),
       par.strip.text=list(cex=0.4),
       par.settings=list(axis.text=list(cex=0.4))
)
```

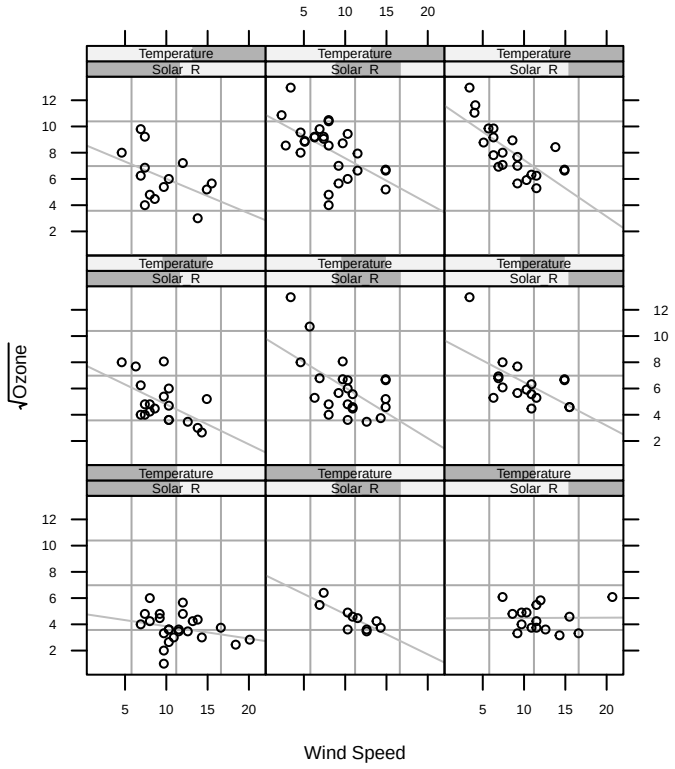



```
p <- ggplot(airQ, aes(x=Solar.R, y=sqrt(Ozone))) +
     geom_point() + facet_grid(Wind_Speed~Temperature)
print(p+geom_smooth(method="lm") + scale_y_continuous(expression(sqrt("Ozone")))
       + scale_x_continuous(expression("Solar Radiation"))+ theme(aspect.ratio=1))
```

```
## `geom_smooth()` using formula 'y ~ x'

## Warning: Removed 82 rows containing non-finite values (stat_smooth).

## Warning: Removed 82 rows containing missing values (geom_point).
```




```
xyplot(sqrt(Ozone) ~ Wind|Solar_R*Temperature,
             data=airquality,
             panel=function(x,y,...){
#            panel.loess(x, y, span=1, degree=1, ...)
               panel.grid()
               panel.lmline(x, y, col="grey",...)
               panel.xyplot(x, y, col=1, cex=0.5, ...)
             },
             aspect=1,
             ylab=list(label=expression(sqrt("Ozone")), cex=0.6),
             xlab=list(label="Wind Speed", cex=0.6),
             scales=list(x=list(alternating=c(1, 2, 1))),
                                    #       between=list(y=1),
             par.strip.text=list(cex=0.4),
             par.settings=list(axis.text=list(cex=0.4)))
```

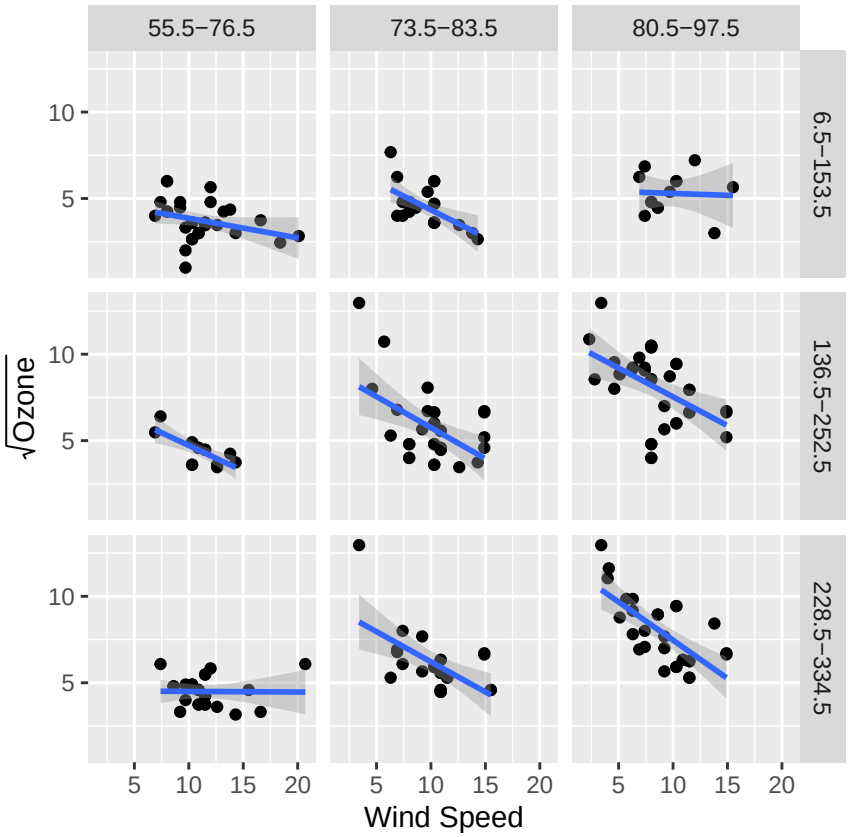



```
p <- ggplot(airQ, aes(x=Wind, y=sqrt(Ozone))) +
     geom_point() + facet_grid(Solar_Radiation~Temperature)
print(p+geom_smooth(method="lm") + scale_y_continuous(expression(sqrt("Ozone")))
       + scale_x_continuous(expression("Wind Speed"))+ theme(aspect.ratio=1))
```

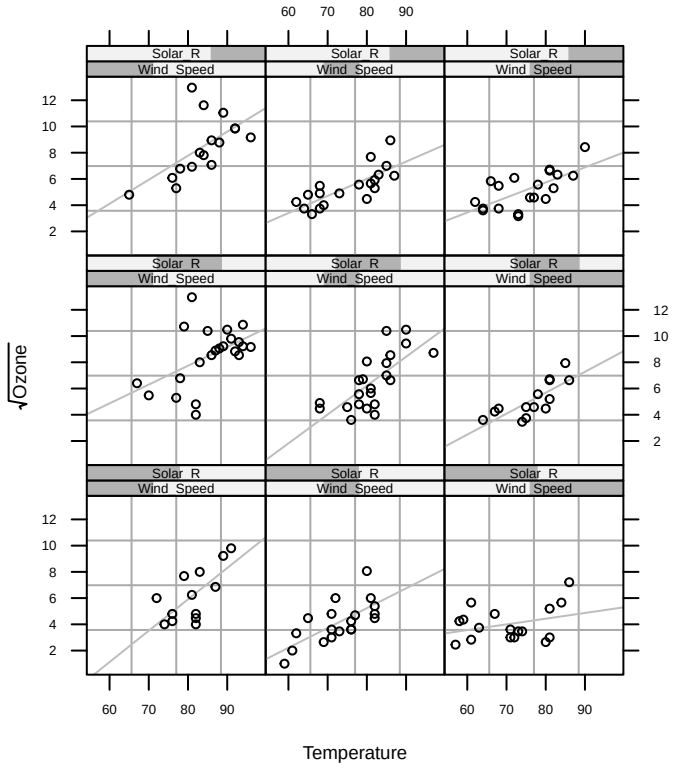
```
## `geom_smooth()` using formula 'y ~ x'
```

```
## Warning: Removed 82 rows containing non-finite values (stat_smooth).
```

```
## Warning: Removed 82 rows containing missing values (geom_point).
```

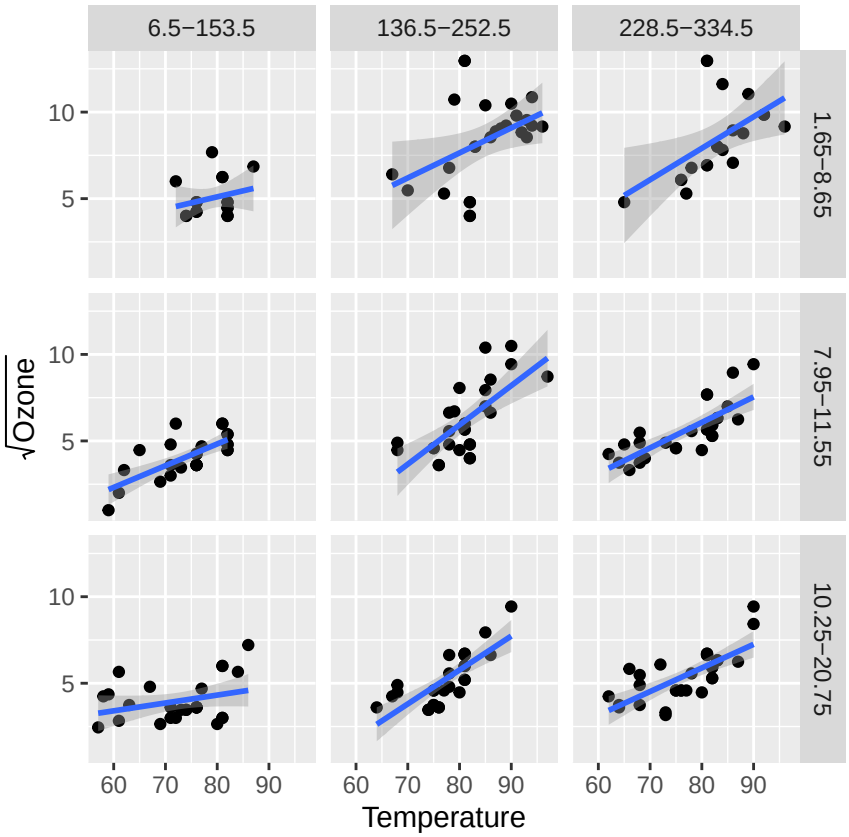



```
xyplot(sqrt(Ozone) ~ Temp|Wind_Speed*Solar_R,
       data=airquality,
       panel=function(x,y,...){
#             panel.loess(x, y, span=1, degree=1, ...)
            panel.grid()
            panel.lmline(x, y, col="grey",...)
            panel.xyplot(x, y, col=1, cex=0.5, ...)
       },
       aspect=1,
       ylab=list(label=expression(sqrt("Ozone")), cex=0.6),
       xlab=list(label="Temperature", cex=0.6),
       scales=list(x=list(alternating=c(1, 2, 1))),
#       between=list(y=1),
       par.strip.text=list(cex=0.4),
       par.settings=list(axis.text=list(cex=0.4)))
```




```
p <- ggplot(airQ, aes(x=Temp, y=sqrt(Ozone))) +
     geom_point() + facet_grid(Wind_Speed~Solar_Radiation)
print(p+geom_smooth(method="lm") + scale_y_continuous(expression(sqrt("Ozone")))
       + scale_x_continuous(expression("Temperature")) + theme(aspect.ratio=1))
```

```
## `geom_smooth()` using formula 'y ~ x'

## Warning: Removed 82 rows containing non-finite values (stat_smooth).

## Warning: Removed 82 rows containing missing values (geom_point).
```



10.2 The River Cam Example

This data set was collected by M Bruce Beck (1978) on the River Cam near Cambridge England. The data have been used in many studies in over twenty years since Beck's PhD work. The data set includes measurements of water quality variables dissolved oxygen (DO) and biochemical oxygen demand (BOD5), as well as selected variables of weather and physical conditions of the river (daily hours of sunlight, precipitation, water temperature, and river discharge). The data were collected daily from June 6 through August 25, 1972 (81 observations). All the variables were measured at both end of the 4.5 kilometer stretch of the River Cam. They were used for testing and evaluating stream water quality models of DO and BOD (Beck, 1978). The data became famous after the publication of a paper by Beck and Young (1976) on model structure identification, a method for detecting weakness of a model. Beck and Young (1976) concluded that algal growth constituted a significant part of the BOD-DO dynamic of the River Cam. The paper recommended a model that uses a simple low-pass filter to simulate algae growth. This low-pass filter has two thresholds for temperature and the sustained sunlight effect. These two threshold were determined without justification. Let us use the downstream DO as the response and study how DO is affected by other variables.

```
cam <- read.table(paste(dataDIR, "cam.dat", sep="/"), header=T)
head(cam)
```

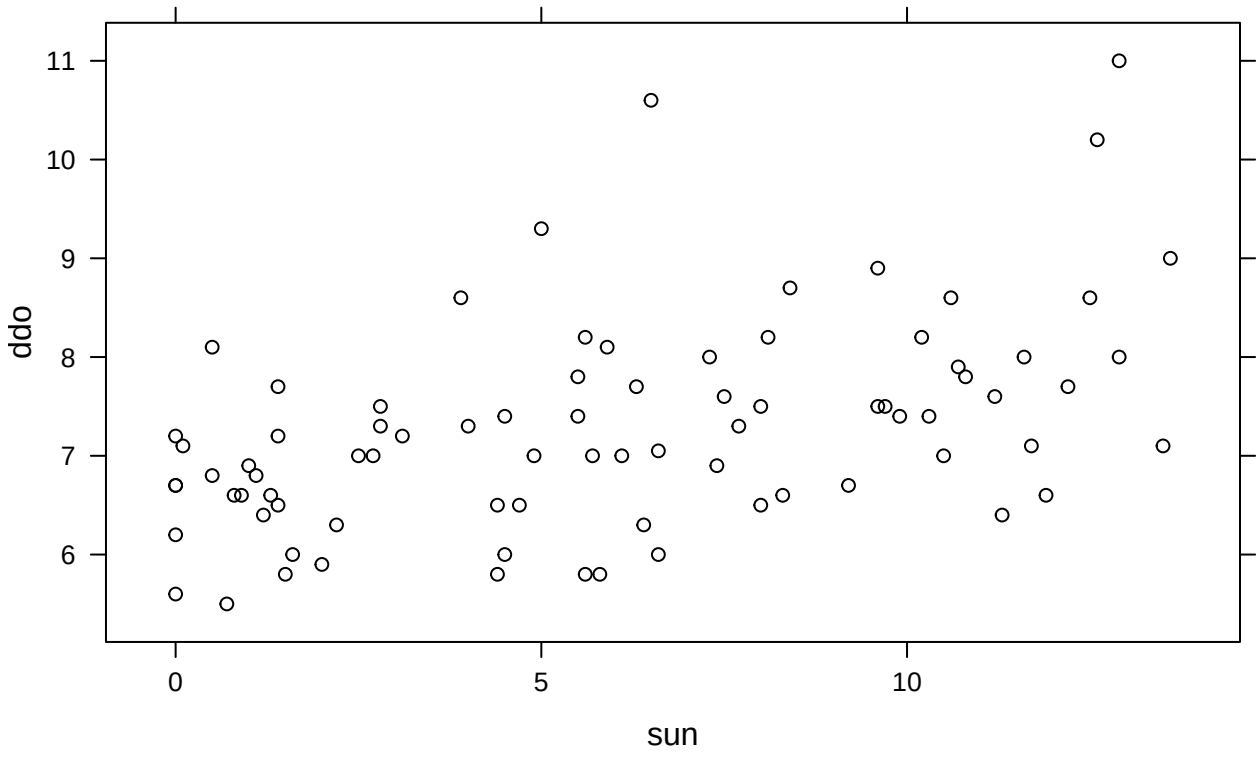
```
##   day  udo ubod ddo dbod    q  temp  sun rain
## 1   0 9.67 2.00 8.0 2.30 1.71 16.50 12.9  0.0
## 2   1 9.56 2.25 8.2 1.05 1.77 16.00  8.1  1.2
## 3   2 9.25 2.40 8.0 1.65 1.64 15.50  7.3  1.5
## 4   3 9.36 2.75 7.5 1.55 1.70 15.50  2.8  3.5
## 5   4 9.57 1.90 7.2 1.60 1.55 15.75  3.1  0.0
```

```
## 6   5 9.43 2.75 7.3 2.90 1.80 15.50  7.7  0.0
```

In the 1970s and 1980s, water quality modeling was largely to understand how pollution (usually organic matter pollution) would change DO. The measure of organic pollution is the amount of DO the pollutant consumes in 5 days (because water in River Thames usually does not linger more than 5 days). The famous DO-BOD model of the 1930s was the basis of many models (including many models still in use). Beck and Young was first to add algal component to the model and demonstrated how to decide when the component is needed. I worked on this data set in late 1990s. Here are some coplots.

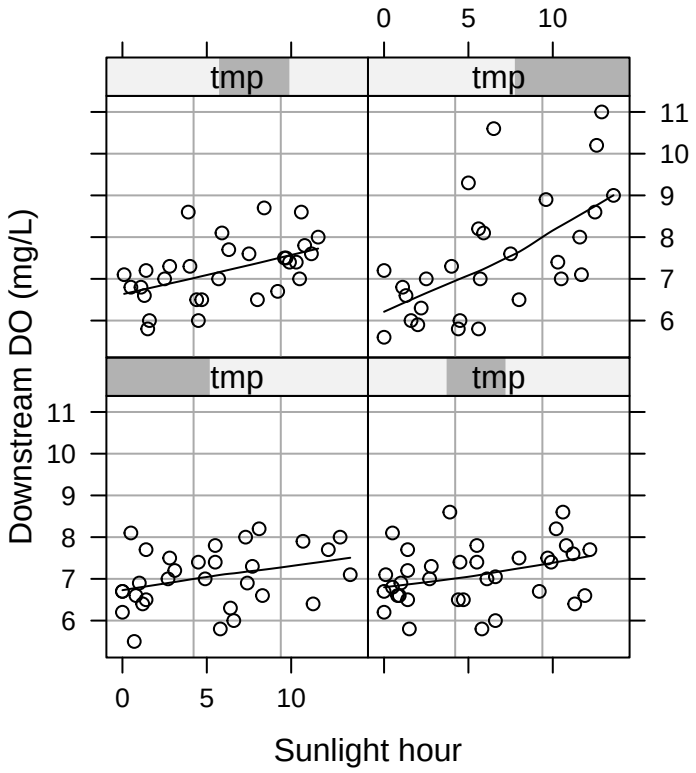
10.2.1 The effect of sun light hour

```
xyplot(ddo ~ sun, data=cam) ## not very clear
```



But algae won't grow if it is too cold, with or without sun.

```
tmp <- equal.count(cam$temp, 4, 0.25)
xyplot(ddo ~ sun | tmp, data = cam,
        xlab = "Sunlight hour", ylab = "Downstream DO (mg/L)",
        panel = function(x, y) {
            panel.grid(h=-1, v= 2)
            panel.xyplot(x, y)
            panel.loess(x,y, span=1)
        },
        aspect = 1)
```



There is always a sunlight effect, but the effect is obvious only when the temperture is high.

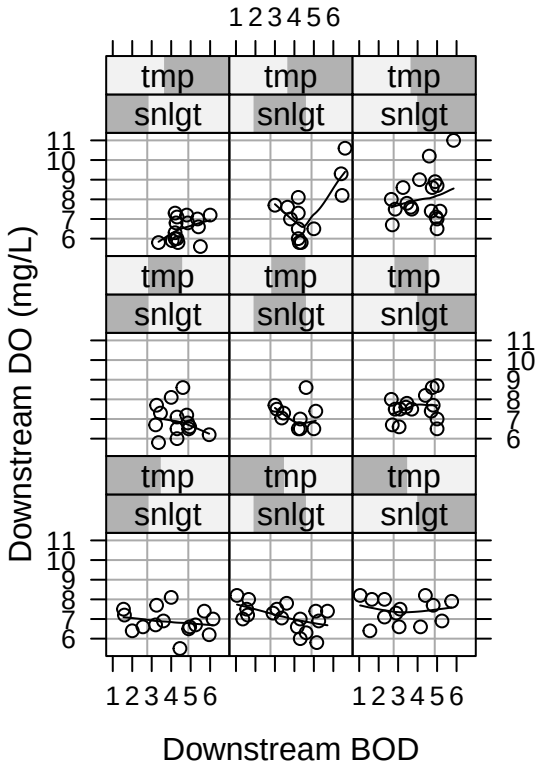
```
snlgt <- equal.count(cam$sun, 6, 0.25)
xyplot(ddo ~ temp | snlgt, data = cam,
        xlab = "Temperature (C)", ylab = "Downstream DO (mg/L)",
        panel = function(x, y) {
            panel.grid(h=-1, v= 2)
            panel.xyplot(x, y)
            panel.loess(x,y, span=1)
        }, layout=c(3,2),
        aspect = 1)
```

Temperature is not a factor when sunlight hour is low. The effect of temperature is likely a hockey stick model with a threshold of 19C.

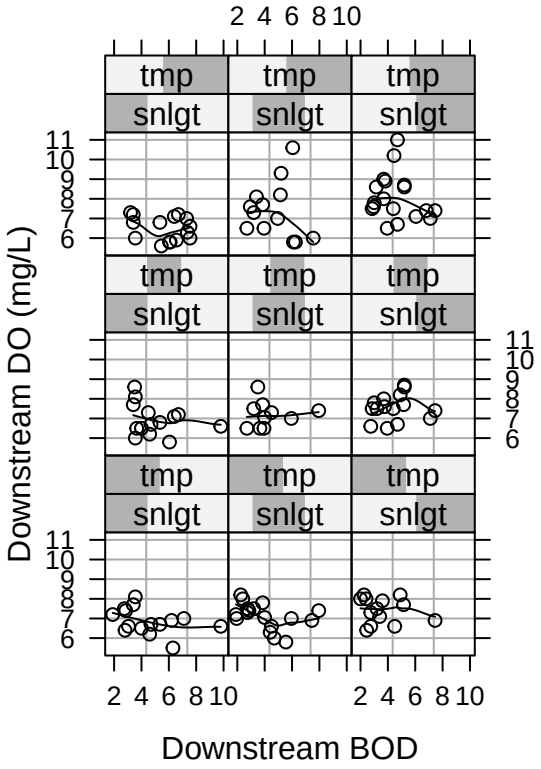
But we can't control sunlight hour and temperature. These plots simply show that these two factors explains much of the variation in DO. When studying the effect of pollution (BOD), we cannot ignore the natural factors.

