**Syllabus for [R] workshop** 22 & 23 March 2008 in 310 Baker Instructor: Jesse Brunner

- R is a very powerful, flexible "language and environment for statistical computing and graphics"
- R is a free, open-source implementation of the [S] language; *most* of what works with S-plus will work with [R]
- R is used by an increasing number of statisticians (and scientists) and is thus growing and improving constantly, as is the documentation
- R is memory hungry and can be slow for large simulations; it is not a replacement for C or Fortran. Nor does it replace Maple or Matlab if you need to do symbolic calculations (e.g., integrals)

**Getting [R], packages, and finding help**

http://www.r-project.org/ is the project home page, with announcements, links, etc.
http://cran.r-project.org/ has current versions of [R] for Linux, Mac, and Windows
http://finzi.psych.upenn.edu/search.html is a very useful searchable database of help files, manuals, and mailing lists

```
> help(function_name) or ?function_name
> apropos(bit_of_function_name)
```

**[R] as a fancy calculator**

addition: `1 + 2` subtraction: `3 - 1` multiplication: `2 * 3` division: `4 / 5`
powers: `4 ^ 2` exponent: `exp(2)` $\log_e$: `log(2)` $\log_{10}$: `log10(2)`

Assignments: `a <- 3/2` or `a = 3/2`
Series: `x <- 1:10` or `x <- seq(from=1, to=10, by=1)`
or `x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)`
Calculations on vectors: `3*x`, `x^2`, etc.
Summary information: `sum(x)`, `max(x)`, `min(x)`, `range(x)`, `mean(x)`, `median(x)`, `var(x)`, `sd(x)`, `summary(x)`, `length(x)`, `mode(x)`

**Working with data in [R]**

Subsets:

```
> y <- sqrt(x)
> y
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068
 [5] 2.449490 2.645751 2.828427 3.000000 3.162278
> y[2]
[1] 1.414214
> y[-2]
[1] 1.000000 1.732051 2.000000 2.236068 2.449490
[5] 2.645751 2.828427 3.000000 3.162278
> y[2:5] or y[c(2,3,4,5)]
[1] 1.414214 1.732051 2.000000 2.236068
> y[y > 2]
[1] 2.236068 2.449490 2.645751 2.828427 3.000000 3.162278
> y[y > 2 & y <= 3]
[1] 2.236068 2.449490 2.645751 2.828427 3.000000
```

| Operator | Functionality |
|---|---|
| & | And |
| \| | Or |
| ! | Not |
| == | Equal to |
| != | Not equal to |
| < | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal |

**Data Structures**: `vector`, `matrix`, `array`, `data.frame`, `list`

```
> matrix(1:50, ncol = 5)  # See what it looks like with byrow = TRUE
> array(1:50, dim = c(20, 5) ) # same thing
> array(1:50, dim = c(5, 5, 4) ) # Not the same
```

Working with matrices:

```
> z <- cbind(x, y) # Contrast with rbind(x, y)
> z[1:3, 2]  or  z[1:3, "y"]
 [1] 1.000000 1.414214 1.732051
> z[1:3, 1:2]  or  z[1:3, ]  or  z[1:3, c("x", "y")]
     x        y
[1,] 1 1.000000
[2,] 2 1.414214
[3,] 3 1.732051
```

Working with dataframes:

```
> df <- as.data.frame(z)
> df
    x        y
1   1 1.000000
2   2 1.414214
3   3 1.732051
4   4 2.000000
5   5 2.236068
6   6 2.449490
7   7 2.645751
8   8 2.828427
9   9 3.000000
10 10 3.162278
```

Some commands to get useful information about objects:

```
> str(df)
'data.frame':     10 obs. of  2 variables:
 $ x: num  1 2 3 4 5 6 7 8 9 10
 $ y: num  1.00 1.41 1.73 2.00 2.24 ...
> dim(df)
[1] 10  2
> attributes(df)
$names
[1] "x" "y"
$row.names
 [1]  1  2  3  4  5  6  7  8  9 10
$class
[1] "data.frame"
```

```
> df$y or df[ , 2]
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
 [8] 2.828427 3.000000 3.162278
> df$y[1:4] or  df[1:4, 2]
[1] 1.000000 1.414214 1.732051 2.000000

> df$sum <- df$x + df$y  # You can implicitly create new columns in a dataframe
> df$sum
 [1]  2.000000  3.414214  4.732051  6.000000  7.236068  8.449490
 [7]  9.645751 10.828427 12.000000 13.162278
> df$sum = 0 #Be very careful with assignments

> which(df$y > 2)  #Gives the index of those entries that meet the criteria
[1]  5  6  7  8  9 10

> df$x %% 2   #This is x mod 2, or "is there a remainder?"
 [1] 1 0 1 0 1 0 1 0 1 0
> which( (df$x %% 2) == 1)  #Index of those with a remainder (i.e., the odds)
[1] 1 3 5 7 9
> df$odd = "Even"  #New column with the word "Even" at each entry
> df$odd[ which( (df$x %% 2) == 1 ) ] = "Odd"  #Changing the odd entries to "Odd"
> df$odd
 [1] "Odd"  "Even" "Odd"  "Even" "Odd"  "Even" "Odd"  "Even" "Odd"  "Even"
```

One more data type: factors

```
> str(df)
'data.frame':     10 obs. of  4 variables:
 $ x  : num  1 2 3 4 5 6 7 8 9 10
 $ y  : num  1.00 1.41 1.73 2.00 2.24 ...
 $ sum: num  0 0 0 0 0 0 0 0 0 0
 $ odd: chr  "Odd" "Even" "Odd" "Even" ...
```

These are just words to [R]. It works much better with factors as we will see later

```
> df$odd = factor(df$odd) #Let's change these words to levels of a factor
> str(df)
'data.frame':     10 obs. of  4 variables:
 $ x  : num  1 2 3 4 5 6 7 8 9 10
 $ y  : num  1.00 1.41 1.73 2.00 2.24 ...
 $ sum: num  0 0 0 0 0 0 0 0 0 0
 $ odd: Factor w/ 2 levels "Even","Odd": 2 1 2 1 2 1 2 1 2 1
> levels(df$odd)
[1] "Even" "Odd"   #Note: By default [R] orders levels alphabetically
```

Working with lists:

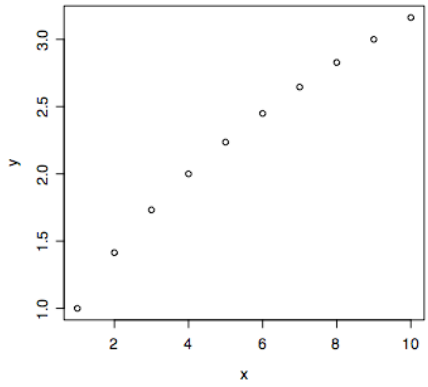
```
> lst <- list(x[1:4], y[1:10], c("green", "yellow"))
> lst
[[1]]
[1] 1 2 3 4

[[2]]
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
 [8] 2.828427 3.000000 3.162278

[[3]]
[1] "green"  "yellow"
> lst[[2]]   #Note the double brackets
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
 [8] 2.828427 3.000000 3.162278
> lst[[3]] [1]   #Indexing is a little different with lists
[1] "green"
```

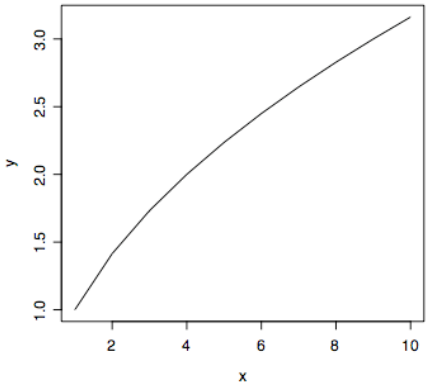
**Working in the [R] environment**

```
> ls()   #Find all of the objects in your environment
[1] "df"  "lst" "x"   "y"   "z"
> z.1 = z + 1; z.2 = z + 2   #Just making a couple more matrices
> ls()        #Here they are
[1] "df"  "lst" "x"   "y"   "z"   "z.1" "z.2"
> rm(x, y, z) #Removing  three objects
> ls()
[1] "df"  "lst" "z.1" "z.2"
> rm(list = ls(pat = "z")) #Removing those with a "z" in the name… Fancy!
> ls()
[1] "df"  "lst"

> search()   #Shows loaded packages [as well as attached data frames… not recommended]
 [1] ".GlobalEnv"        "tools:RGUI"        "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

**The essentials of plotting data in [R]**

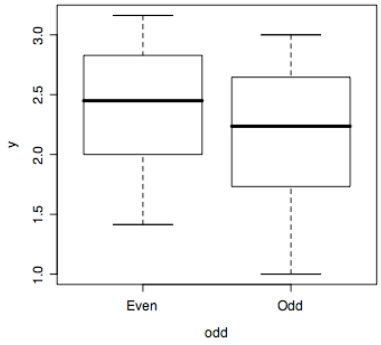
```
> plot(y ~ x, data = df) or plot(df$y ~ df$x)
```




```
> plot(y ~ x, data = df, type = "l")
```
# "l" for line. Try "b" for both, "h" for histogram-like vertical lines, "s" for stair steps, and "n" for no plotting