```
snlgt <- equal.count(cam$sun, 3, 0.25)
tmp <- equal.count(cam$temp, 3, 0.25)
xyplot(ddo ~ dbod | snlgt*tmp, data = cam,
        xlab = "Downstream BOD", ylab = "Downstream DO (mg/L)",
        panel = function(x, y) {
            panel.grid(h=-1, v= 2)
            panel.xyplot(x, y)
##            panel.lmline(x,y)
```

```
            panel.loess(x, y, span=1)
        },
        aspect = 1)
```




```
xyplot(ddo ~ ubod | snlgt*tmp, data = cam,
        xlab = "Downstream BOD", ylab = "Downstream DO (mg/L)",
        panel = function(x, y) {
            panel.grid(h=-1, v= 2)
            panel.xyplot(x, y)
##            panel.lmline(x,y)
            panel.loess(x, y, span=1)
        },
        aspect = 1)
```



I tried many variations of these coplots (number of intervals, how the loess line is fit, ...). The underlying relationship between DO and BOD is highly nonlinear. As a result, we did not find a clean pattern as in the airquality data example.

11 Week 13

11.1 Nutrient Loading and Flow from Maumee

Heidelberg University monitors water quality on several tributaries near Lake Erie. These long-term intensive monitoring data are often used for estimating loadings of various pollutants to Lake Erie. Because the Maumee River basin is the primary agriculture watershed and its loadings of nutrients to Lake Erie western basin are considered as the most important factor in predicting harmful algal blooms in the lake. In fact, almost all predictions of HABs are based on spring and summer loadings of TP from Maumee River.

Because most monitoring programs do not measure water quality daily, calculating loadings of TP based on weekly or less frequently sampled data is a topic of study for many years. Currently, the dominant approach is to develop a regression model using available TP concentration data as the response valable and corresponding flow as the predictor. The model is then used to "estimate'' TP (or other pollutants) concentrations for days without monitroing data.

The concentration – flow relationship is often noisy. As a result, the simple log-log linear regression is often inadequate. Many authors developed load estimation methods for load estimation. Frequently, the Heidelberg monitoring data were used as a test case. To use the Heidelberg data to test a load estimation method, we often sample a subset of the data to build the load estimation model and predict annual loads by predicting the concentrations for the days set-aside during model fitting. I found that almost all load estimation

models failed to consider estimation uncertainty. This uncertainty can be reflected in the differences in a concentration-flow model when fit using different subsets of the Waterville data.

11.1.1 Objectives

The project is to prepare the Waterville data for a study on estimating the uncertainty in the estimated TP loading using various sampling schedules. The uncertainty is represented in the estimation standard deviation.

11.1.2 Methods

There are two sources of uncertainty in an estimated annual TP load. One is the model prediction error and the other is the sampling variation. Most existing methods ignore both. Some authors discussed the model prediction error but often limited to the residual variance. No attention was given to the sampling error.

In this project, we will focus on the sampling error. To evaluate variation due to sampling design, we can use simulation. For example, when evaluating a monthly sampling plan, we can repeatedly sample the data within a canlendar month.

The project will include: - Exploratory data analysis - Adding categorical variables to represent two sampling plans

11.1.3 Results

```
wvldata <- tbl_df(read.csv(paste(dataDIR, "maumeedata.csv", sep="/"),
                  header=T, stringsAsFactors=F, na.strings = "#N/A"))
```

11.1.3.1 Reading data

```
## Warning: `tbl_df()` was deprecated in dplyr 1.0.0.
## Please use `tibble::as_tibble()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was generated.
```

```
wvldata
```

```
## # A tibble: 19,547 x 15
##    Datetime..date.~ Days.since.7410~ Sample.Time.Win~ Flow..CFS SS..mg.L..suspe~
##    <chr>                      <dbl>            <dbl>     <dbl>            <dbl>
##  1 1/10/75 0:00                101              0.25      28958              499
##  2 1/10/75 18:00               102.             0.25      26029              579
##  3 1/11/75 0:00                102              0.25      29693              656
##  4 1/11/75 6:00                102.             0.375     30100              670
##  5 1/11/75 18:00               103.             0.375     30236              632
##  6 1/12/75 0:00                103              0.25      24890              305
##  7 1/12/75 6:00                103.             0.25      28232              671
##  8 1/12/75 12:00               104.             0.25      27250              252
##  9 1/12/75 18:00               104.             0.25      25903              372
## 10 1/13/75 0:00                104              0.25      17880              408
## # ... with 19,537 more rows, and 10 more variables: TP..mg.L.as.P <dbl>,
```

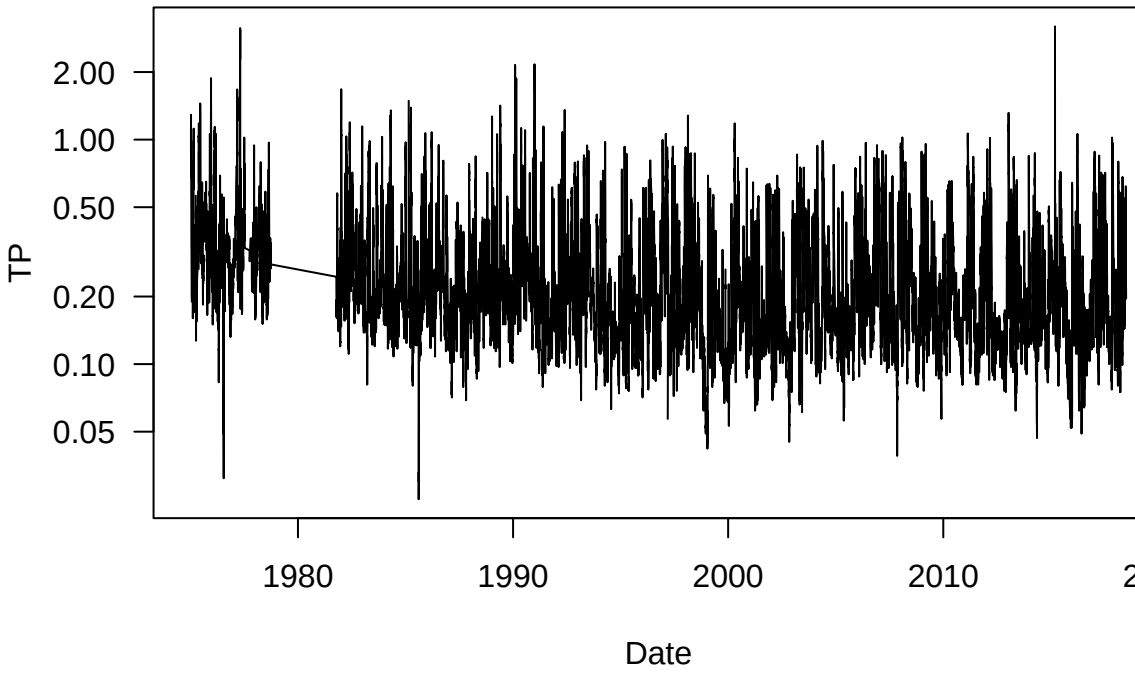
```
## #   SRP..mg.L..as.P <dbl>, NO23..mg.L.as.N <dbl>,
## #   TKN..mg.L..Total.Kjeldahl.nitrogen. <dbl>, Chloride..mg.L <dbl>,
## #   Sulfate..mg.L <dbl>, Silica..mg.L <dbl>, Conductivity.._mho <int>,
## #   Future..NA <lgl>, Month <int>
```

```
names(wvldata)<-c(
"Date", "Days741001","SampleWindow","Flow","SS","TP","SRP",
"NO23","TKN","Chloride","Sulfate","Silica","Conductivity",
"Future","Month")
  wvldata[wvldata<0] <-  NA
```

```
wvldata$Rdate <- as.Date(wvldata$Date, format="%m/%d/%y %H:%M")
  wvldata$mnth <- ordered(format(wvldata$Rdate, "%b"), levels=month.abb)
  wvldata$yrmn <- format(wvldata$Rdate, "%Y-%b")
  wvldata$yrwk <- format(wvldata$Rdate, "%Y-%U")
  wvldata$week <- format(wvldata$Rdate, "%U")
  wvldata$wknd <- format(wvldata$Rdate, "%w") ## weekend = no sampling
  wvldata$wknd <- wvldata$wknd==0 | wvldata$wknd==6
  wvldata$julian <- format(wvldata$Rdate, "%j")
  wvldata$yr <- format(wvldata$Rdate, "%Y")
```

11.1.3.2 Processing dates

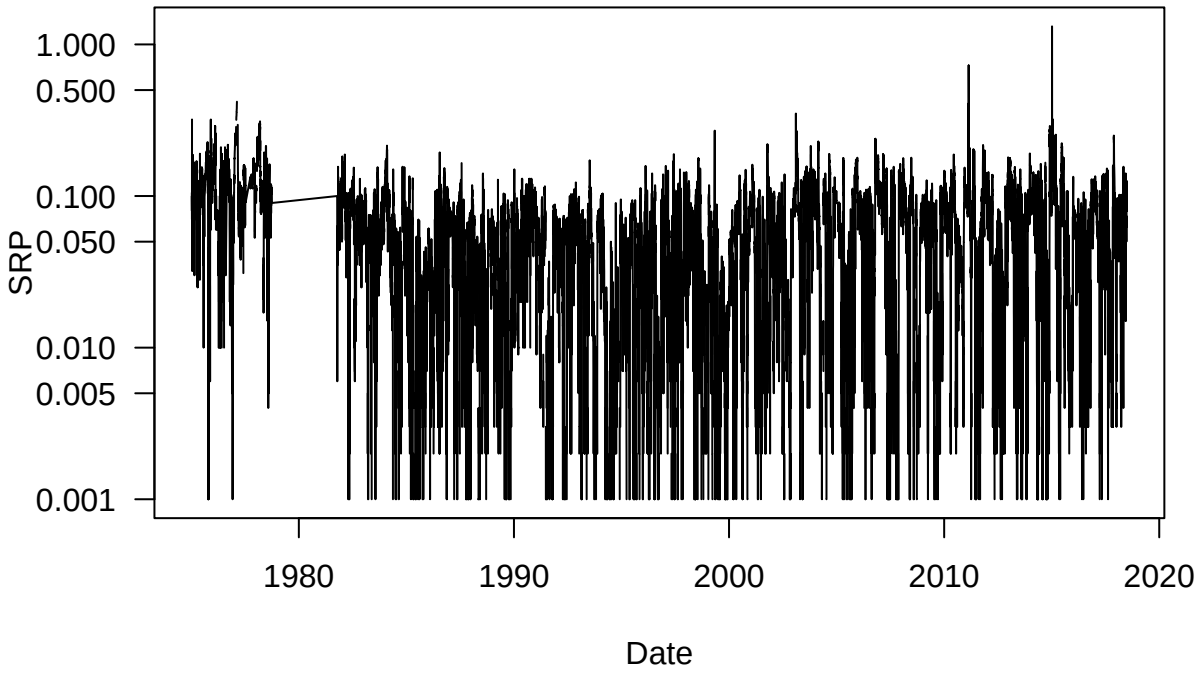
```
  plot(TP ~ Rdate, data=wvldata, type="l", las=1, xlab="Date", ylab="TP", log="y")
```



11.1.3.3 Basic plots

```
plot(SRP ~ Rdate, data=wvldata, type="l", xlab="Date", ylab="SRP", las=1, log="y")
```

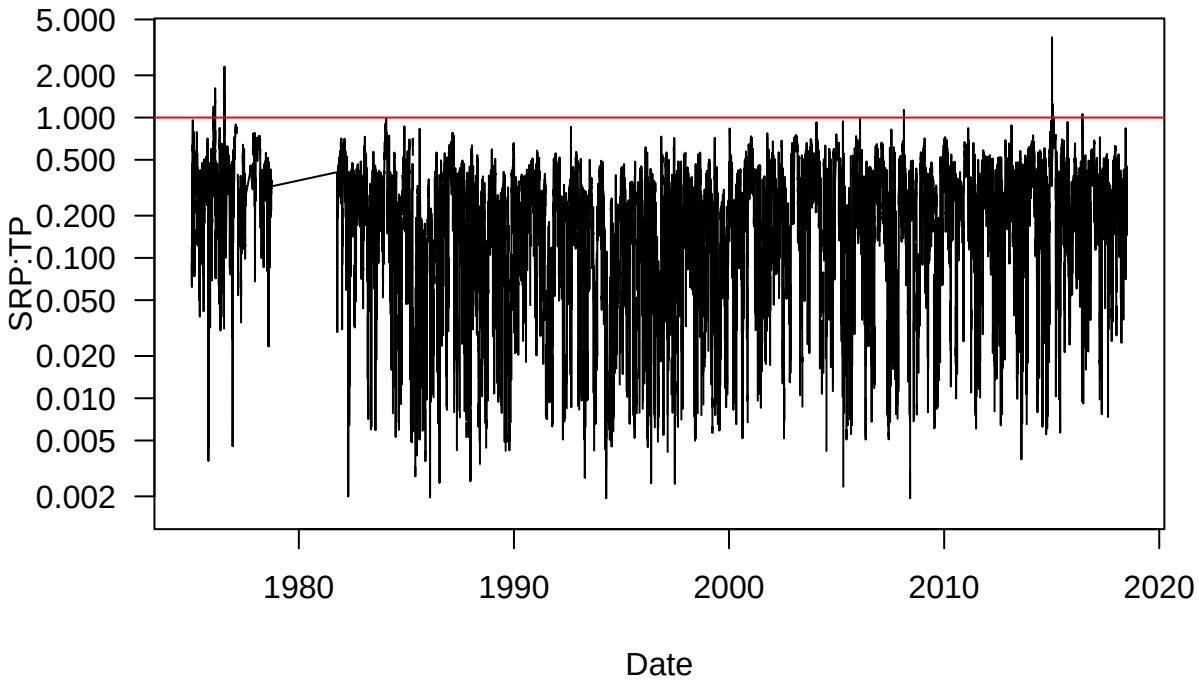
```
## Warning in xy.coords(x, y, xlabel, ylabel, log): 295 y values <= 0 omitted from
## logarithmic plot
```



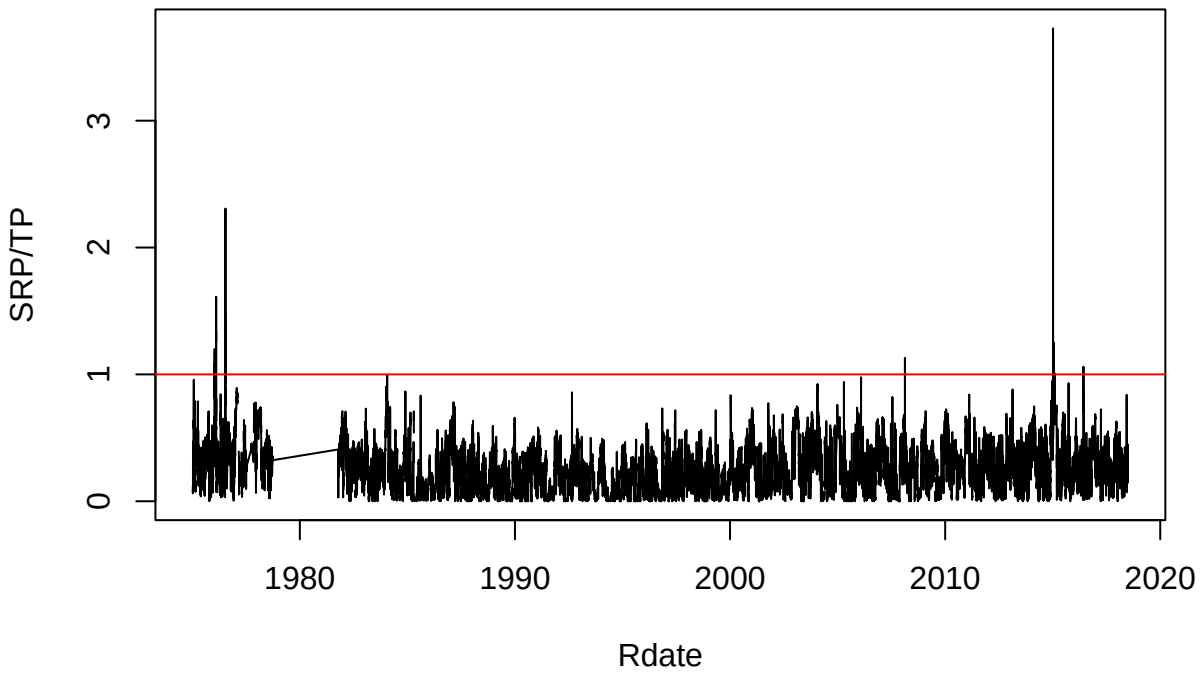

```
  plot(SRP/TP ~ Rdate, data=wvldata, type="l", xlab="Date", ylab="SRP:TP", las=1, log="y")
```

```
## Warning in xy.coords(x, y, xlabel, ylabel, log): 294 y values <= 0 omitted from
## logarithmic plot
```

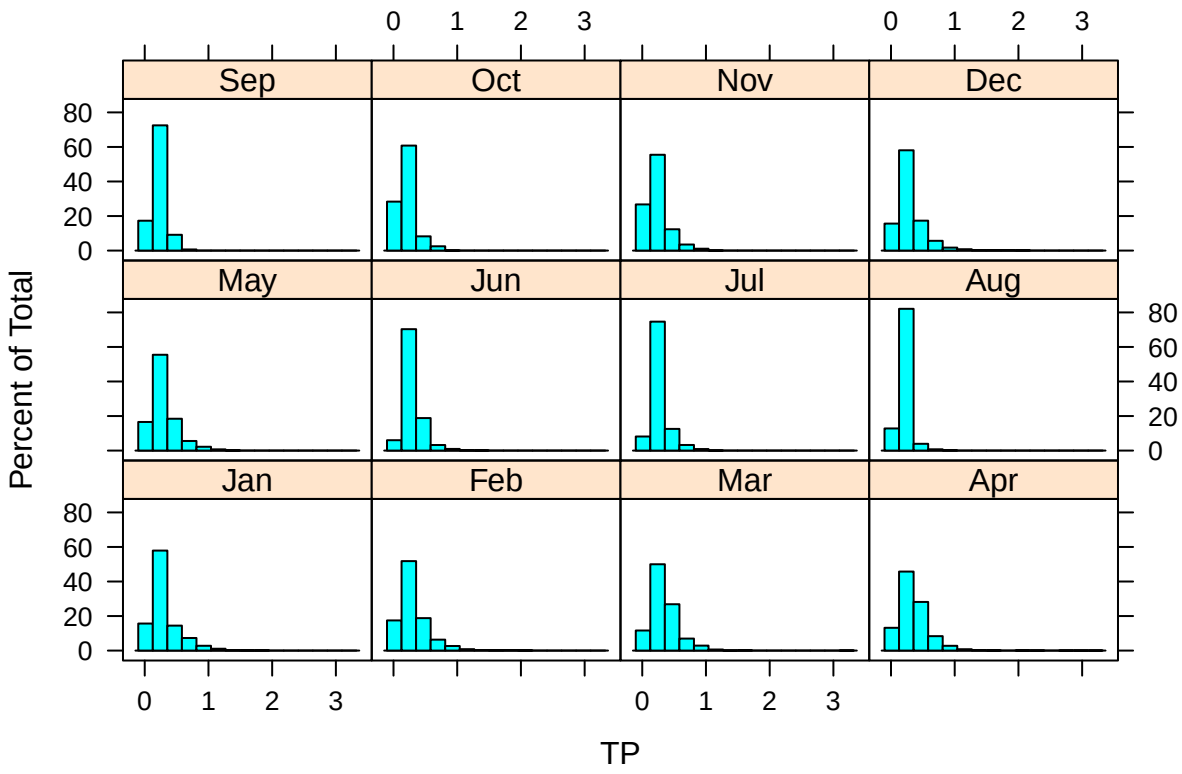
```
  abline(h=1, col="red")
```



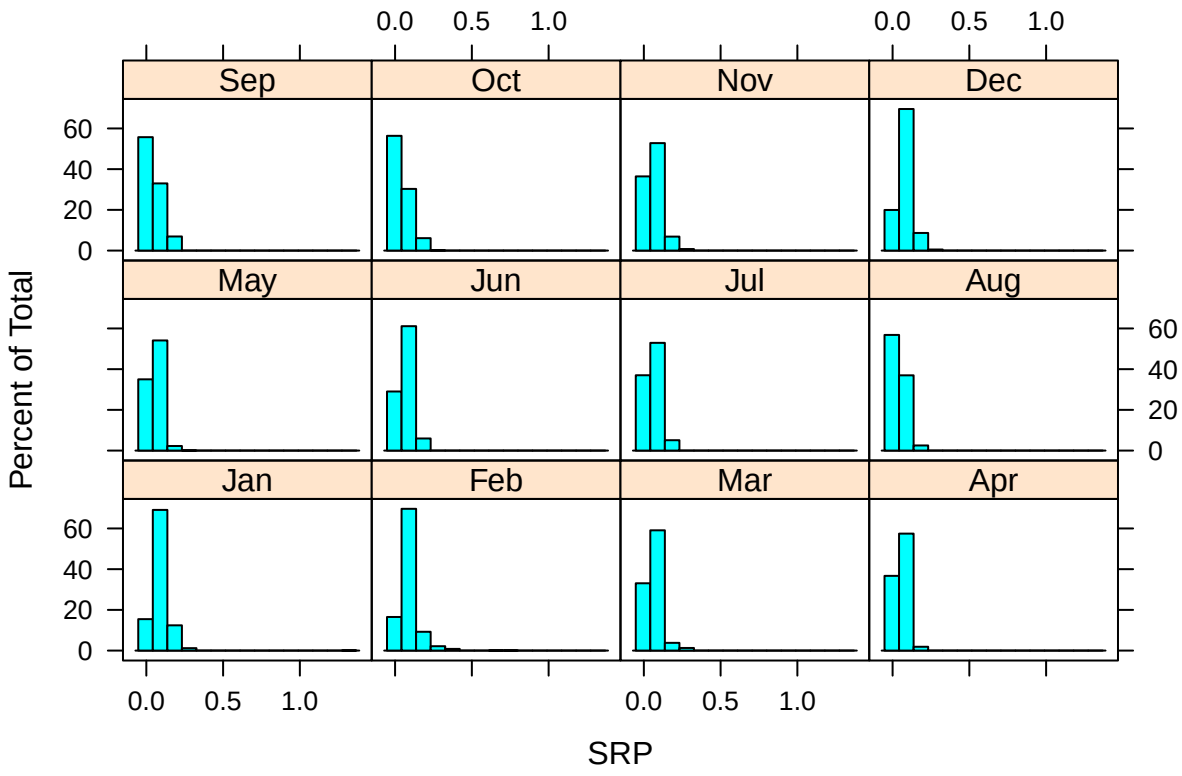

```
plot(SRP/TP ~Rdate, data=wvldata, type="l")
abline(h=1, col="red")
```



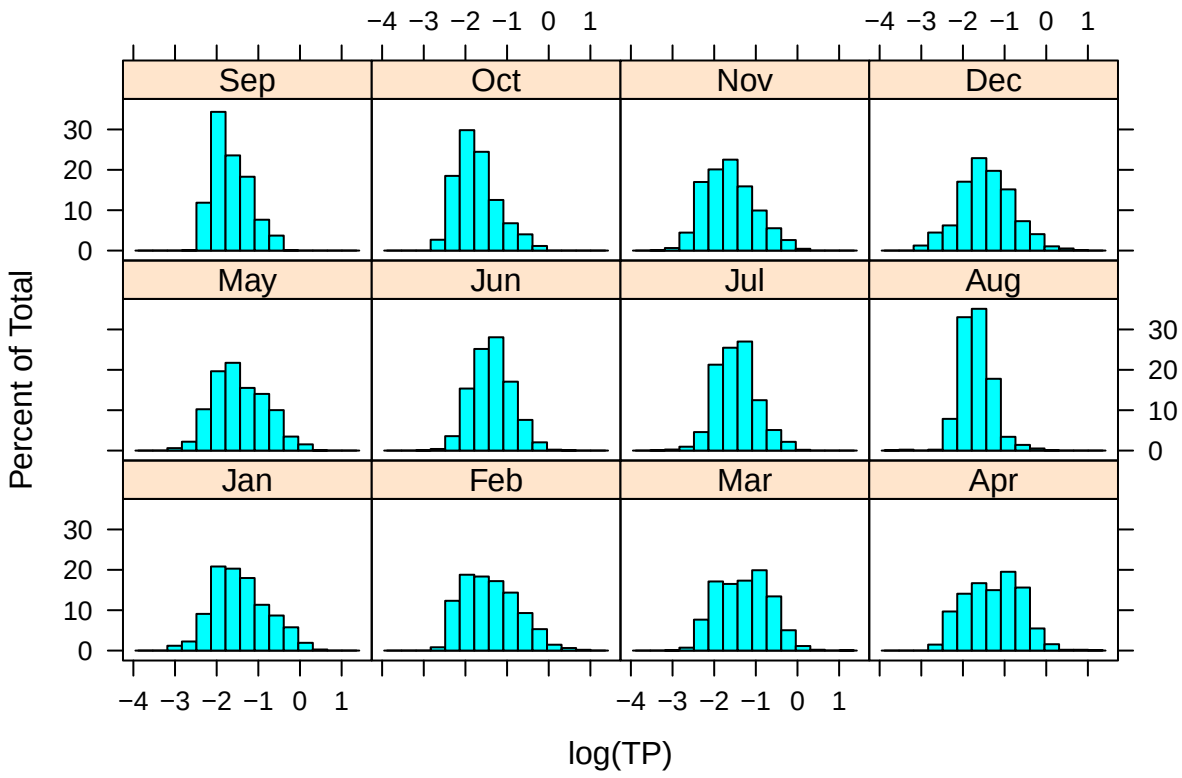

```
histogram(~TP|mnth, data=wvldata)
```



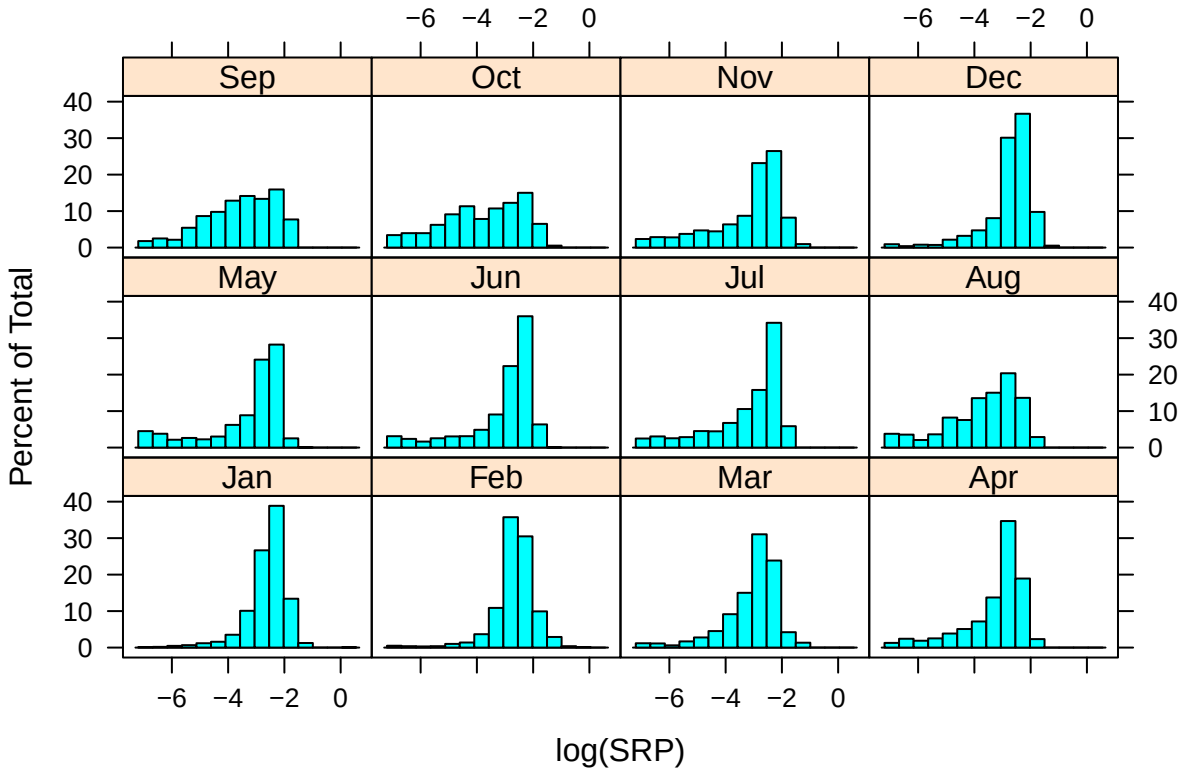

```
histogram(~SRP|mnth, data=wvldata)
```



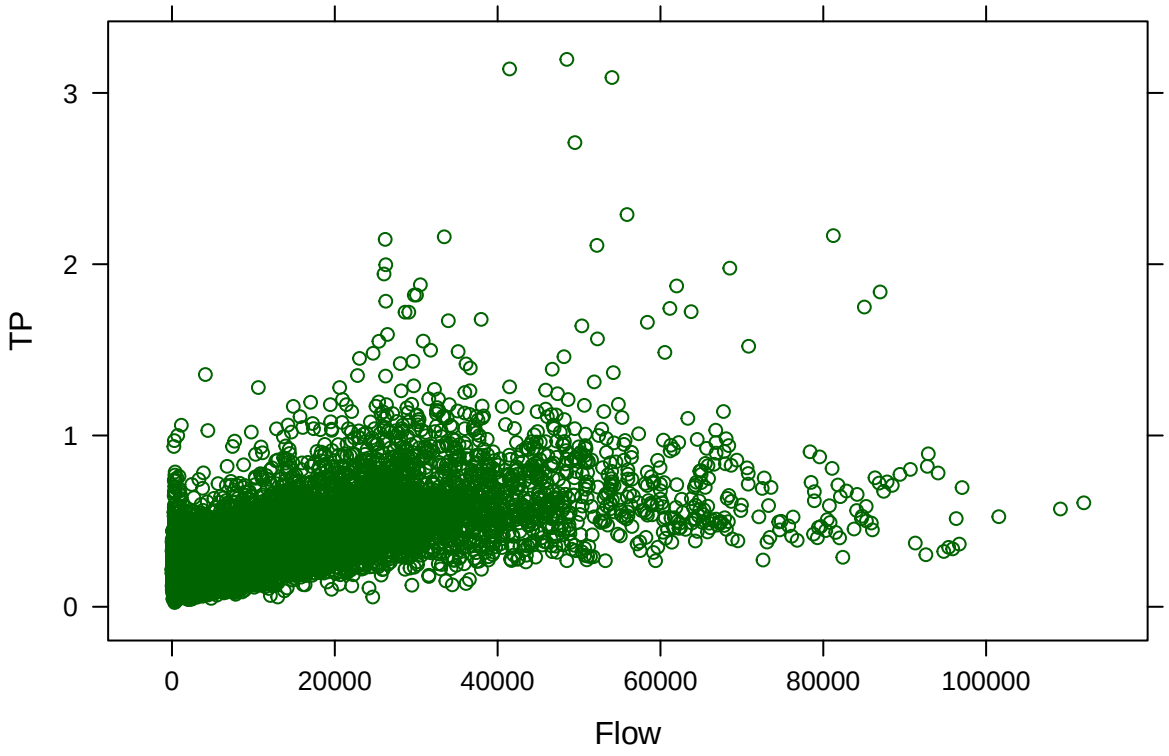

```
histogram(~log(TP)|mnth, data=wvldata)
```



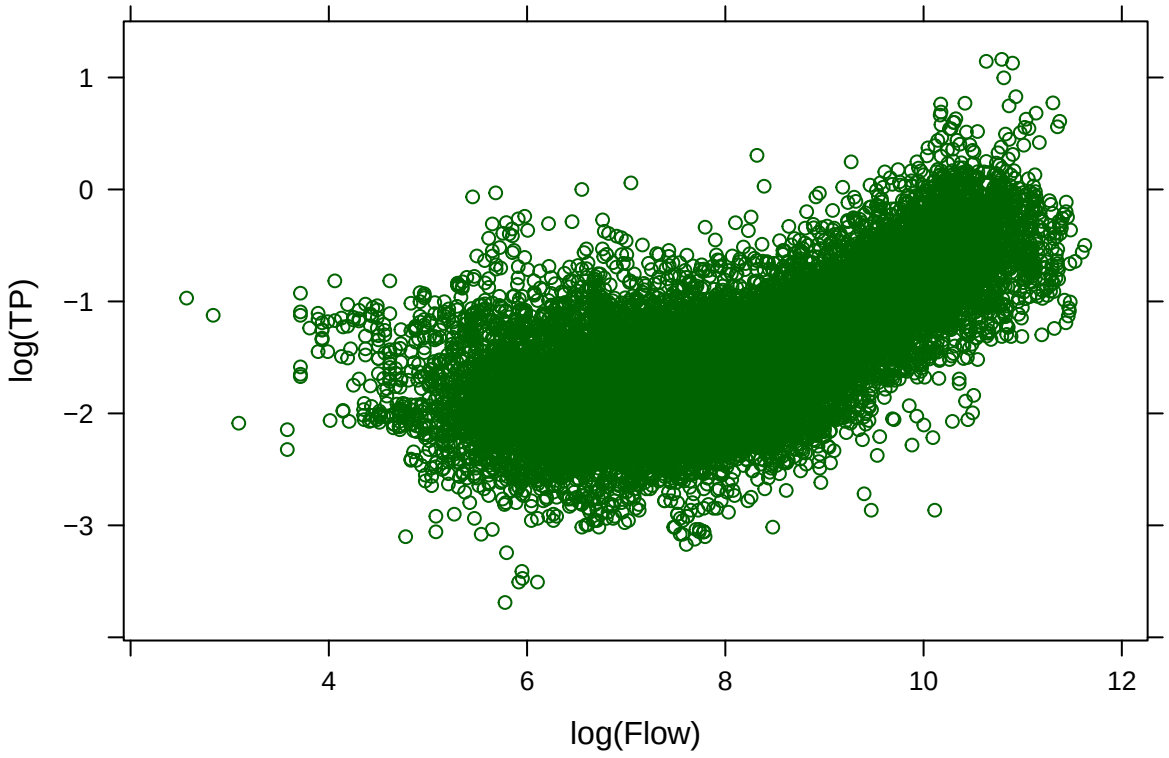

```
histogram(~log(SRP)|mnth, data=wvldata) ## negative SRP
```



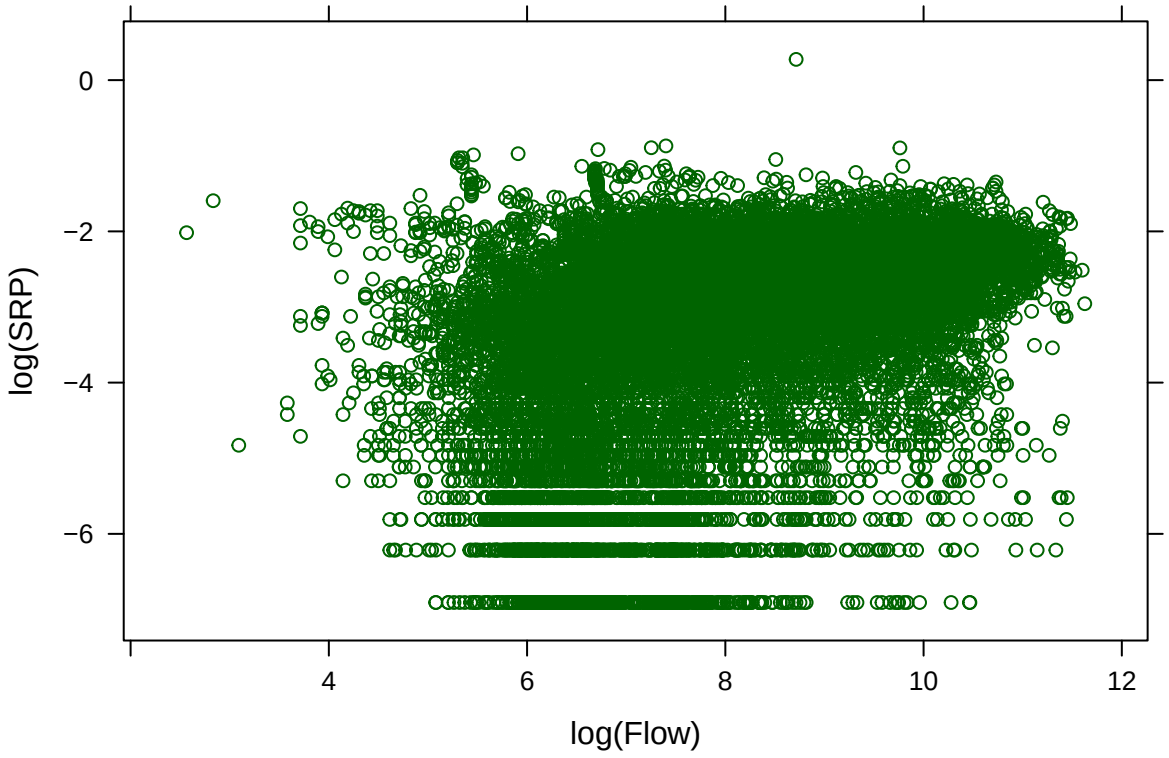

```
trellis.par.set(theme=col.whitebg())
xyplot(TP~Flow, data=wvldata)
```



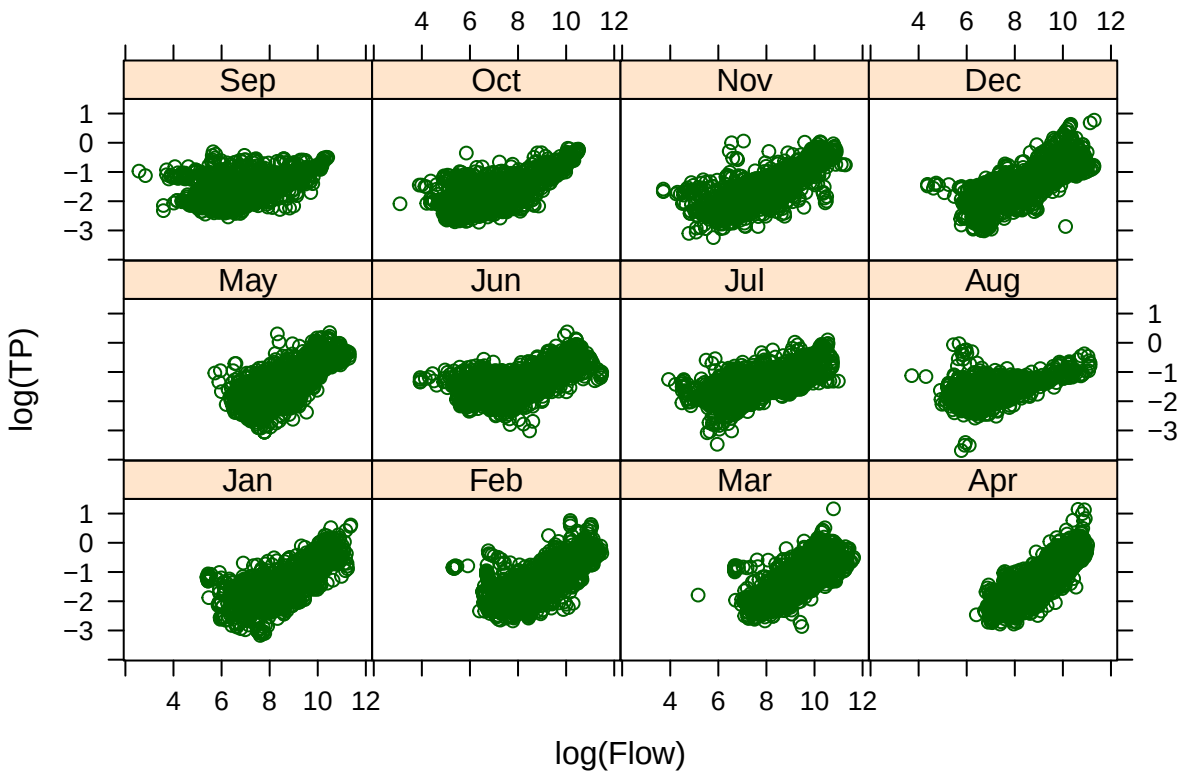

```
xyplot(log(TP)~log(Flow), data=wvldata)
```



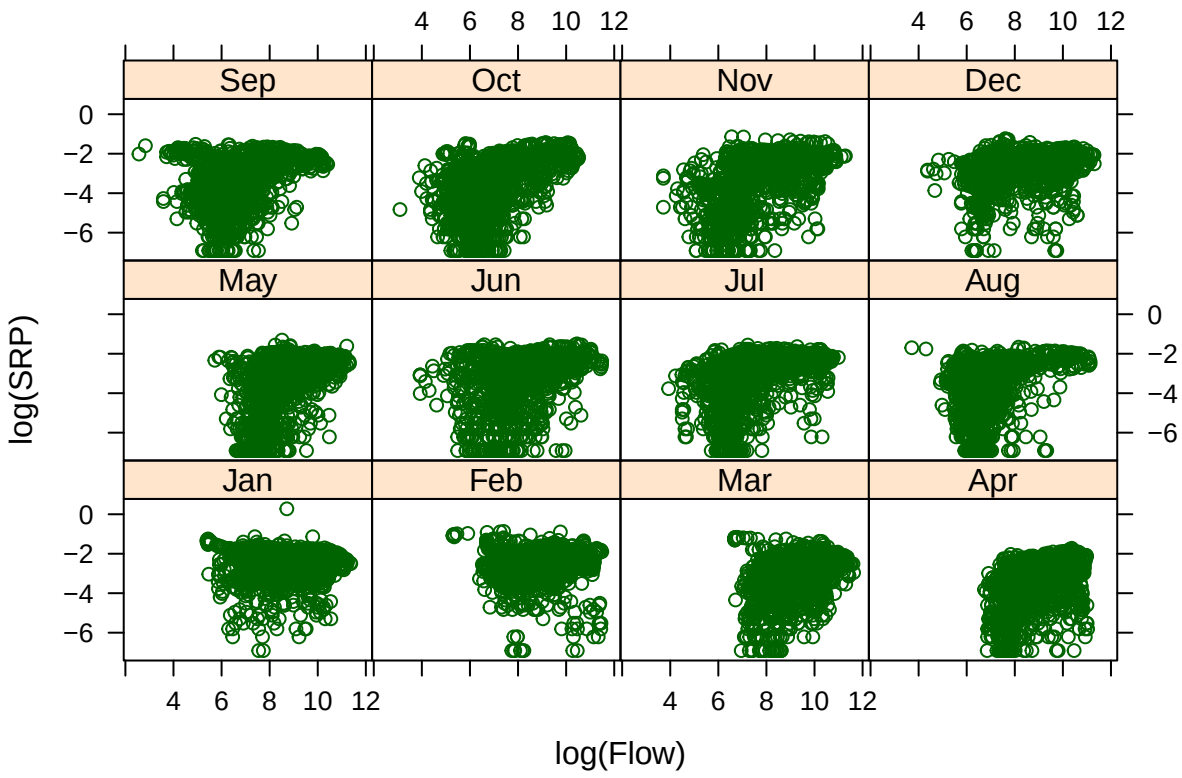