```
> plot(y ~ odd, data = df)
```
#the plot function is generic and can deal with most data types, like factors such as "odd"

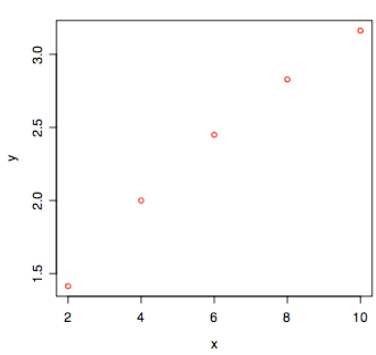


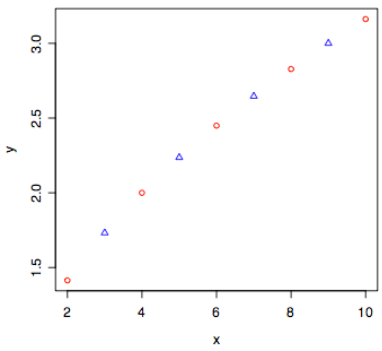

Let's get fancy and plot odd and even "observations" separately

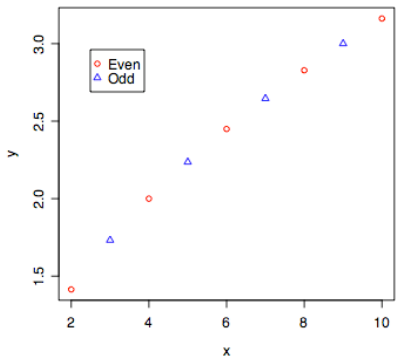
```
> plot(y ~ x, data = df[df$odd == "Even", ], col = "red", pch = 1)
```
`col` is the color. `pch` is the plotting character. See the help for `points` for more.

```
> points(y ~ x, data = df[df$odd == "Odd", ], col = "blue", pch = 2)
```
points adds points to a plot without erasing it

```
> legend(locator(1), legend = (c("Even", "Odd")), pch = 1:2, col = c("red",
  "blue"))
```
Legend requires a vector of the names of the series, and optionally point types, colors, line types, etc. `locator(1)` tells it to put the legend wherever you click once.



**Getting data into and out of [R]**

One way:

```
> x <- scan()   #Note: This only works with numerical values
1: 1
2: 2
3: 3
4: 4
5: 5
6:
Read 5 items
```

```
> edit(x) or fix(x)  #What happens if you edit a dataframe?
```

Easiest way to get data in and out: Use a spreadsheet, save in a comma separated values format (name.csv), and then read in the data.

Hints: 1) Keep the table clean (i.e., one table per sheet, no extraneous rows or columns) [It's a good idea to check for extra rows that might have been added when importing]
2) Do not use spaces in titles or variable names
3) Do not start variable names with a numerals or symbols

```
> DF <- read.csv(file = "SampleData.csv", header = TRUE)
> str(DF) #Check everything out to make sure it looks like you expect
> write.csv(DF, file = "DataFrame.csv") #Export your dataframe as a csv file
```

Note: *Many* formats are supported to varying degrees—see the "R Data Import/Export" manual under "Documentation" on the CRAN site—but simple file types are best. If you use tab-delimited files, for instance, the commands are:

```
> read.table(file="TabDelData.txt", sep="\t", header = TRUE)
> write(DF, file = "DF-tabdel.txt", sep = "\t")
```

Saving your workspace, certain objects, and your command history:

```
> save.image(file = "WorkspaceName.Rdata") #Saves everything in your workspace,
  which you can load again using:
> load(file = "WorkspaceName.Rdata")

> save(ObjName, file = "ObjName.Rdata") #Saves your object in an [R] format

> savehistory(file = "hist1.Rhistory")
> loadhistory(file = "hist1.Rhistory")
```

NOTE: You can save and load objects and workspace images from the menus, too.

Hint: Use a separate text editor file (e.g., in Word) to document track your *useful* commands (and not all of the ones that didn't work), your output, and your personal notes and comments

Figuring out and changing the working directory:

```
> getwd()
[1] "/Users/brunner"
> setwd("/Users/brunner/Desktop/")
```

It's usually easier to use the menus

**Exercise**: Size and body condition of salamander larvae

1) Go to http://www.esf.edu/efb/brunner/Rworkshop.htm and download the SalData.csv dataset. Import it into [R]
2) Find summary statistics (mean, sd, max, min) for Mass and SVL for male and female salamanders. Try `hist()` and `stem()`
3) Test whether males are larger (I would suggest a t-test).
4) I'm interested in body condition. Try plotting Mass against SVL for the two sexes.
5) Create a new column for the body condition index (BCI) Mass divided by SVL. Test whether BCI varies by sex.

**Basics of regression and ANOVA:**

First let's regess Mass on SVL.

```
> lm.SVL <- lm(Mass ~ SVL, data = Sal)
> lm.SVL

Call:
lm(formula = Mass ~ SVL, data = Sal)

Coefficients:
(Intercept)          SVL
   -13.9706       0.4012
```

OK, that wasn't very helpful... `lm.SVL` is a "fitted" object. Typing its name just returns bare bones information about it. We can use `summary()` to get useful, formatted information from the object/

```
> summary(lm.SVL)
Call:
lm(formula = Mass ~ SVL, data = Sal)

Residuals:
    Min      1Q  Median      3Q     Max
-4.0095 -0.7955  0.2059  0.6989  9.0823

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -13.97061    0.76915  -18.16   <2e-16 ***
SVL           0.40118    0.01337   30.00   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.52 on 194 degrees of freedom
Multiple R-squared: 0.8226,   Adjusted R-squared: 0.8217
F-statistic: 899.8 on 1 and 194 DF,  p-value: < 2.2e-16
```

Curious what all is *in* the object `lm.SVL`? Try using the `names()` function on it. You can access the different parts of this just like you would the bits of a list (e.g., `lm.SVL$coefficients`).

We may also want to extract particular bits from this fitted object, e.g.:

```
> logLik(lm.SVL)
'log Lik.' -359.1460 (df=3)
```

Or we can use the `predict()` function to generate predictions for new data

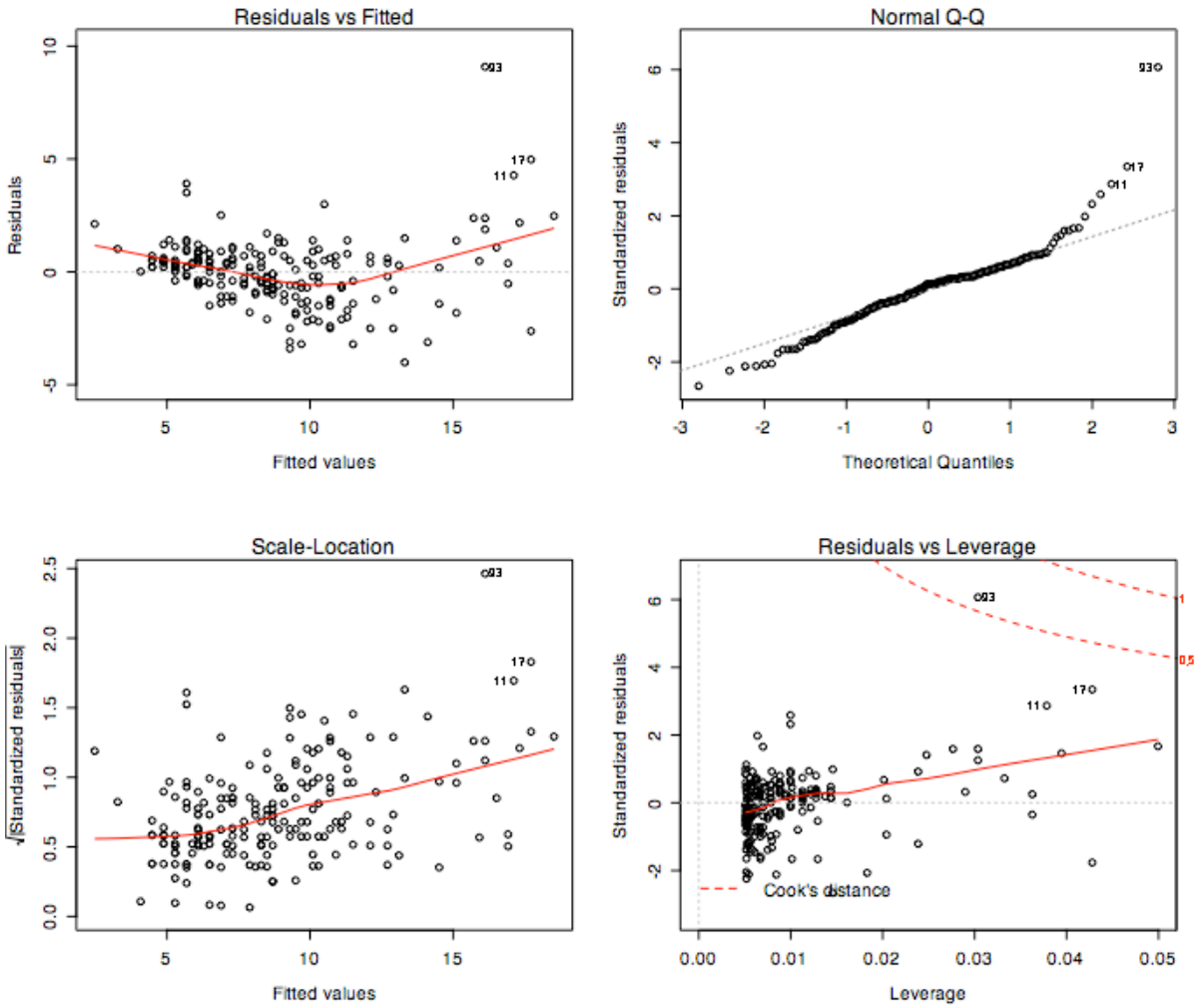
```
> NewSVL <- data.frame(SVL = c(10, 20, 40, 80, 160))
> predict(lm.SVL, NewSVL)
        1         2         3         4         5
-9.958836 -5.947061  2.076487 18.123585 50.217780
```

Potentially useful functions for `lm` objects:

```
print()
summary()
df.residual()
coef()
residuals()
deviance()
fitted()
logLik()
AIC()
```

One other handy feature built into `lm` objects is the ability to generate some very useful diagnostic plots.

```
> par(mfrow = c(2,2)) # This tells [R] to divide the plotting window into two rows and two columns so it will accept all four graphs at once. Otherwise it plots them one at a time.
> plot(lm.SVL)
```




As you can see, [R] has a lot of fancy graphics already built-in.
Try this one, too:

```
> termplot(lm.SVL, se = T, partial.resid = T) #We're turning on the standard error and partial residuals; their default if off (False). Note that you can use "T" for "True" and "F" for "False"
```

You may wonder how `plot(lm.SVL)` knew to plot four diagnostic plots instead of just x ~ y. Well, it turns out that `plot()` has different methods for different objects (really classes) or types of data. `plot()` knew that we gave it a `lm` fitted object so it used `plot.lm()`, which has different methods than, say, `plot.ts()` for time series. Many function in [R] are generic like this. For instance, we were really using `summary.lm()` and `predict.lm()` above. It is sometimes useful to get help on the particular method used (e.g., `help(plot.lm)` ). You can use the `methods()` function to find all of the methods available for a generic function.

What about a typical ANOVA?
First, let's create some size classes:

```
> Sal$SizeClass <- cut(Sal$SVL, breaks = c(0, 53, 65, 81), labels = c("Sm",
  "Med", "Lg")) #Cut divides a continuous variable according to breaks. Don't forget to
  specify the lower end (here zero).
> summary(Sal$SizeClass)

Sm Med  Lg
 77  92  27
```

We first fit the `lm` object, and then we call `anova()` on it[ξ].

```
> anova( lm(Mass ~ Sex * SizeClass, data = Sal) )

Analysis of Variance Table
Response: Mass
               Df  Sum Sq Mean Sq  F value    Pr(>F)
Sex             1  127.39  127.39  33.4609 2.950e-08 ***
SizeClass       2 1667.48  833.74 218.9914 < 2.2e-16 ***
Sex:SizeClass   2    8.27    4.14   1.0865    0.3395
Residuals     190  723.36    3.81
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Note: this ANOVA table is displaying the results of Type I tests—each factor is tested in order. So the F-test for Sex is really comparing the model with Sex to one with only a mean, while the F-test for SizeClass is comparing the model with both Sex and Size Class to the one with only Sex.
A better way to go about things is with Type II tests, which compare the full model to a model without the factor of interest. I would recommend the `Anova()` function (capital A) in the `car` library created by John Fox for his *An R and S-plus Companion to Applied Regression*. There are several other useful functions in it.

```
> library(car) #Assuming you have installed the package
> Anova( lm(Mass ~ Sex * SizeClass, data = Sal) )
Anova Table (Type II tests)

Response: Mass
               Sum Sq  Df  F value Pr(>F)
Sex              0.26   1   0.0677 0.7950   NOT THE SAME
SizeClass     1667.48   2 218.9914 <2e-16 ***
Sex:SizeClass    8.27   2   1.0865 0.3395
Residuals      723.36 190
```

[ξ] Actually, you can also use the `aov()` function, which calls `lm()` for you, but you still need to use the `summary()` function to get test statistics, so you don't really gain anything. Your choice.

Some useful shortcuts:

Assign a name to the fitted object within the function call

```
> anova( aov.SexBySizeC <- lm(Mass ~ Sex * SizeClass, data = Sal) )
Analysis of Variance Table

Response: Mass
               Df  Sum Sq Mean Sq  F value    Pr(>F)
Sex             1  127.39  127.39  33.4609 2.950e-08 ***
SizeClass       2 1667.48  833.74 218.9914 < 2.2e-16 ***
Sex:SizeClass   2    8.27    4.14   1.0865    0.3395
Residuals     190  723.36    3.81
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

> ls()
[1] "NewSVL"         "Sal"         "aov.SexBySizeC"  "lm.SVL"
```

Refit a model with new data

```
> anova(aov.SexBySizeC.100 <- update(aov.SexBySizeC, data = Sal[1:100,]))
```

OR

```
> remove <- 101:196
> anova(aov.SexBySizeC.100 <- update(aov.SexBySizeC, subset = -remove))
```

#Just using the first 100 observations… sort of a silly example, but removing "outliers" or certain groups is easy this way. The minus (-) in from of the vector, `remove`, says "don't use these observations." We could just as easily made a vector, `use <- 1:100` and written "`subset = use`"

```
Analysis of Variance Table
Response: Mass
              Df  Sum Sq Mean Sq  F value    Pr(>F)
Sex            1   86.07   86.07  16.0020 0.0001261 ***
SizeClass      2 1179.42  589.71 109.6369 < 2.2e-16 ***
Sex:SizeClass  2   25.03   12.51   2.3266 0.1032243
Residuals     94  505.60    5.38
```

Or with different predictors

```
> anova(aov.SexSizeC <- update(aov.SexBySizeC, . ~ Sex + SizeClass))
```

OR equivalently

```
> anova(aov.SexSizeC <- update(aov.SexBySizeC, . ~ . -Sex:SizeClass))
```

# the period (.) mean "previous value", so ". ~ Sex + SizeClass" means use the previous left side of the formula and this new right side, while ". ~ . -Sex:SizeClass" means use the previous left side and the previous right, only remove the interaction term (`Sex:SizeClass`). Notice the minus sign again meaning remove or don't use.