```
xyplot(log(SRP)~log(Flow), data=wvldata)
```




```
xyplot(log(TP)~log(Flow)|mnth, data=wvldata)
```




```
xyplot(log(SRP)~log(Flow)|mnth, data=wvldata)
```



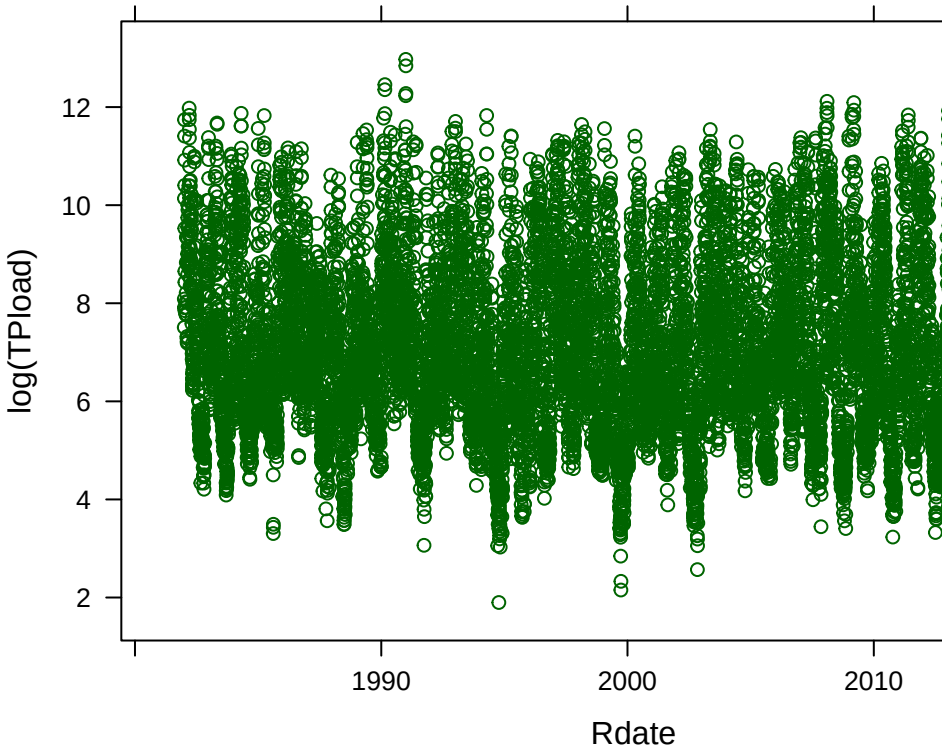
11.1.3.4 Use `reshape2` Using `reshape`, we trim the data to include only the necessary variables.

```
wvl.molten <- melt(as.data.frame(wvldata), id=c("Rdate", "mnth","yrmn","yrwk","julian","yr"),  measure.

## A known problem with `dplyr`: when using `tbl_df`, `melt` will return an error message
### Error in match.names(clabs, names(xi)) : names do not match previous names

tmp <- wvl.molten$yr > "1981"

wvl.daily <- dcast(wvl.molten[tmp,], Rdate ~ variable, mean)
tmp <- range(wvl.daily$Rdate)
date <- seq(tmp[1], tmp[2], 1)
temp.dates <- data.frame(Rdate=date,
                         mnth=ordered(format(date, "%b"), levels=month.abb),
                         yrmn = format(date, "%Y-%b"),
                         yrwk = format(date, "%Y-%U"),
                         week = format(date, "%U"),
                         wknd = format(date, "%w")==0 | format(date, "%w")==6, ## weekend = no sampling
                         julian = format(date, "%j"),
                         yr = format(date, "%Y")
                       )

  wvl.daily <- merge(x=wvl.daily, y=temp.dates, by="Rdate", all=T)
  wvl.molten2 <- melt(wvl.daily, id=c("Rdate", "mnth","yrmn","yrwk","wknd","julian", "week","yr"),
                      measure.vars=c("Flow","SS","TP","SRP","NO23","TKN"))
```

```
tbl_df(wvl.daily)
```

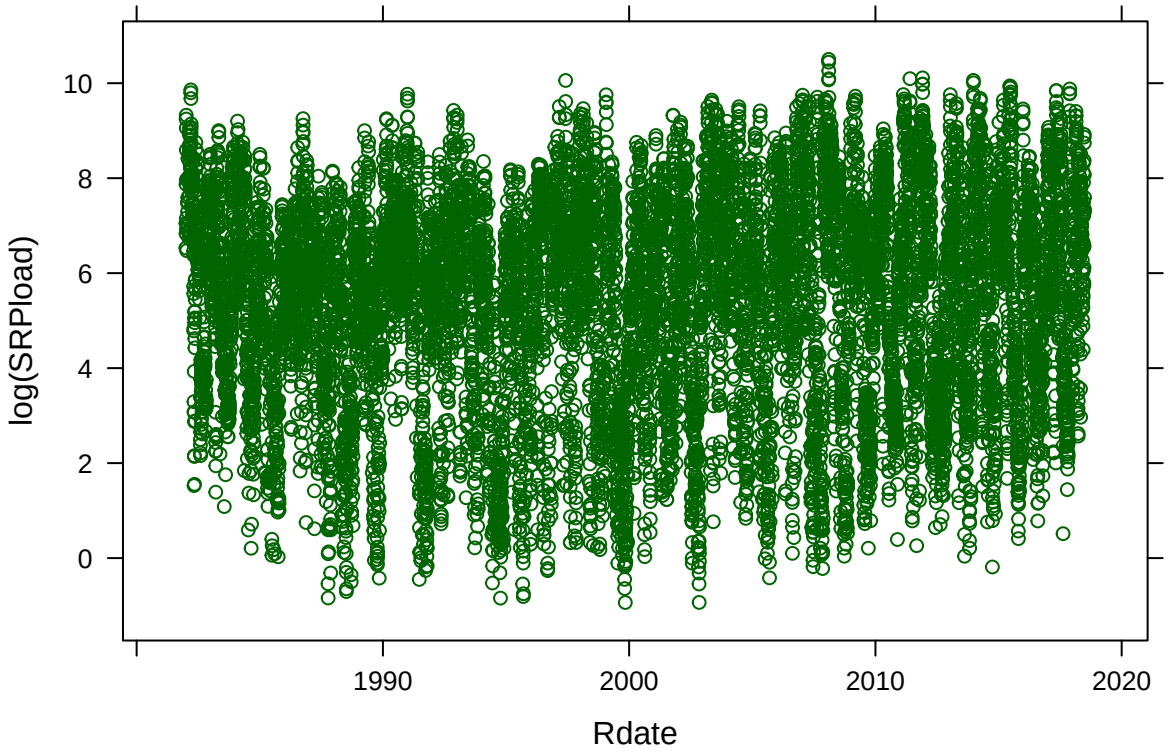
```
## # A tibble: 13,332 x 14
##    Rdate        Flow    SS    TP   SRP  NO23   TKN mnth  yrmn  yrwk  week  wknd
##    <date>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <ord> <chr> <chr> <chr> <lgl>
##  1 1982-01-01  4381   42.5 0.17  0.064  6.08  0.97 Jan   1982~ 1982~ 00    FALSE
##  2 1982-01-02  8005   57   0.166 0.059  5.82  1.12 Jan   1982~ 1982~ 00    TRUE
##  3 1982-01-03  7572   27.5 0.147 0.05   5.83  1.08 Jan   1982~ 1982~ 01    TRUE
##  4 1982-01-04 16696. 726.  0.618 0.064  5.70  1.94 Jan   1982~ 1982~ 01    FALSE
##  5 1982-01-05 38935. 890.  1.32  0.109  5.72  2.75 Jan   1982~ 1982~ 01    FALSE
##  6 1982-01-06 35276. 395.  1.04  0.13   6.22  2.22 Jan   1982~ 1982~ 01    FALSE
##  7 1982-01-07 28234. 238.  0.799 0.124  6.46  1.93 Jan   1982~ 1982~ 01    FALSE
##  8 1982-01-08 21072  109.  0.64  0.105  6.64  1.58 Jan   1982~ 1982~ 01    FALSE
##  9 1982-01-09 11475   50.8 0.489 0.103  6.93  1.27 Jan   1982~ 1982~ 01    TRUE
## 10 1982-01-10  5912   37.5 0.396 0.089  6.97  1.2  Jan   1982~ 1982~ 02    TRUE
## # ... with 13,322 more rows, and 2 more variables: julian <chr>, yr <chr>
```

```
wvl.daily$TPload <- wvl.daily$TP * wvl.daily$Flow * 0.0283168 * 0.001 * 86400 ## kg/day
wvl.daily$TNload <- (wvl.daily$TKN+wvl.daily$NO23) * wvl.daily$Flow * 0.0283168 * 0.001 * 86400
wvl.daily$TKNload <- wvl.daily$TKN * wvl.daily$Flow * 0.0283168 * 0.001 * 86400
wvl.daily$SRPload <- wvl.daily$SRP * wvl.daily$Flow * 0.0283168 * 0.001 * 86400

xyplot(log(TPload) ~ Rdate, data=wvl.daily)
```

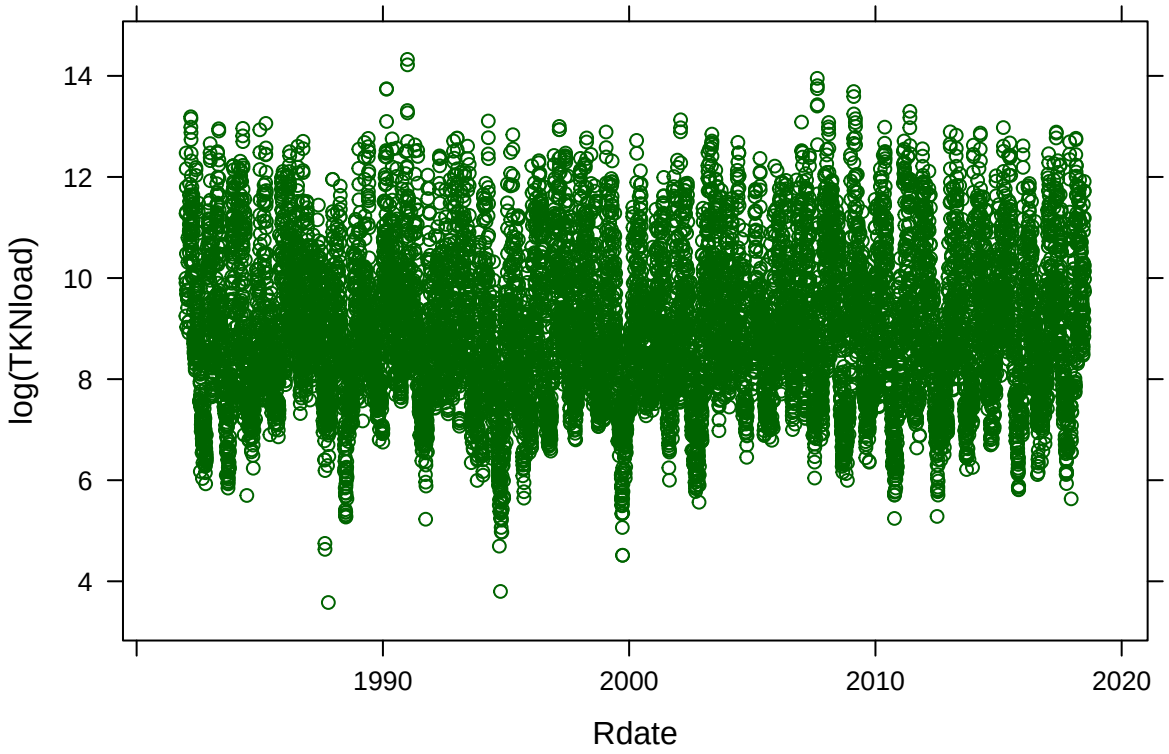


11.1.3.5 Calculating Daily Loads

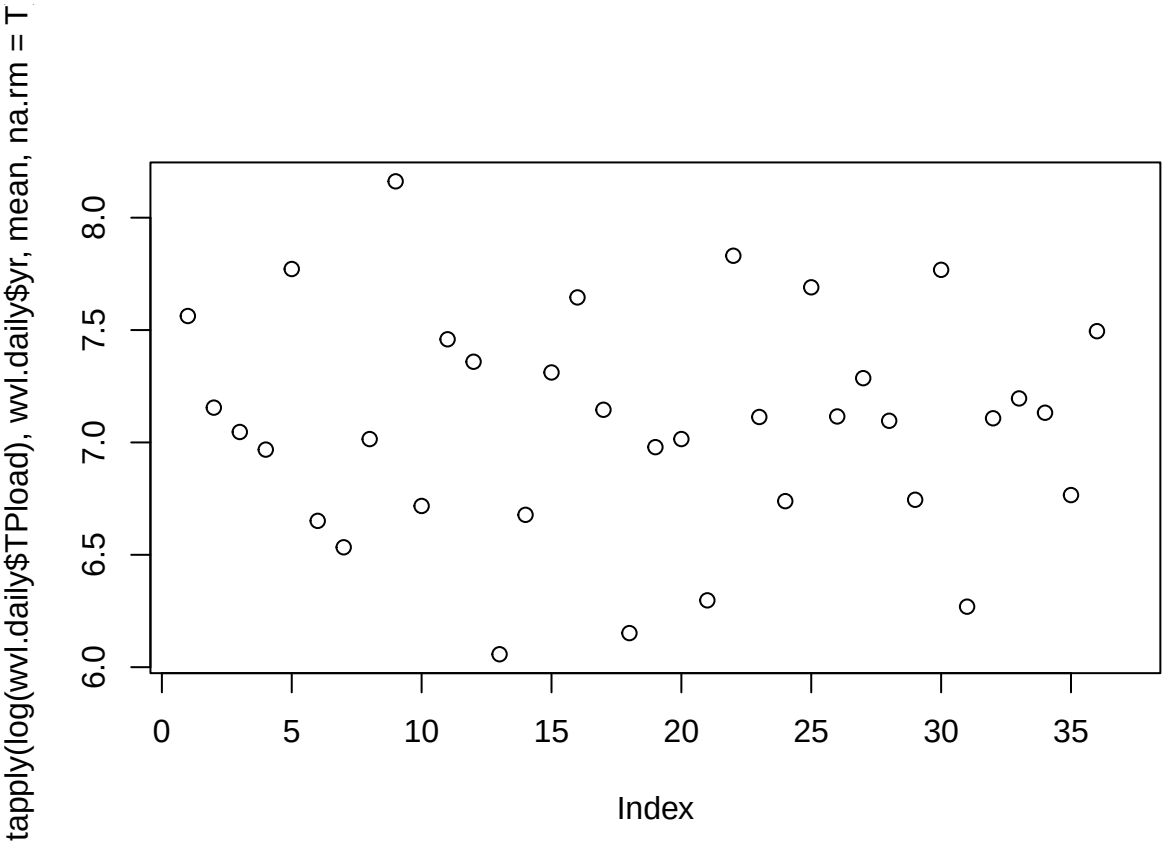
```
xyplot(log(SRPload) ~ Rdate, data=wvl.daily)
```



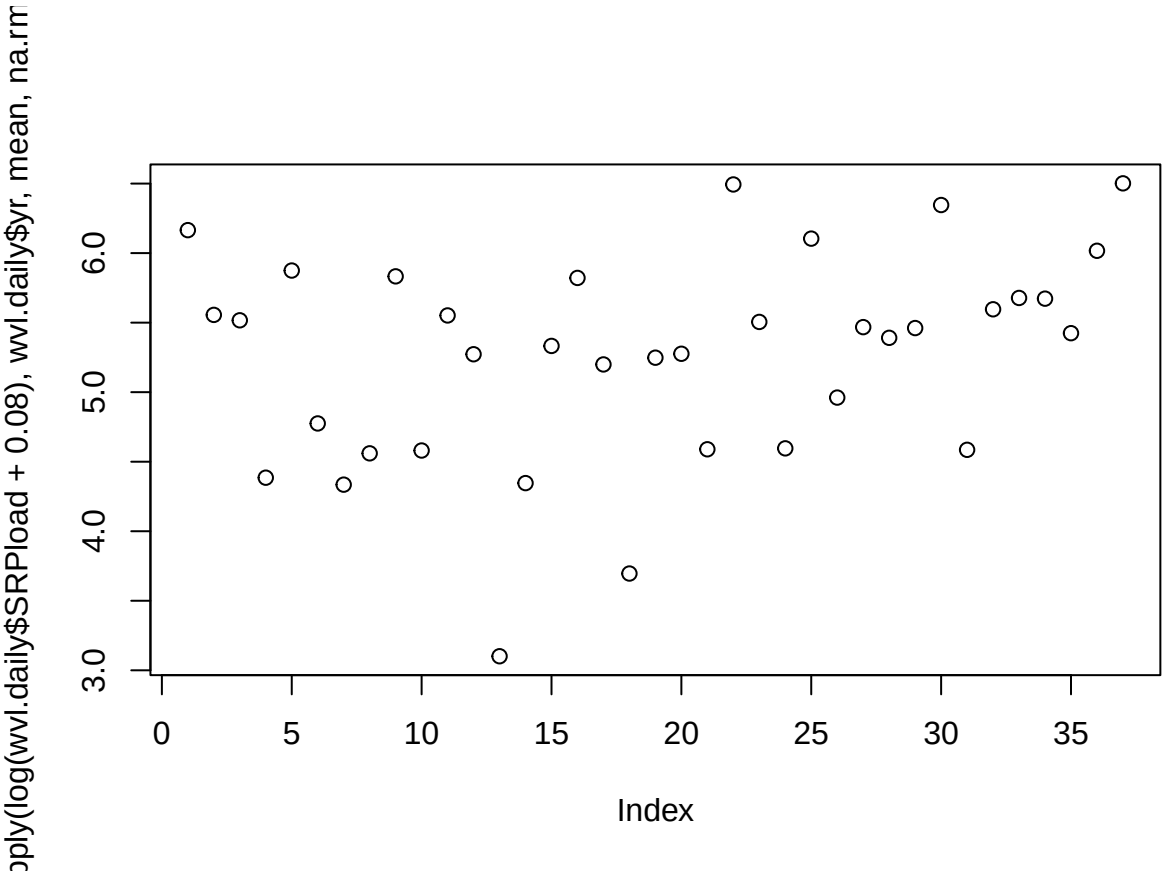

```
xyplot(log(TKNload) ~ Rdate, data=wvl.daily)
```




```
plot(tapply(log(wvl.daily$TPload), wvl.daily$yr, mean, na.rm=T))
```




```
plot(tapply(log(wvl.daily$SRPload+0.08), wvl.daily$yr, mean, na.rm=T))
```



The cumulative sum is calculated by the function `cumsum`

```
  tp.cumsum <- tapply(wvl.daily$TPload, wvl.daily$yr, cumsum)
```

A few missling values in flow and/or concentration resulted in the cumulative loads unusable. A useful way to impute missing values in a data with a two-way table structure is the use of the median polishing (Mosteller and Tukey, 1977). Median polishing a an exploratory data analysis tool. In this case, nutrient loading has a seasonal pattern and a long-term trend. We can use week or month to describe the season and year to describe the long-term trend. In other words, nutrient load is affected by two factors. A simple way to explore the effects of these factors is to assume that their effects are additive. Such that, we can decompose a weekly mean load as a sum of three terms: the long-term trend, the seasonal trend, and the remainder. If we use week as a measure of seasonal effect, we are interested in estimating the weekly means for all years and the data can be transformed into a matrix with rows representing years and columns representing weeks.

```
wvl.weekly <- dcast(wvl.molten2, yr+week ~ variable, median, na.rm=T)
year.weeks <- tapply(wvl.weekly$TP, wvl.weekly$yr, length)

to2 <- function(x){
    ## conver a single digit integer to 0x
    return(ifelse (x<10, paste("0",x, sep=""), as.character(x)))
}

  ## construct a matrix of TP for median polish
  TP.weekly <- matrix(NA, nrow=length(year.weeks), ncol=max(year.weeks))
  for (i in 1:length(year.weeks)){
    for (j in 1:max(year.weeks)){
```

```
      temp <- wvl.weekly$yr==names(year.weeks)[i] & wvl.weekly$week==to2(j-1)
      if (sum(temp)>0)
        TP.weekly[i,j] <- wvl.weekly$TP[temp]
    }
  }

  med.TP <- medpolish(TP.weekly, na.rm=T)
```

```
## 1: 151.6753
## Final: 150.9543
```

The resulting object `med.TP` contains the row (year) and column (season measured by week) effects and the overall median. To replace a missing value of TP, we go back to the daily data file and find the missing value. The year and week associated with the missing value will be used to extract the row and column effect:

```
## TP
  temp <- is.na(wvl.daily$TP)
  if (sum(temp) >0){
    row.yr <- as.numeric(wvl.daily$yr)-min(as.numeric(wvl.daily$yr))+1
    col.wk <- as.numeric(wvl.daily$week) + 1
    wvl.daily$TP[temp] <- med.TP$overall + med.TP$row[row.yr[temp]] + med.TP$col[col.wk[temp]]
  }
```

Now we need to do the same for `SRP`, `TKN`, `NO23`, and `flow`. To make the process tidy, I will write a function.

```
## an R function for median polishing

  NAimpute <- function(col, daily=wvl.daily, weekly=wvl.weekly){
    yr.wks <- tapply(weekly[,col], weekly$yr, length)
    wkly <- matrix(NA, nrow=length(yr.wks), ncol=max(yr.wks))
    for (i in 1:length(yr.wks)){
      for (j in 1:max(yr.wks)){
        temp <- weekly$yr==names(yr.wks)[i] & weekly$week==to2(j-1)
        if (sum(temp)>0)
        wkly[i,j] <- weekly[temp, col]
      }
    }
    med <- medpolish(wkly, na.rm=T)
    tmp <- is.na(daily[,col])
    print(paste("Number of NAs to be imputed:", sum(tmp)))
    if (sum(tmp)>0){
      row.yr <- as.numeric(daily$yr)-min(as.numeric(daily$yr))+1
      col.wk <- as.numeric(daily$week) + 1
      daily[tmp, col] <- med$overall + med$row[row.yr[tmp]] + med$col[col.wk[tmp]]
    }
    return(daily[,col])
  }
```

With this function, we can process the data easily:

```
  wvl.daily$TP <- NAimpute(col="TP")
```

```
## 1: 151.6753
## Final: 150.9543
## [1] "Number of NAs to be imputed: 0"
```

```
  wvl.daily$SRP <- NAimpute(col="SRP")
```

```
## 1: 50.37966
## 2: 49.43145
## Final: 49.34175
## [1] "Number of NAs to be imputed: 1385"
```

```
  wvl.daily$SRP <- ifelse(wvl.daily$SRP<0, 0, wvl.daily$SRP)
  wvl.daily$Flow <- NAimpute(col="Flow")
```

```
## 1: 7567429
## 2: 7477808
## Final: 7471490
## [1] "Number of NAs to be imputed: 755"
```

```
  wvl.daily$NO23 <- NAimpute(col="NO23")
```

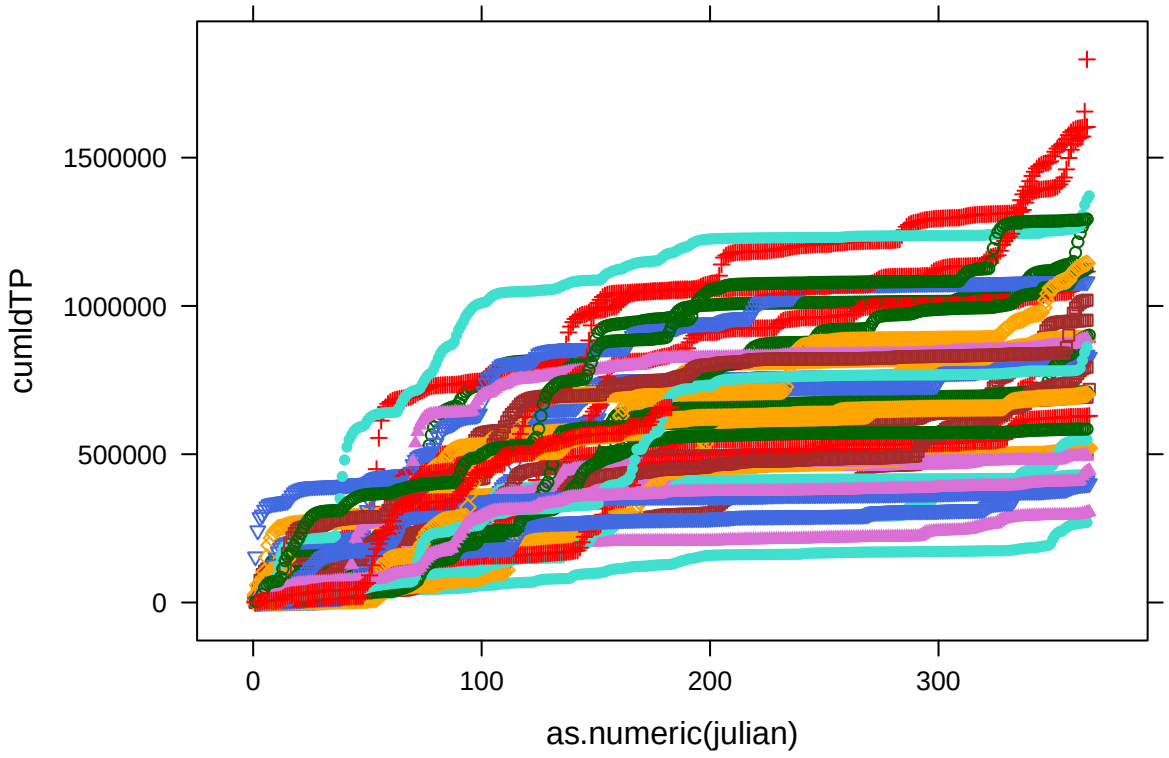
```
## 1: 2996.822
## 2: 2915.593
## Final: 2911.965
## [1] "Number of NAs to be imputed: 934"
```

```
  wvl.daily$TKN <- NAimpute(col="TKN")
```

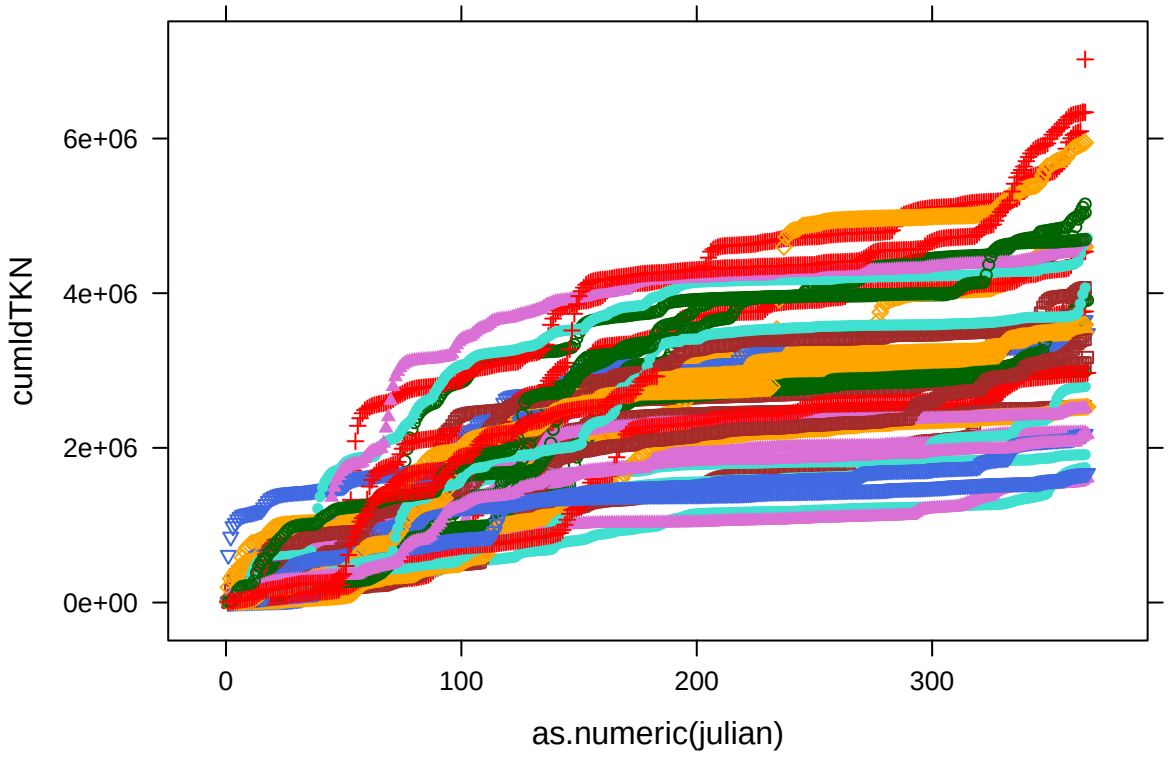
```
## 1: 502.5386
## Final: 500.6853
## [1] "Number of NAs to be imputed: 880"
```

With missing values imputed, I will now calculate daily loads and the cumulative loads

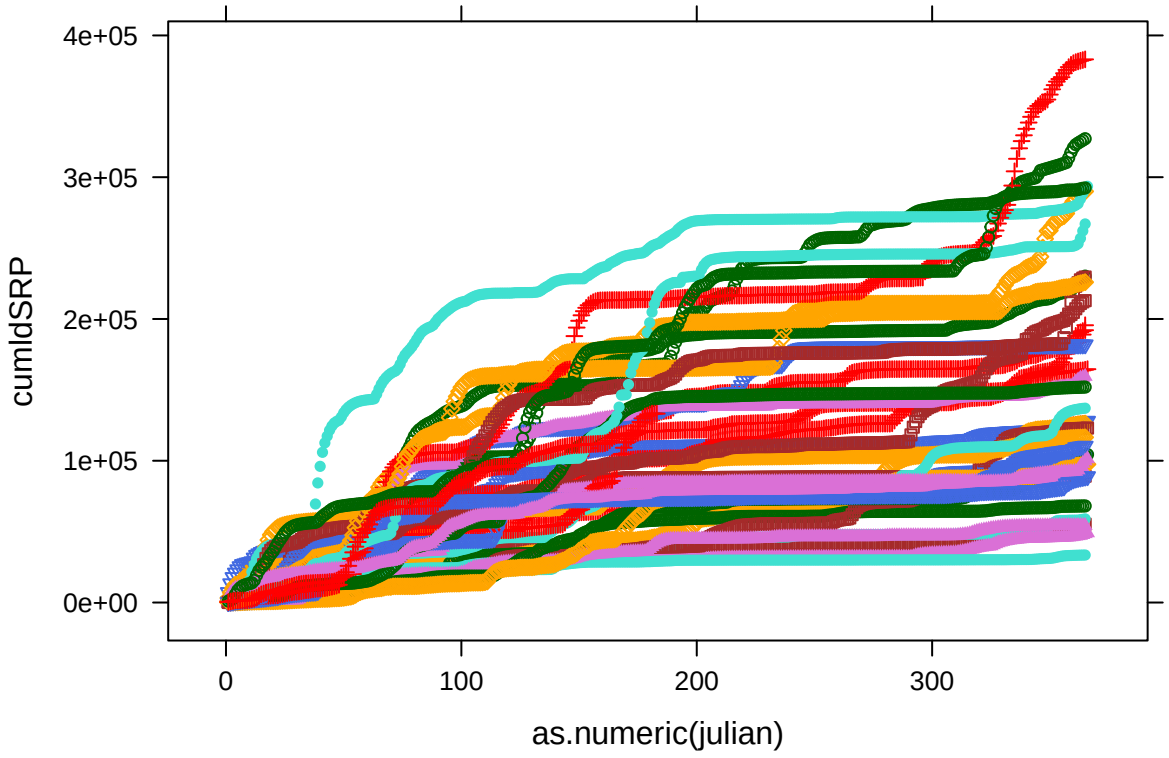
```
  wvl.daily$TPld <- as.vector(wvl.daily$TP) * as.vector(wvl.daily$Flow)
  wvl.daily$cumldTP <- unlist(tapply(wvl.daily$TPld, wvl.daily$yr, cumsum))
  wvl.daily$SRPld <- wvl.daily$SRP * wvl.daily$Flow
  wvl.daily$cumldSRP <- unlist(tapply(wvl.daily$SRPld, wvl.daily$yr, cumsum))
  wvl.daily$TKNld <- wvl.daily$TKN * wvl.daily$Flow
  wvl.daily$cumldTKN <- unlist(tapply(wvl.daily$TKNld, wvl.daily$yr, cumsum))
  wvl.daily$NOxld <- wvl.daily$NO23 * wvl.daily$Flow
  wvl.daily$cumldNOx <- unlist(tapply(wvl.daily$NOxld, wvl.daily$yr, cumsum))
  wvl.daily$cumldFLW <- unlist(tapply(wvl.daily$Flow, wvl.daily$yr, cumsum))

  xyplot(cumldTP ~ as.numeric(julian), data=wvl.daily, group=yr)
```



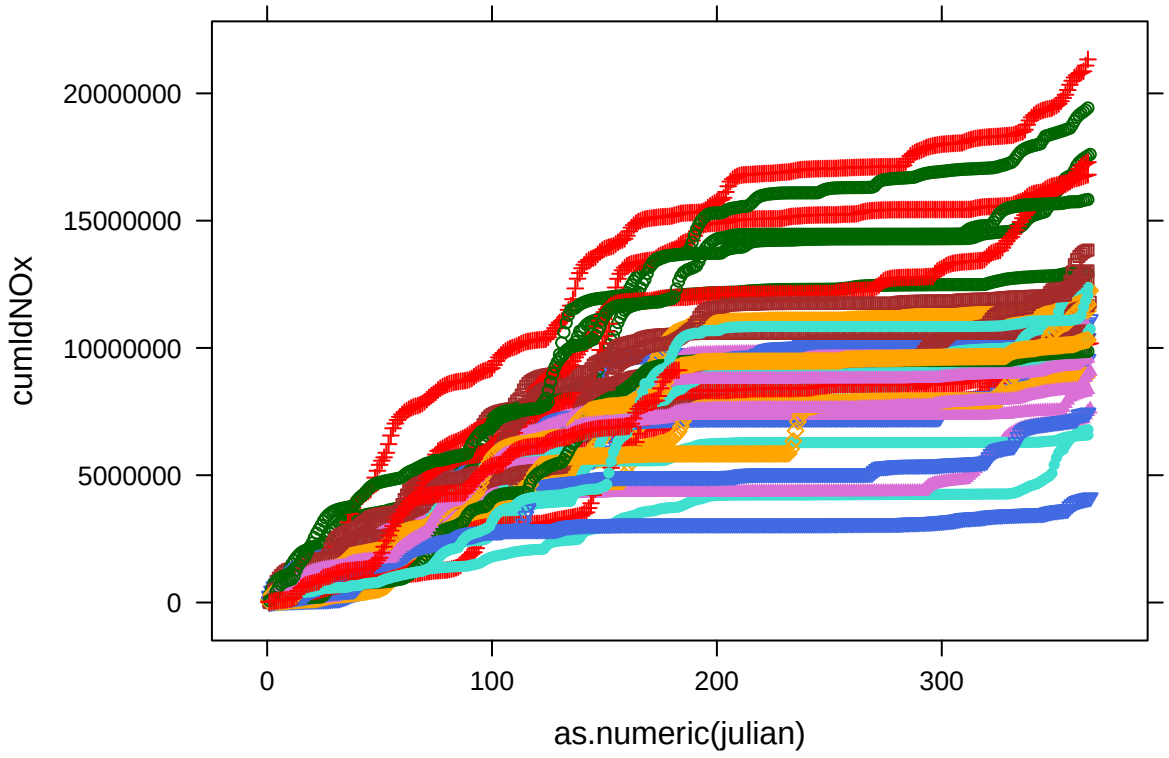

```
xyplot(cumldTKN ~ as.numeric(julian), data=wvl.daily, group=yr)
```



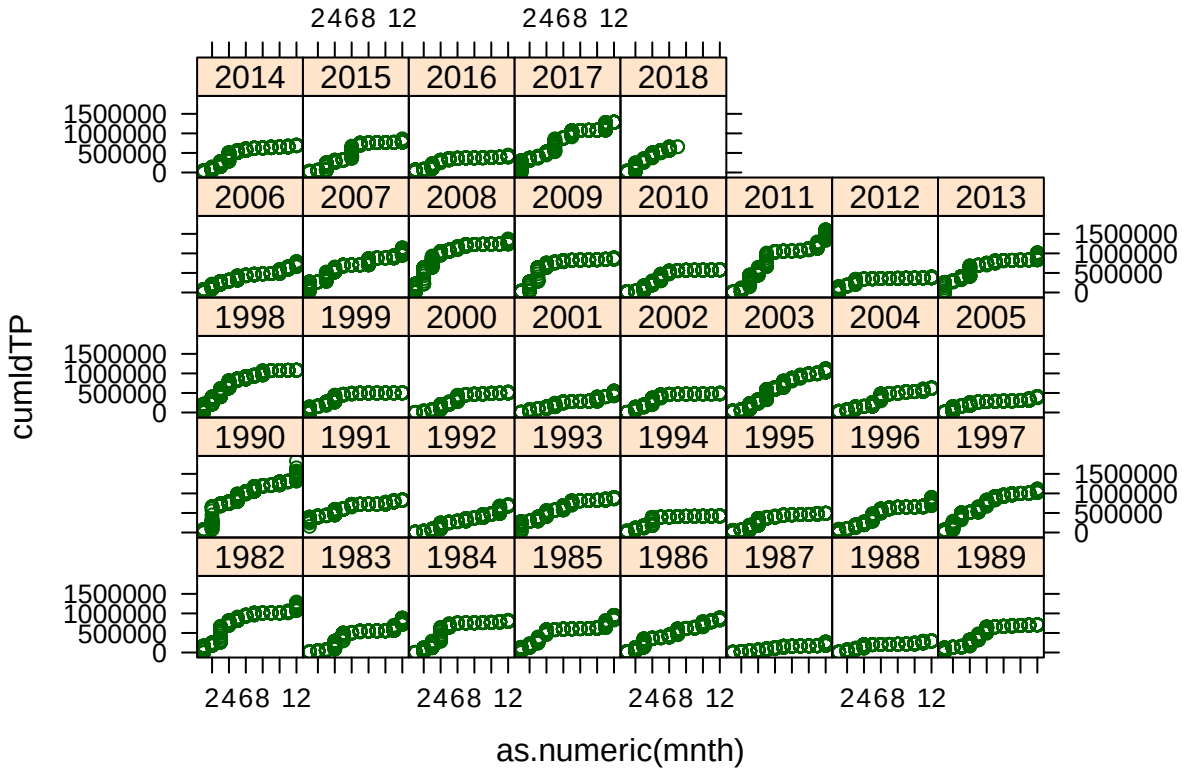

```
xyplot(cumldSRP ~ as.numeric(julian), data=wvl.daily, group=yr)
```



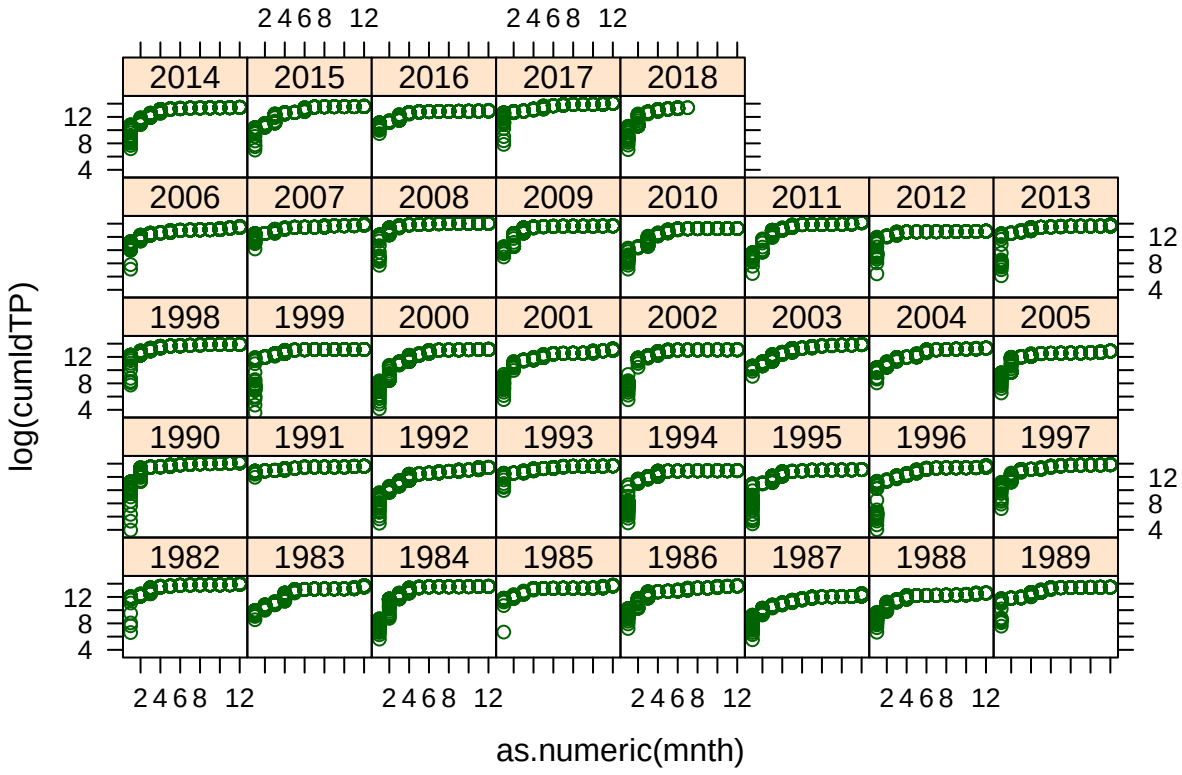