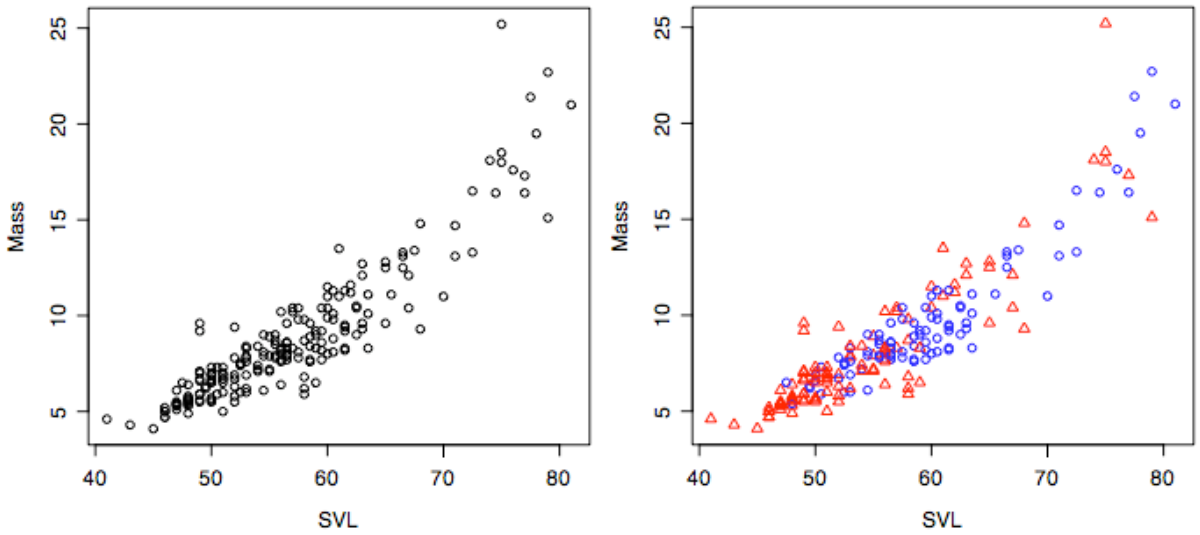
```
Analysis of Variance Table
Response: Mass
           Df  Sum Sq Mean Sq F value    Pr(>F)
Sex         1  127.39  127.39  33.431 2.950e-08 ***
SizeClass   2 1667.48  833.74 218.794 < 2.2e-16 ***
Residuals 192  731.64    3.81
```

What do you expct this statement to do?

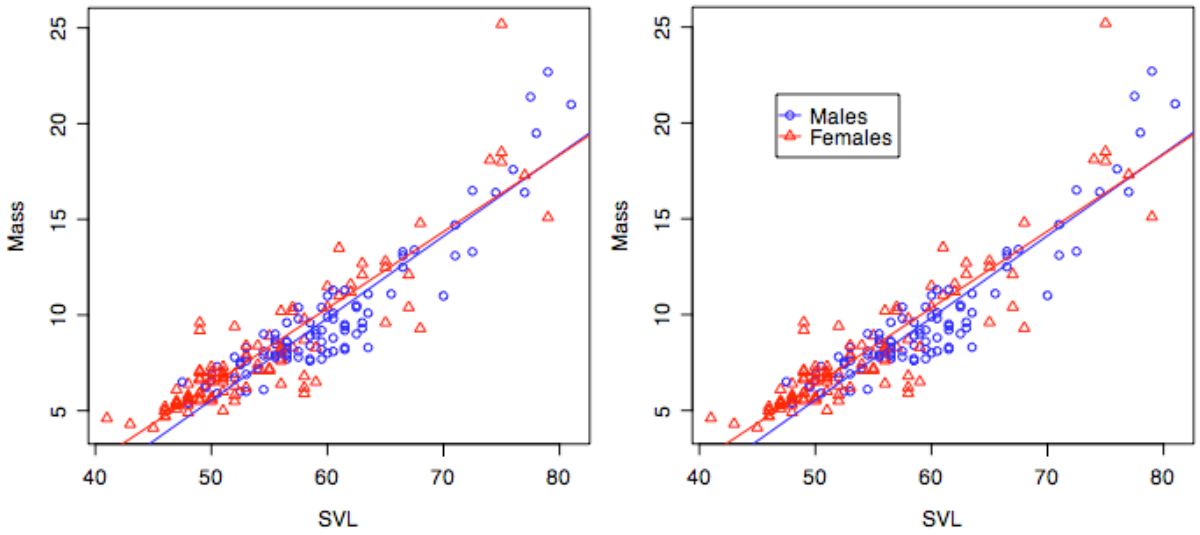
```
> anova( update(aov.SexBySizeC, . ~ . +Treat))
```

**More on plotting: adding trend lines to figures**

```
> plot(Mass ~ SVL, data = Sal)

> colors = ifelse(Sal$Sex == "M", "Blue", "Red") #condition, yes, no
> symbols = ifelse(Sal$Sex == "M", 1, 2)
> plot(Mass ~ SVL, data = Sal, col = colors, pch = symbols)
```