```
xyplot(cumldNOx ~ as.numeric(julian), data=wvl.daily, group=yr)
```



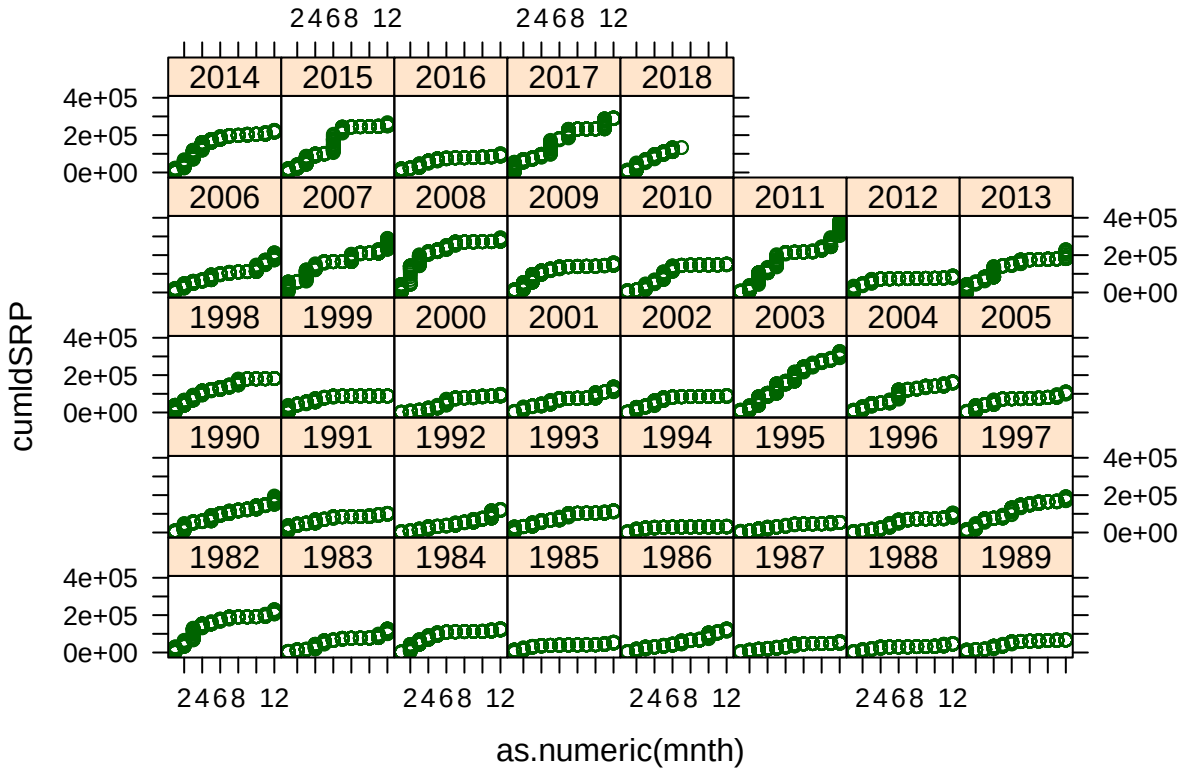

```
xyplot(cumldTP~ as.numeric(mnth)|yr, data=wvl.daily, subset=as.numeric(wvl.daily$yr)>14)
```



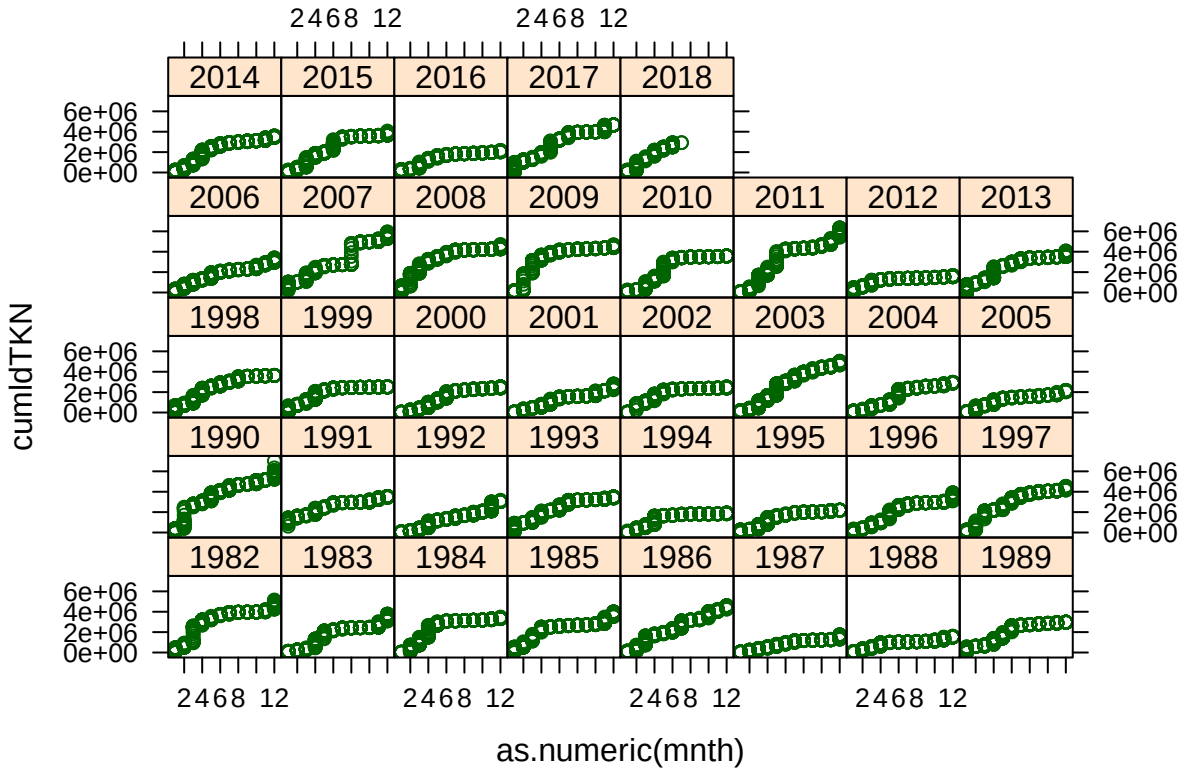

```
xyplot(log(cumldTP)~ as.numeric(mnth)|yr, data=wvl.daily, subset=as.numeric(wvl.daily$yr)>14)
```



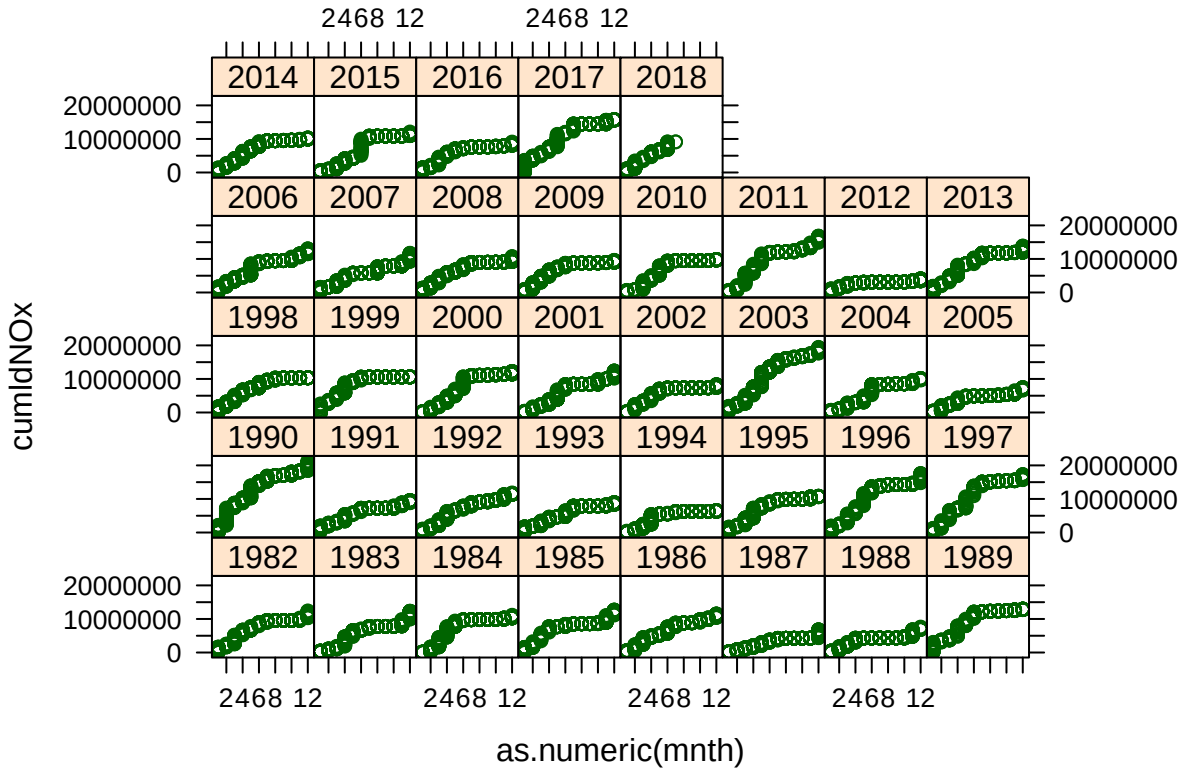

```
xyplot(cumldSRP~ as.numeric(mnth)|yr, data=wvl.daily, subset=as.numeric(wvl.daily$yr)>14)
```



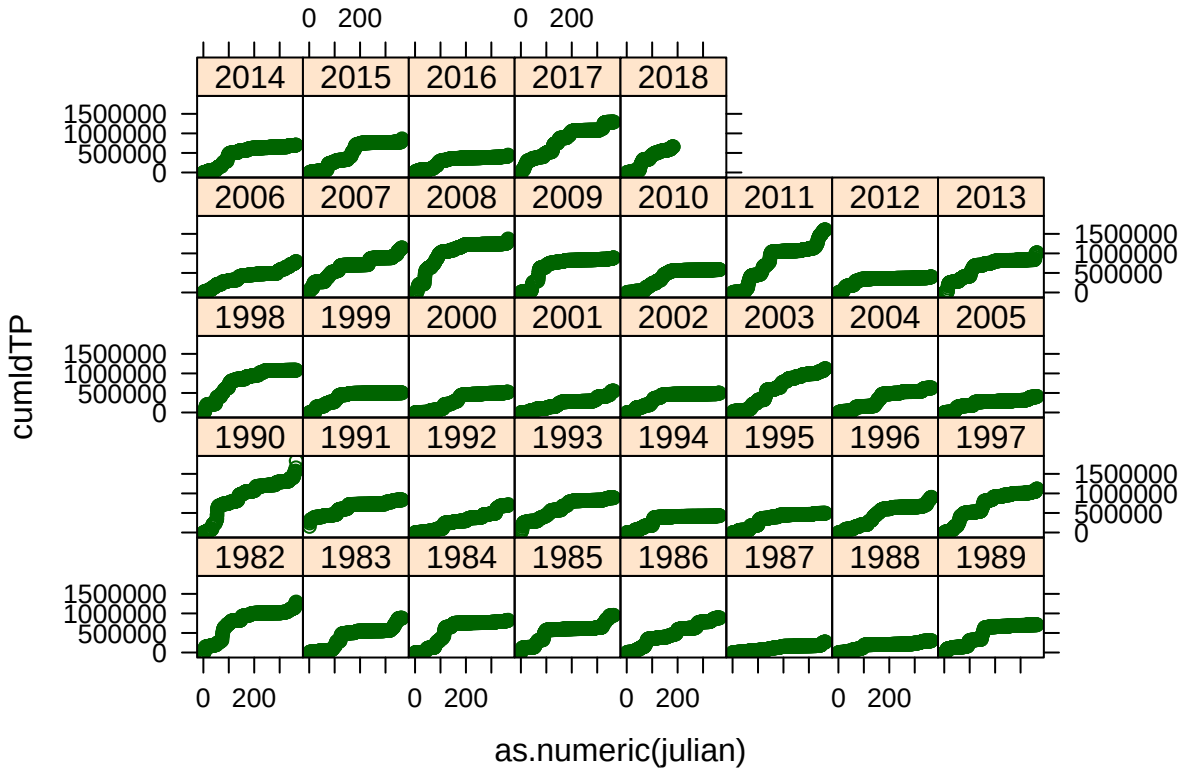

```
xyplot(cumldTKN~ as.numeric(mnth)|yr, data=wvl.daily, subset=as.numeric(wvl.daily$yr)>14)
```



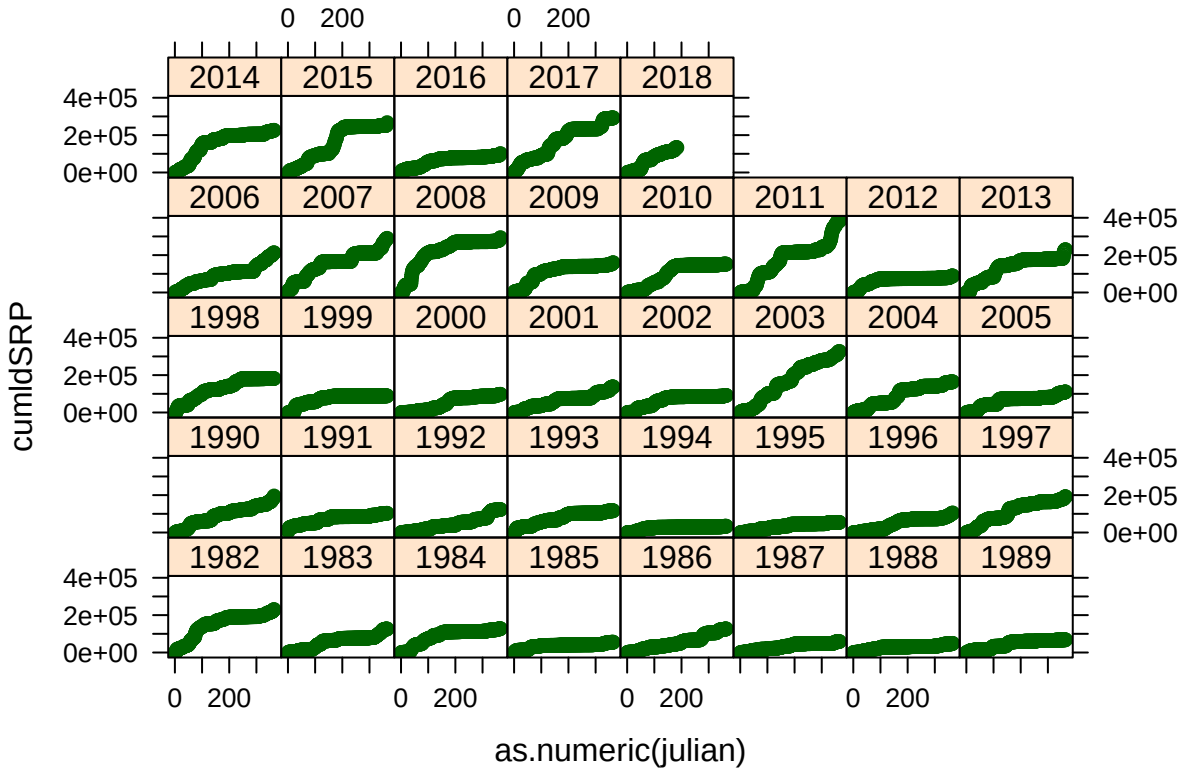

```
xyplot(cumldNOx~ as.numeric(mnth)|yr, data=wvl.daily, subset=as.numeric(wvl.daily$yr)>14)
```



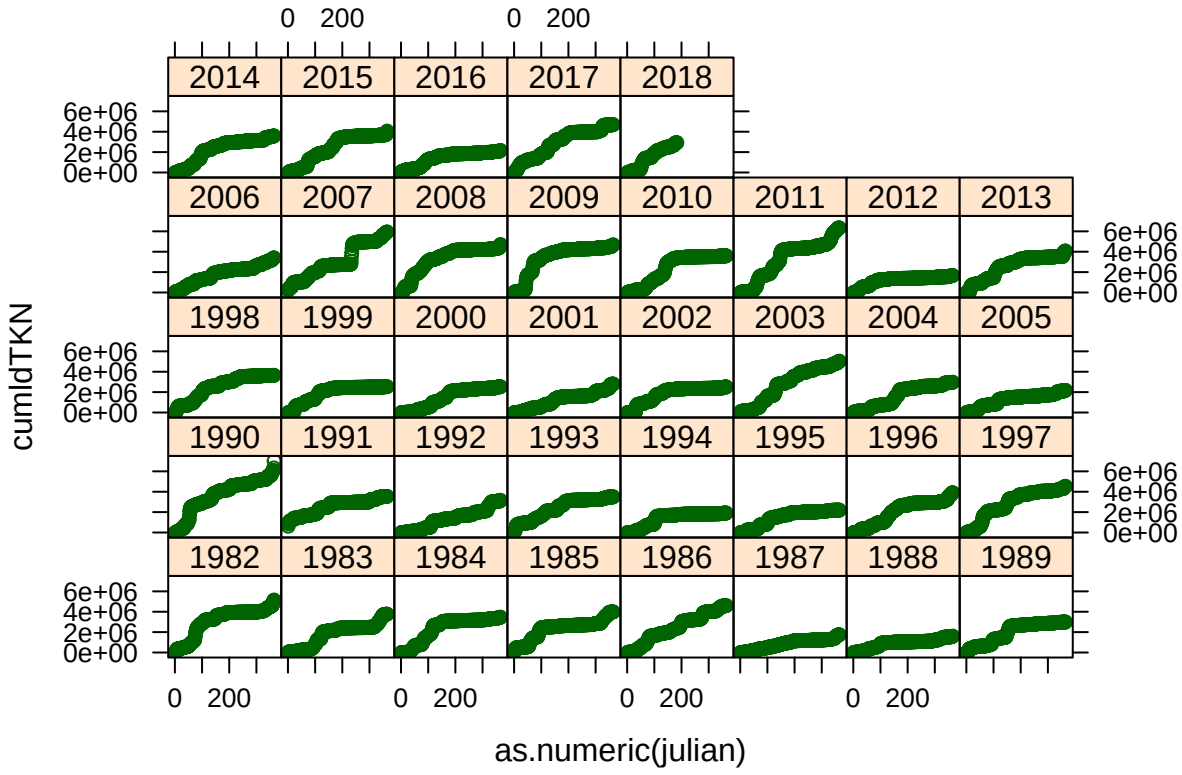

```
xyplot(cumldTP~ as.numeric(julian)|yr, data=wvl.daily, subset=as.numeric(wvl.daily$yr)>14)
```



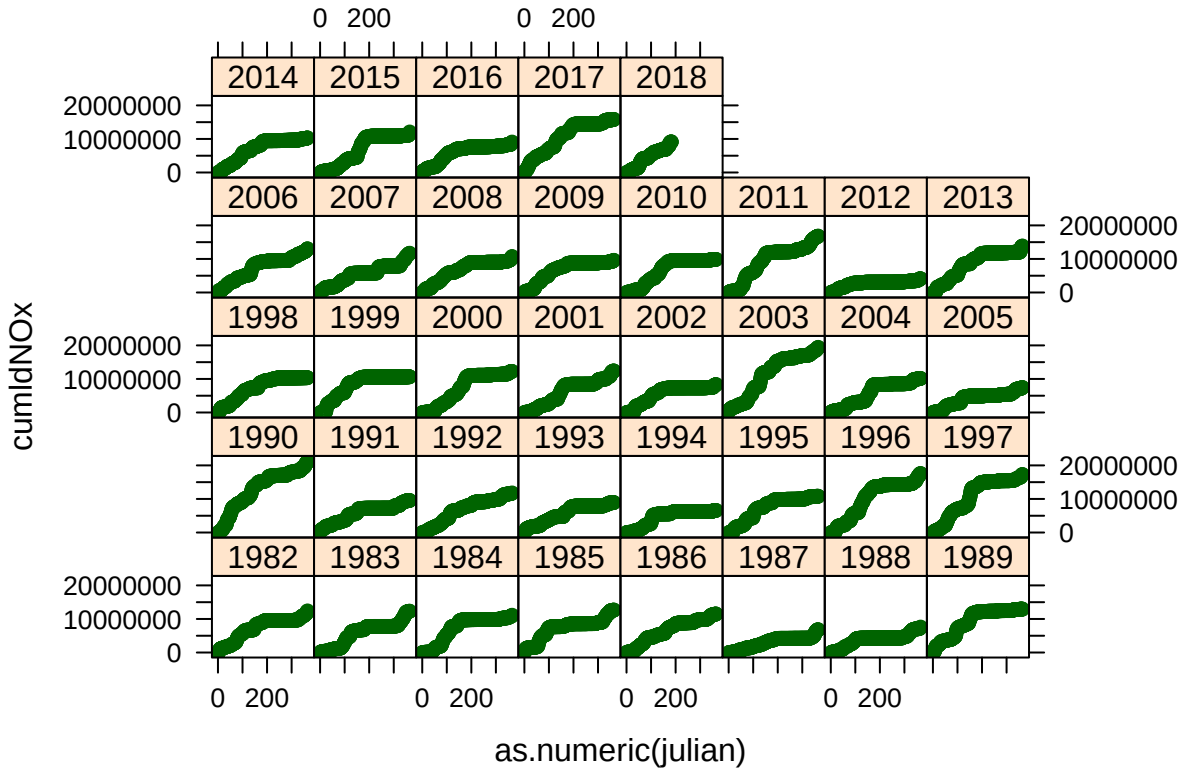

```
xyplot(cumldSRP~ as.numeric(julian)|yr, data=wvl.daily, subset=as.numeric(wvl.daily$yr)>14)
```



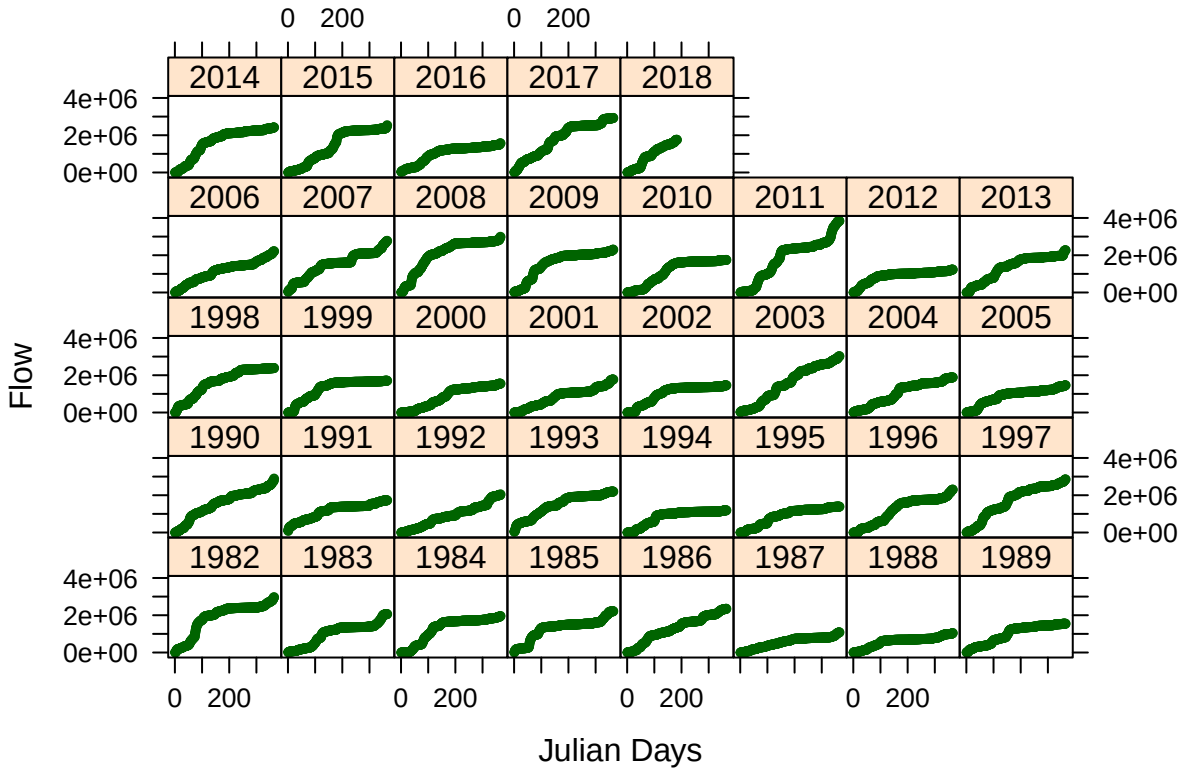

```
xyplot(cumldTKN~ as.numeric(julian)|yr, data=wvl.daily, subset=as.numeric(wvl.daily$yr)>14)
```



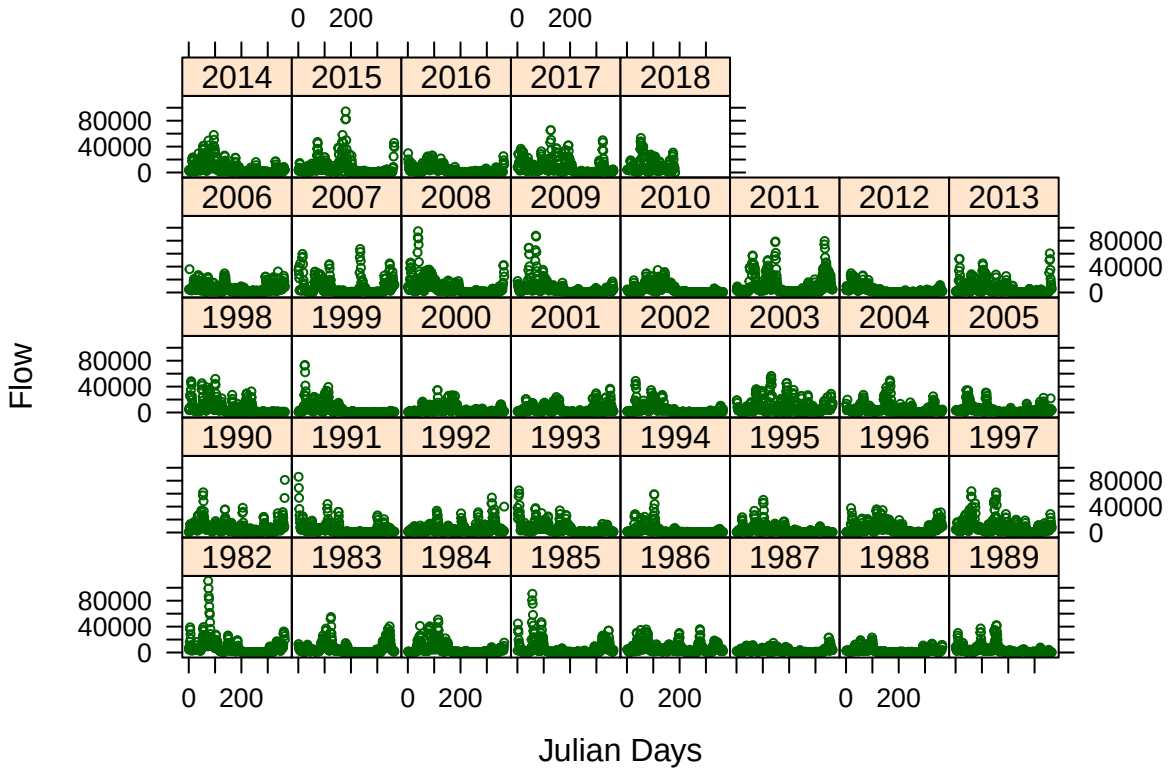

```
xyplot(cumldNOx~ as.numeric(julian)|yr, data=wvl.daily, subset=as.numeric(wvl.daily$yr)>14)
```




```
xyplot(cumldFLW~ as.numeric(julian)|yr, data=wvl.daily, #subset=as.numeric(wvl.daily$yr)>14,
       xlab="Julian Days", ylab="Flow", cex=0.5)
```




```
xyplot(Flow~ as.numeric(julian)|yr, data=wvl.daily, #subset=as.numeric(wvl.daily$yr)>14,
       xlab="Julian Days", ylab="Flow", cex=0.5)
```




```
TPld <-  ts(wvl.daily$TPld, start=c(1982,1), freq=365.25)
write.csv(wvl.daily, file="maumeedaily.csv")
```

12 Week 14: Maps

##Introduction

Have you ever crunched some numbers on data that involved spatial locations? If the answer is no, then boy are you missing out! So much spatial data to analyze and so little time.

Since your time is precious, you know that attempting to create spatial plots in languages like Matlab or applications like Excel can be a tedious, long process. (And GIS softwares are not free, nor open source.) Thankfully there are a number of R packages for processing spatial data. Many of them can handle map.

We will use the Lake Erie HABs monitoring data as an example.

```
eriedata <- read.csv(paste(dataDIR, "ErieData08_15.csv", sep="/"), header=T)
head(eriedata)
```

```
##   Station Depth  DepthC Year    Day    Date Time Latitude Longitude DepthST
## 1     LE1  0.75 surface 2008 26-Jun 6/26/08        41.750   -83.104      NA
## 2     LE2  0.75 surface 2008 26-Jun 6/26/08        41.821   -83.186      NA
## 3     LE3  0.75 surface 2008 26-Jun 6/26/08        41.788   -83.356      NA
## 4     LE4  0.75 surface 2008 26-Jun 6/26/08        41.742   -83.401      NA
## 5     LE5  0.75 surface 2008 26-Jun 6/26/08        41.716   -83.455      NA
```

```
## 6     LE6  0.75 surface 2008 26-Jun 6/26/08        41.733   -83.297      NA
##   Secchi Temp Co0 SpCo0 BeamAtten Tramission chla PC PAR DO au.chl AquaFluor_PC
## 1    3.4   NA  NA    NA        NA         NA   NA NA  NA NA     NA           NA
## 2    4.9   NA  NA    NA        NA         NA   NA NA  NA NA     NA           NA
## 3    3.2   NA  NA    NA        NA         NA   NA NA  NA NA     NA           NA
## 4    2.2   NA  NA    NA        NA         NA   NA NA  NA NA     NA           NA
## 5    0.2   NA  NA    NA        NA         NA   NA NA  NA NA     NA           NA
## 6    1.4   NA  NA    NA        NA         NA   NA NA  NA NA     NA           NA
##   AquaFluor_CHL Turbidity     TP TDP SRP NH4 NO3 SiO2 Cl TSS VSS POC PON DOC
## 1            NA        NA  21.67  NA  NA  NA  NA   NA NA  NA  NA  NA  NA  NA
## 2            NA        NA  22.61  NA  NA  NA  NA   NA NA  NA  NA  NA  NA  NA
## 3            NA        NA  58.16  NA  NA  NA  NA   NA NA  NA  NA  NA  NA  NA
## 4            NA        NA  71.25  NA  NA  NA  NA   NA NA  NA  NA  NA  NA  NA
## 5            NA        NA 245.25  NA  NA  NA  NA   NA NA  NA  NA  NA  NA  NA
## 6            NA        NA  39.45  NA  NA  NA  NA   NA NA  NA  NA  NA  NA  NA
##   CDOM400 CDOMintercept CDOMslope ExChla ExPC  pMC dMC pMC_LC_MS TcyanoCells
## 1      NA            NA        NA   1.54   NA 0.02  NA        NA          NA
## 2      NA            NA        NA   2.15   NA 0.01  NA        NA          NA
## 3      NA            NA        NA   1.72   NA 0.01  NA        NA          NA
## 4      NA            NA        NA   2.22   NA 0.03  NA        NA          NA
## 5      NA            NA        NA  51.20   NA 0.13  NA        NA          NA
## 6      NA            NA        NA   9.60   NA 0.03  NA        NA          NA
##   TcyanoVolume
## 1           NA
## 2           NA
## 3           NA
## 4           NA
## 5           NA
## 6           NA
```

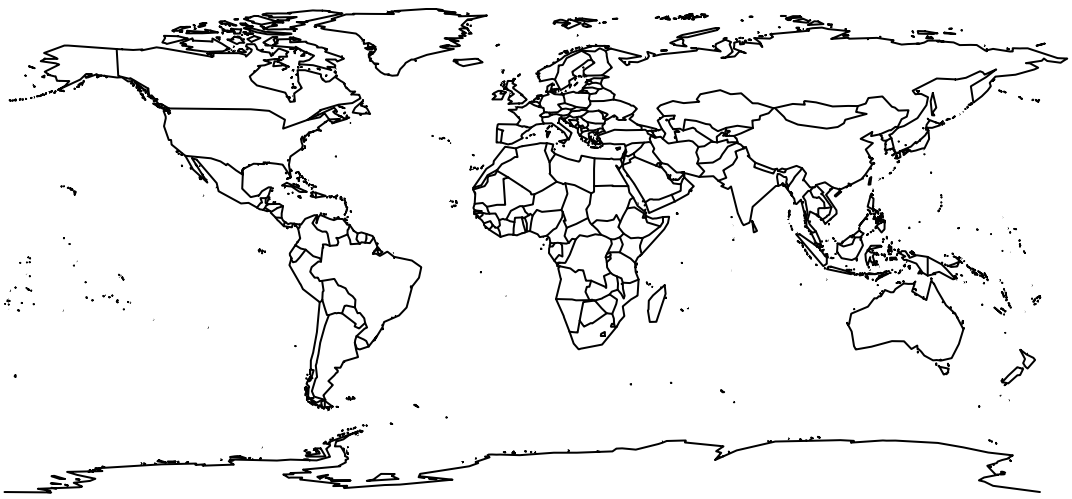
```
    eriedata$Latitude[eriedata$Latitude>90 & !is.na(eriedata$Latitude)] <-
    mean(eriedata$Latitude[eriedata$Station=="WE8" & eriedata$Year=="2013" & eriedata$Latitude<90])

    eriedata$Longitude[eriedata$Longitude < -88  & !is.na(eriedata$Longitude)] <- -83.1940

    eriedata$stdate <- paste(eriedata$Station, eriedata$Date)
    for (i in 1:dim(eriedata)[1]){
        if(is.na(eriedata$Latitude[i])){
            temp <- eriedata$Latitude[eriedata$stdate == eriedata$stdate[i]]
            eriedata$Latitude[i] <- temp[1]
#            print(temp)
        }
    }
    for (i in 1:dim(eriedata)[1]){
        if(is.na(eriedata$Longitude[i])){
            temp <- eriedata$Longitude[eriedata$stdate == eriedata$stdate[i]]
            eriedata$Longitude[i] <- temp[1]
#            print(temp)
        }
    }

    eriedata$Latitude[eriedata$Latitude>48] <-
        mean(eriedata$Latitude[eriedata$Station=="WE8" &
                                   eriedata$Latitude < 48])
```

```
    eriedata$Latitude[eriedata$Latitude < 41.1 &
                          eriedata$Station == "WE2"] <- 41.7622
    eriedata$Latitude[eriedata$Latitude<41.1 &
                          eriedata$Station == "WE7"] <- 41.6749
```

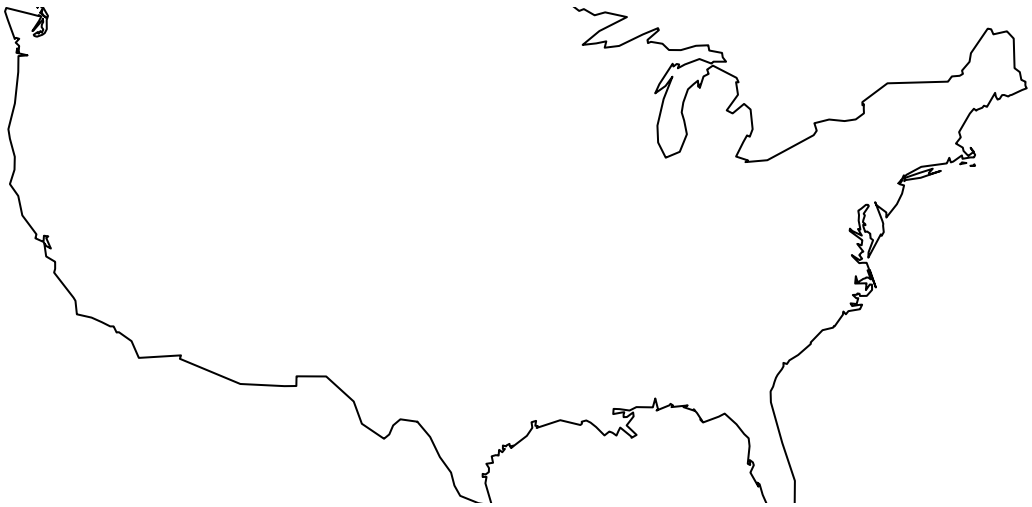
12.1 The maps Package

The oldest package is the `maps` package (along with `maptools` and `mapproj`). These packages allow us to make simple maps and display spatial data onto maps.

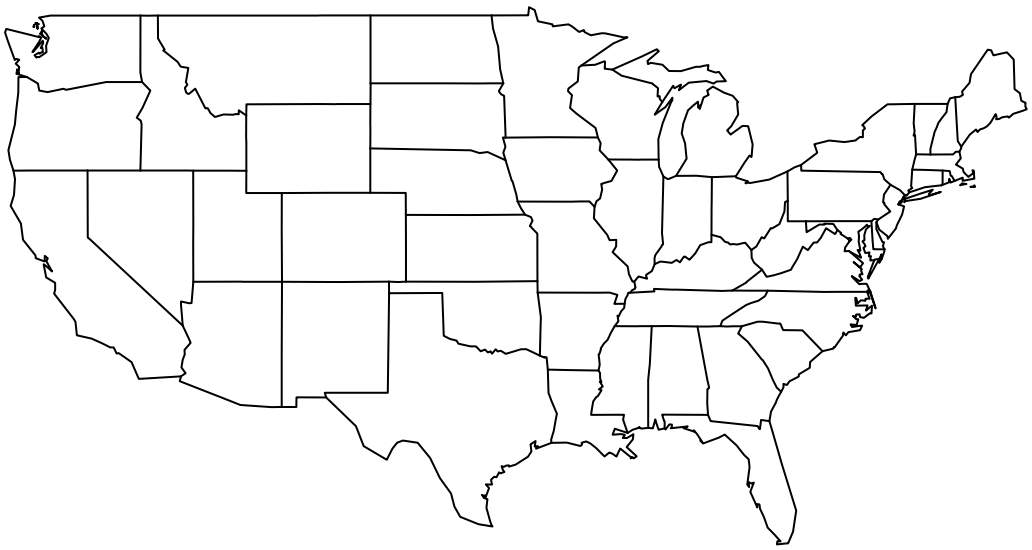
```
maps::map()
```



```
maps::map("usa")
```

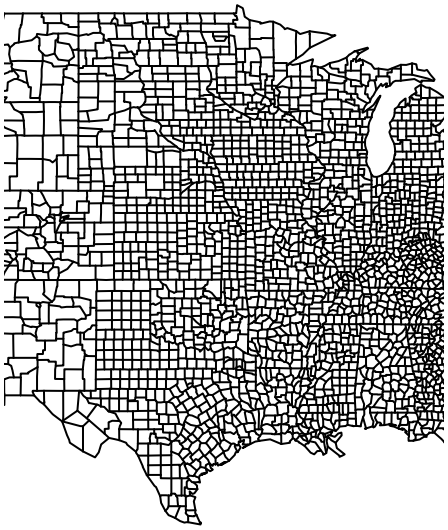


```
maps::map("state")
```



```
maps::map("state", region=c("ohio","michigan"))
```

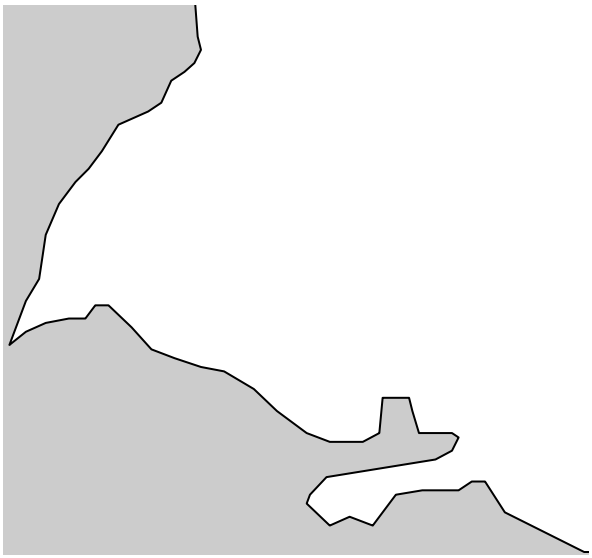
```
maps::map("county")
```



```
maps::map("county", "ohio")
```

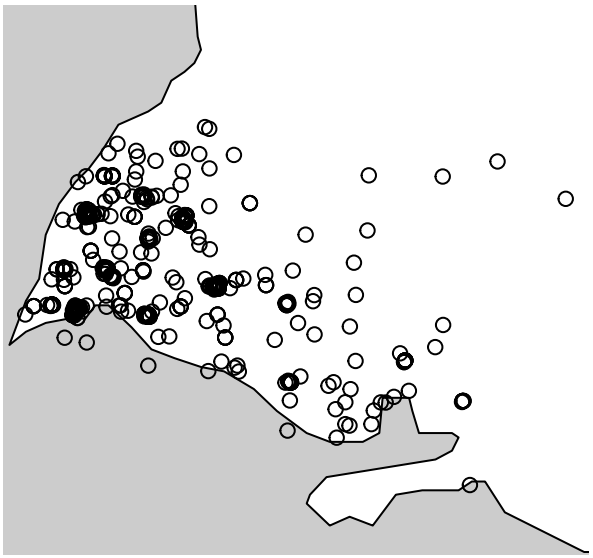
We can plot a specific region by defining the ranges of latitude and longitude. For example, the US portion of the Western Lake Erie:

```
maps::map("usa", fill=TRUE, col="grey80", xlim=c(-83.5,-82.5),
        ylim=c(41.4, 42.1))
```



Data points can be added to the map just like in a normal scatter plot (with `points`):

```
maps::map("usa", fill=TRUE, col="grey80", xlim=c(-83.5,-82.5),
        ylim=c(41.4, 42.1))
points(x=eriedata$Longitude, y=eriedata$Latitude)
```



To show the study location on a world map, we can add the world map to a corner. This operation requires add a map to an existing figure. We need to first generate the map and save it to an object (instead of plotting).

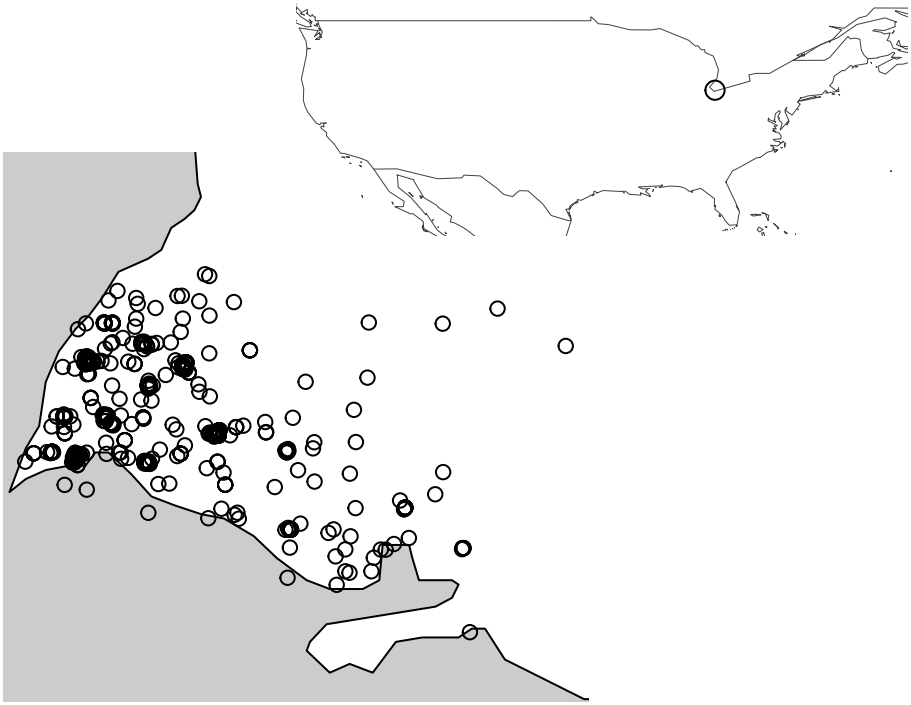
```
maplocs <- maps::map(projection="sp_mercator", wrap=TRUE, lwd=0.1, ## very thin line
                     col="grey", xlim=c(-180, 0),
                     interior=FALSE, orientation=c(90, 180, 0), add=TRUE,
                     plot=FALSE)  ## not plotting
    xrange <- range(maplocs$x, na.rm=TRUE)
    yrange <- range(maplocs$y, na.rm=TRUE)
    aspect <- abs(diff(yrange))/abs(diff(xrange))  ## set the appropriate aspect ratio
    # customised to 6.5 by 4.5 figure size (in inches)
    par(fig=c(0.5, 0.99, 0.99 - 0.5*aspect*4.5/6.5, 0.99),  ### fig sets coordinates of a new figure reg
        mar=rep(0, 4), new=TRUE) ## fig is always used together with new=TRUE
    plot.new() ## add a new plot
    plot.window(xlim=c(1,2.00),
                ylim=c(0.45,1))  ## set up coordinates for a new graphics window
    maps::map(projection="sp_mercator", wrap=TRUE, lwd=0.25, fill=F,
        col=gray(0.25), interior=TRUE, orientation=c(90, 180, 0),
        add=TRUE)
```

We need to add a box to show the study area.