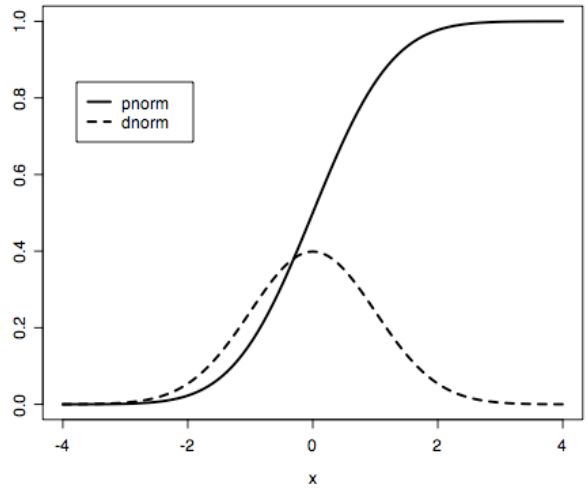

```
> abline( lm(Mass ~ SVL, data = Sal[Sal$Sex == "M", ]), col = "Blue")
#abline()adds straight lines to plots… here it gets the intercept and slope from the fitted lm
object, but you can also specify them manually, abline(a=intercept, b=slope)
> abline( lm(Mass ~ SVL, data = Sal[Sal$Sex == "F", ]), col = "Red")

> legend( locator(1), legend = c("Males", "Females"), col = c("Blue",
"Red"), lwd = 1, pch = c(1,2)) #I used lwd = 1 (line width) to force it include a line
```



**Statistical distribution in [R]**

[R] has statistical functions built in which can give you the probability density (PDF), probability distribution (CDF), quantiles, or random deviates for *many* statistical distributions.

| Distribution | R name | Additional arguments | Argument defaults |
|---|---|---|---|
| beta | beta | shape1 (α), shape2 (β) | |
| binomial | binom | size (n), prob (p) | |
| Chi-square | chisq | df (degrees of freedom υ) | |
| continuous uniform | unif | min (θ1), max (θ2) | min = 0, max = 1 |
| exponential | exp | rate (= 1/β) | rate = 1 |
| *F* distribution | f | df1 (υ1), df2 (υ2) | |
| gamma | gamma | shape (α), scale (β) | scale = 1 |
| hypergeometric | hyper | m = r, n = N−r, k = n (sample size) | |
| normal | norm | mean (μ), sd (σ) | mean = 0, sd = 1 |
| Poisson | pois | lambda (λ) | |
| *t* distribution | t | df (degrees of freedom υ) | |
| Weibull | weibull | shape, scale | scale = 1 |




`dnorm` calculates the probability density at[§] whatever `x` you provide. This is the curve we usually see.

`pnorm` calculates the area to the left of `x` on the `dnorm` curve

`qnorm` calculates the "cutoff" value of `x` that corresponds with the quantile or percentage you give it.

[§]This is probably more than you need, but evaluating a PDF at `x` actually finds the probability density *very near* `x`. At `x` the probability is zero because the PDF is continuous.

Wondering how I plotted these curves? You could, of course, make a vector of x-values from -4 to 4 with lots of points along it and then plot the `pnorm` curve (because it goes higher and therefore sets the y-limits right) and add the `dnorm` curve using `lines()` (although `points(…, type = "l")` would work as well):

```
> x <- seq(from = -4, to = 4, length = 100)
> plot(pnorm(x) ~ x, type = "l")
> lines(dnorm(x) ~ x, type = "l", lty = 2)
```

Instead, I used the `curve()` function:

```
> curve(pnorm, from = -4, to = 4)  # Note I only specify the name of the function
> curve(dnorm, from = -4, to = 4, lty = 2, add = T) # add it to the current plot
```

For instance, if you wanted to find the probability of observing $x \approx 1.96$ from a normal with a mean of zero and standard deviation you would evaluate the density function (PDF) at 1.96.

```
> dnorm(1.96, mean = 0, sd = 1)
[1] 0.05844094
```

If you wanted to know how likely it was to get a value of 1.96 or less you would evaluate the distribution function at 1.96

```
> pnorm(1.96, mean = 0, sd = 1)
[1] 0.9750021
```

Mostly we're interested in the probability of getting a value as big or bigger than x, like:

```
> 1 - pnorm(1.96, mean = 0, sd = 1)
[1] 0.02499790
```

The cutoff value of `x` associated with the 97.5$^{th}$ percentile is:

```
> qnorm(0.975, mean = 0, sd = 1)
[1] 1.959964
```

If you wanted five random variables from the normal:

```
> set.seed(0) #This is just so that we get the same answers.
> rnorm(5, mean = 0, sd = 1)
[1]  1.2629543 -0.3262334  1.3297993  1.2724293  0.4146414
```

These random variables are really useful for simulating data. David Williams, for instance, was interested in how error or imprecision in the "fixes" of GPS collars might influence some metrics of space use. So he added noise to his data using the `rnorm()` function and refit his statistics.

In the case of the salamander data, I simulated the mortality data, `Died`, using the `rbinom()` function, where the probability of death was a function of the body mass index (BMI). Something like this:

```
> Sal$BMI = Sal$Mass/Sal$SVL
> summary(Sal$BMI)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.09111 0.12900 0.14330 0.15140 0.16820 0.33600

> Sal$Died = 0 #Make a new vector of "live" animals
> ex <- which(Sal$Treat == "Exposed")
Then each animals "exposed" to virus gets a 1 ("dead") with probability of BMI + 0.6
> Sal$Died[ex] <- rbinom(n = length(Sal$BMI[ex]), size = 1,
      prob = Sal$BMI[ex] + 0.6)
```

`size`, by the way, is just the number of "trials" in each binomial experiment. Since there can only be one death per animal it must be one.