```
symbols(1.7, 0.8, circles = 1, inches=0.05, add=T)
```

```
## Warning in maps::map(projection = "sp_mercator", wrap = TRUE, lwd = 0.25, :
```

```
## projection failed for some data
```



12.2 In Conjunction with ggplot2

This section provides some examples for creating a ggplot map, as well as a choropleth map, in which areas are patterned in proportion to a given variable values being displayed on the map, such as population, life expectancy, or density.

We need R function `map_data()` [in ggplot2] to retrieve the map data. The function `geom_polygon()` [in ggplot2] to create the map

We'll use the `viridis` package to set the color palette of the choropleth map.

Load required packages and set default theme:

Load required packages and set default theme:

```
packages(ggplot2)
packages(dplyr)
packages(viridis)
```

```
## Loading required package: viridis
```

```
## Warning: package 'viridis' was built under R version 4.1.2
```

```
## Loading required package: viridisLite
```

```
## Warning: package 'viridisLite' was built under R version 4.1.2
```

```
##
## Attaching package: 'viridis'
```

```
## The following object is masked from 'package:maps':
##
##     unemp
```

```
theme_set(
  theme_void()
  )
```

The package `viridis` brings to R color scales created by Stéfan van der Walt and Nathaniel Smith for the Python `matplotlib` library.

These color scales are designed to be:

- Colorful, spanning as wide a palette as possible so as to make differences easy to see,
- Perceptually uniform, meaning that values close to each other have similar-appearing colors and values far away from each other have more different-appearing colors, consistently across the range of values,
- Robust to colorblindness, so that the above properties hold true for people with common forms of colorblindness, as well as in grey scale printing, and
- Pretty, oh so pretty

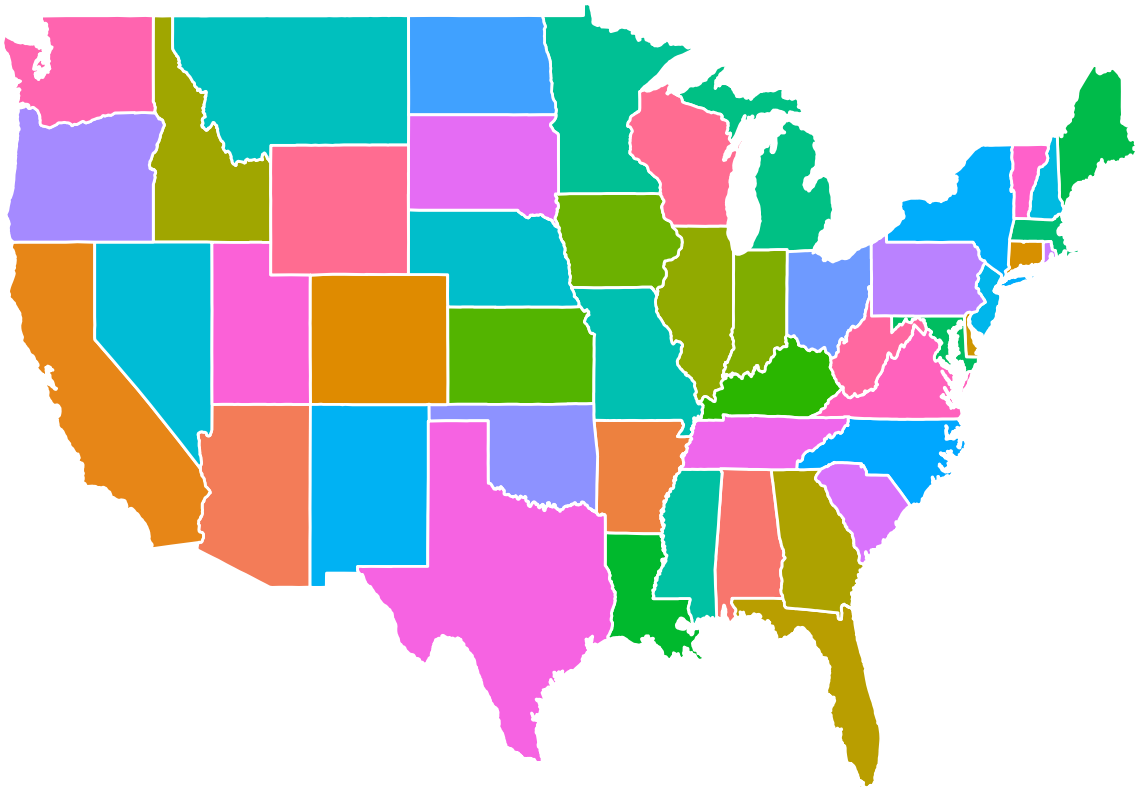
12.2.1 Creating a Simple Map

```
##Retrieve the world map data:

world_map <- map_data("world")
print (p <- ggplot(world_map, aes(x = long, y = lat, group = group)) +
  geom_polygon(fill="lightgray", colour = "white"))
```

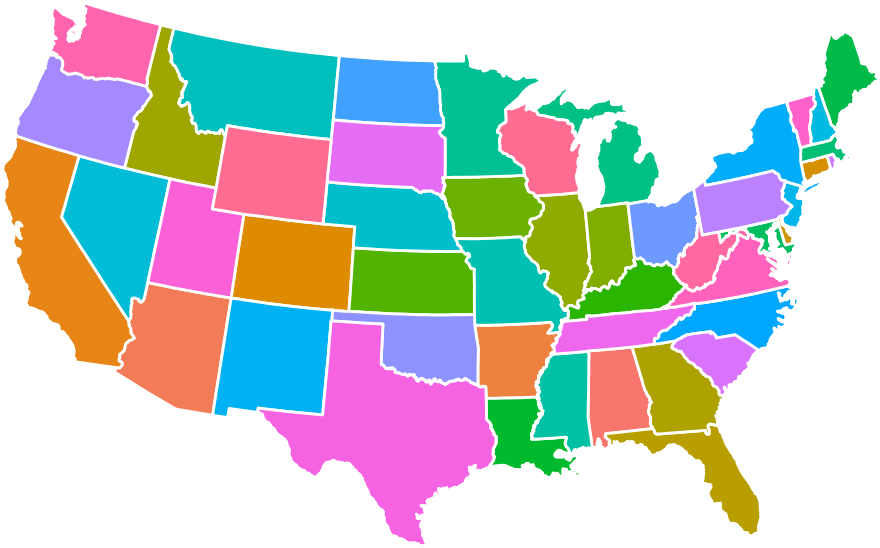
```
states <- map_data("state")
print(p <- ggplot(states, aes(x = long, y = lat, group = group)) +
  geom_polygon(fill="lightgray", colour = "white"))
```

```
print(p <- ggplot(states, aes(x = long, y = lat, group = group)) +
  geom_polygon(aes(fill=region), colour = "white")+guides(fill=F))
```

```
## Warning: `guides(<scale> = FALSE)` is deprecated. Please use `guides(<scale> =
## "none")` instead.
```



```
print(p + coord_map(projection="albers", lat0=39, lat=45))
```



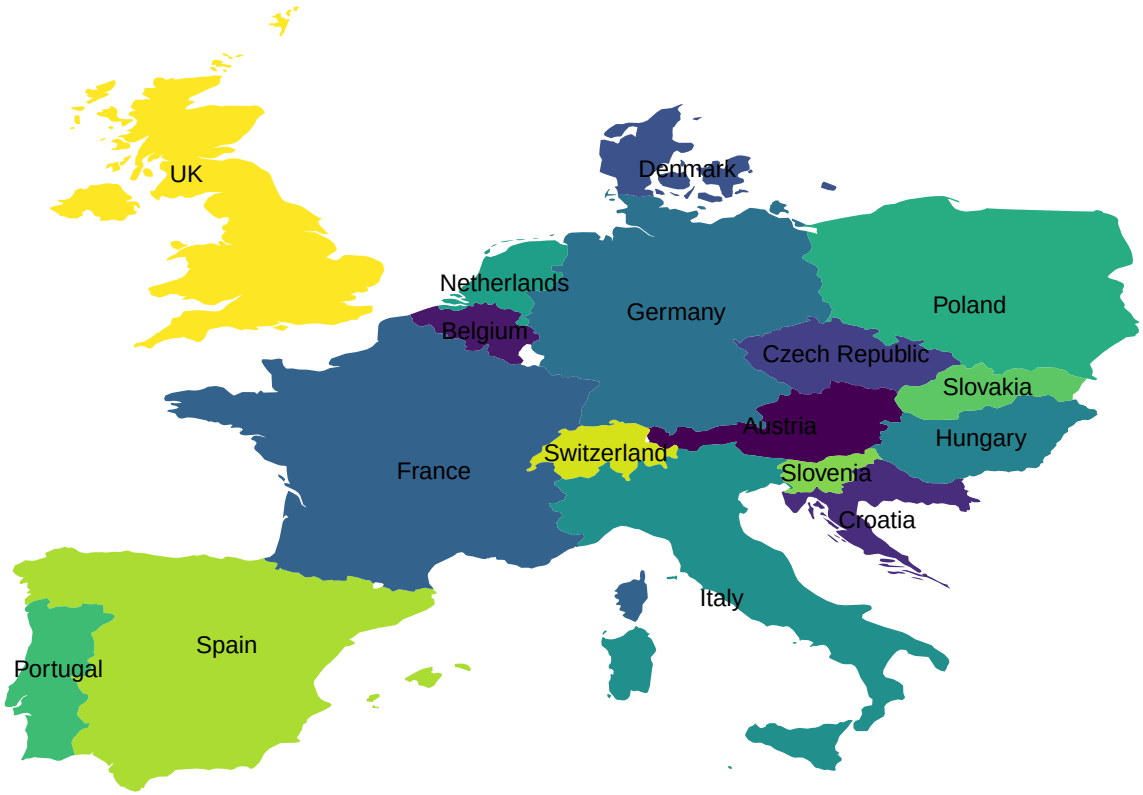
```
counties <- map_data("county")
print(p <- ggplot(counties, aes(x = long, y = lat, group = group)) +
  geom_polygon(fill="lightgray", colour = "white"))
```

Map of specific region

```
some.eu.countries <- c(
  "Portugal", "Spain", "France", "Switzerland", "Germany",
  "Austria", "Belgium", "UK", "Netherlands",
  "Denmark", "Poland", "Italy",
  "Croatia", "Slovenia", "Hungary", "Slovakia",
  "Czech republic"
)
# Retrievethe map data
some.eu.maps <- map_data("world", region = some.eu.countries)

# Compute the centroid as the mean longitude and lattitude
# Used as label coordinate for country's names
region.lab.data <- some.eu.maps %>%
  group_by(region) %>%
  summarise(long = mean(long), lat = mean(lat))
```

Visualize

```
ggplot(some.eu.maps, aes(x = long, y = lat)) +
  geom_polygon(aes( group = group, fill = region))+
  geom_text(aes(label = region), data = region.lab.data,  size = 3, hjust = 0.5)+
  scale_fill_viridis_d()+
  theme_void()+
  theme(legend.position = "none")
```



Make a choropleth Map World map colored by life expectancy

Here, we'll create world map colored according to the value of life expectancy at birth in 2015. The data is retrieved from the WHO (World Health Organozation) data base using the WHO R package.

Retrieve life expectancy data and prepare the data:

```
# packages("~/Downloads/WHO_0.2.1.tar.gz", repos=NULL)
# library("dplyr")
# life.exp <- raster::getData("WHOSIS_000001")             # Retrieve the data
# life.exp <- life.exp %>%
#   filter(year == 2015 & sex == "Both sexes") %>%  # Keep data for 2015 and for both sex
#   select(country, value) %>%                      # Select the two columns of interest
#   rename(region = country, lifeExp = value) %>%   # Rename columns
#   # Replace "United States of America" by USA in the region column
#   mutate(
#     region = ifelse(region == "United States of America", "USA", region)
#     )
```

Merge map and life expectancy data:

```
# world_map <- map_data("world")
# life.exp.map <- left_join(life.exp, world_map, by = "region")
```

Create the choropleth map. Note that, data are missing for some region in the map below: Use the function `geom_polygon()`:

```
# ggplot(life.exp.map, aes(long, lat, group = group))+
#   geom_polygon(aes(fill = lifeExp ), color = "white")+
#   scale_fill_viridis_c(option = "C")
```

Or use the function geom_map():

```
# ggplot(life.exp.map, aes(map_id = region, fill = lifeExp))+
#   geom_map(map = life.exp.map,  color = "white")+
#   expand_limits(x = life.exp.map$long, y = life.exp.map$lat)+
#   scale_fill_viridis_c(option = "C")
```

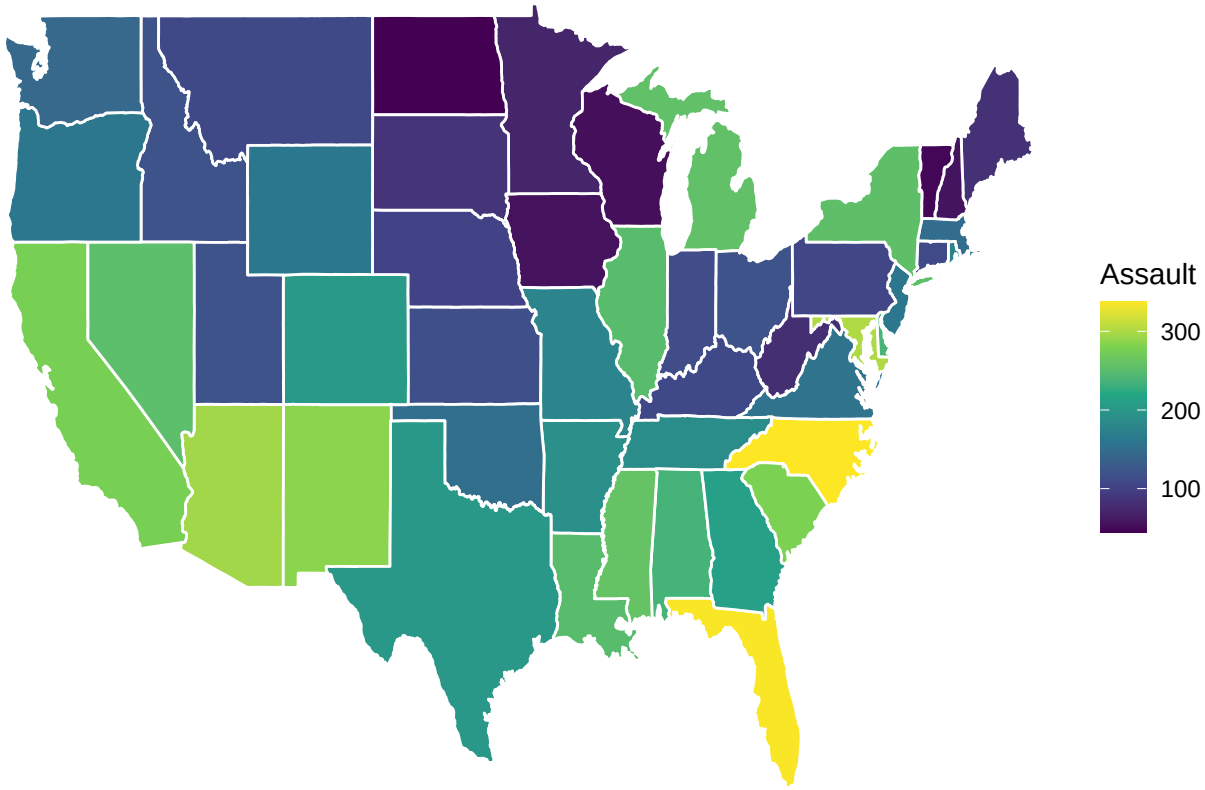
12.2.2 US map colored by violent crime rates

Demo data set: USArrests (Violent Crime Rates by US State, in 1973).

```
# Prepare the USArrests data
arrests <- USArrests
arrests$region <- tolower(rownames(USArrests))
head(arrests)
```

```
##            Murder Assault UrbanPop Rape     region
## Alabama      13.2     236       58 21.2    alabama
## Alaska       10.0     263       48 44.5     alaska
## Arizona       8.1     294       80 31.0    arizona
## Arkansas      8.8     190       50 19.5   arkansas
## California    9.0     276       91 40.6 california
## Colorado      7.9     204       78 38.7   colorado
```

```
##            Murder Assault UrbanPop Rape     region
## Alabama      13.2     236       58 21.2    alabama
## Alaska       10.0     263       48 44.5     alaska
## Arizona       8.1     294       80 31.0    arizona
## Arkansas      8.8     190       50 19.5   arkansas
## California    9.0     276       91 40.6 california
## Colorado      7.9     204       78 38.7   colorado
# Retrieve the states map data and merge with crime data
states_map <- map_data("state")
arrests_map <- left_join(states_map, arrests, by = "region")

# Create the map
ggplot(arrests_map, aes(long, lat, group = group))+
  geom_polygon(aes(fill = Assault), color = "white")+
  scale_fill_viridis_c(option = "D")
```



12.3 The ggmap package

(Note: Google changed its API requirements in July 2018, and `ggmap` users are now required to provide an API key and enable billing. The billing enablement especially is a bit of a downer, although you can use the free tier without incurring charges. As a result of the API change, `ggmap` is out of date. I keep the following code here just in case the issues is resolved in a later update.)

The old map tools in R are very difficult to use and the databases are also old. Fortunately, there are a number of new R libraries being created to make spatial data visualization a more enjoyable endeavor. Of these new options, one useful package is `ggmap`.

The fastest way to get going is with the `qmap` class, which stands for "quick map plot." Play around with the different types of parameter calls to render various plot types.

Some examples to start:

```
# packages(ggmap)
# qmap(location = "toledo, oh")
# qmap(location = "toledo, oh", zoom = 14)
# qmap(location="the university of toledo, oh", zoom=12)
# ## check ut lat/lon
# qmap(location=c(-83.6126, 41.6622), zoom=14)
# qmap(location=c(-83.6126, 41.66), zoom=15)
#
# qmap(location = "lake erie", zoom = 7)
# qmap(location = "lake erie", zoom = 7, maptype="toner")
```

```
# qmap(location = "lake erie", zoom = 7, maptype="watercolor")
# qmap(location = "lake erie", zoom = 7, maptype="s")
#
# qmap(location=c(-83.10, 41.75), zoom=10)
```

Let's now use the HABs monitoring data from NOAA-GLERL.

```
# wle <- get_map(location=c(-83.10, 41.75), zoom=10)
# p <- ggmap(wle)
# ##pdf(file=paste(plotDIR, "HABsamples3.pdf", sep="/"), height=5, width=5)
# p + geom_point(data=eriedata, aes(x=Longitude, y=Latitude), col="red")
# ##dev.off()
```

Presenting Results Using `ggmap` we can present results over a spatial scale graphically. For example, we may want to present the mean TP concentration spatially.

```
# moltern.erie <- reshape2::melt(eriedata[,c("Station","Latitude","Longitude","TP")], id.vars = c("Stat
# erieTP <- dcast(moltern.erie, Station~variable, fun=mean, na.rm=T)
# head(erieTP)
#
# overlay <- stat_density2d(aes(x=Longitude, y=Latitude, fill=..level.., alpha=..level..),
#                     size=2, bins=8, data=erieTP, geom="polygon")
# ##pdf(file=paste(plotDIR, "ErieTPwc.pdf", sep="/"), height=4, width=4)
# p + overlay
```