**Other useful functions for generating fake data:**

**`rep()`** repeats whatever you give it a certain number of times.

**`seq()`** you've seen

**`gl()`** stands for "generate levels." Good for creating treatments.

**`exapand.grid()`** creates dataframes from all combinations of the factors you give it

**`sample()`** takes samples of a given size with or without replacement

**`replicate()`** evaluates a function or expression `n` times

```
> hist( replicate(n = 10000, mean(rexp(10)) ) ) #The means of a bunch of
```

random draws [`n=10000`] from an exponential distribution [`rexp(10)`] converge on a normal

**Making our own functions – a Chi square test**

Let's say we are interested in whether animals in different size classes were more or less likely to die when exposed to this virus. What we want to do, then, is compare the proportion that died in each size class, i.e., a Chi-square test.

First, it probably does not make sense to include the control animals in our test:

```
> Sal.Exp <- Sal[Sal$Treat == "Exposed",]
> str(Sal.Exp)
'data.frame':     109 obs. of  7 variables:
 $ Animal   : int  7 8 10 11 12 13 17 19 25 28 ...
 $ SVL      : num  70 57 67 74 72.5 68 60 71 58 58 ...
 $ Mass     : num  11 8.3 12.1 18.1 16.5 14.8 10.4 13.1 9.8 8.7 ...
 $ Sex      : Factor w/ 2 levels "F","M": 2 1 1 1 2 1 1 2 1 1 ...
 $ Treat    : Factor w/ 2 levels "Control","Exposed": 2 2 2 2 2 2 2 2 2 ...
 $ Died     : int  1 1 1 1 1 1 1 1 1 1 ...
 $ SizeClass: Factor w/ 3 levels "Sm","Med","Lg": 3 2 3 3 3 3 2 3 2 2 ...
```

Next, remember that the Chi-square test is: $\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$ where $O_i$ is the number observed in cell $i$ (e.g., those in the small size class that died), and $E_i$ is the number expected in that cell.

One of the first things we need is to know how many animals in each class died, and how many did not. We can do this with what we know so far in one of two ways:

```
> sum(Sal.Exp$Died[Sal.Exp $SizeClass == "Sm"]) #Since Died =1 when the animal died, this finds the total number that died in the size class
[1] 36
> sum(Sal.Exp$Died[Sal.Exp$SizeClass == "Med"])
[1] 49
> sum(Sal.Exp$Died[Sal.Exp$SizeClass == "Lg"])
[1] 10
```

And then those that did not die:

```
> length(Sal.Exp$Died[Sal.Exp$SizeClass == "Sm"]) -
sum(Sal.Exp$Died[Sal.Exp$SizeClass == "Sm"])#Total number in the size class minus those that died
[1] 3
```

Or we can use the `summary()` function to give us both those that died and those that did not:

```
> summary( as.factor(Sal.Exp$Died[Sal.Exp$SizeClass == "Sm"]) ) #Since Died is numeric (0/1) we must first convert them to factors so that summary gives us counts rather than means
 0  1
 3 36
```

These work, but they're pretty cumbersome. This next function is *very* useful:

```
> (Obs <- table(Sal.Exp$Died, Sal.Exp$SizeClass) )
    Sm Med Lg
  0  3   9  2
  1 36  49 10
```

Let's ignore the fact that we have so few animals that survived, making the Chi-square test a little suspect

Next, we need to figure out the expected number in each cell of this table. Basically, we just need to know the proportion that died over all size classes and then the proportion in each size class. We then multiply these proportions to get the expected values.

```
> (sr <- rowSums(Obs) ) #We'll use some built-in functions to make this quicker
 0  1
14 95
> (sc <- colSums(Obs) )
 Sm Med  Lg
 39  58  12
> (n <- sum(Obs))
[1] 109
> 14*39/109 #The expected number of small, surviving salamanders
[1] 5.009174
> 14*58/109 #The expected number of medium, surviving salamanders
[1] 7.449541
> sr[1]*sc[3]/n #Notice that this is the same as 14*12/n
[1] 1.541284
```

OK, still a little tedious. Let's automate this a bit:

```
> Ex <- matrix(ncol = nc, nrow = nr) #an empty matrix of the right size
> for(i in 1:nr) {      # This is a loop. It starts at i = 1 and keeps incrementing i until
                        # i = nr (i.e., 2). Each time id does whatever is in the curly
                        # brackets {}
+     for(j in 1:nc) {
+         Ex[i,j] = (sr[i] * sc[j]) / n
+     }
+ }                      #This is the end of the i loop
> Ex
          [,1]      [,2]      [,3]
[1,]  5.009174  7.449541  1.541284
[2,] 33.990826 50.550459 10.458716
```

Right, so we have a matrix of observed values (`Obs`) and one of expected values (`Ex`). We need their difference. This easy given the way [R] works with vectors and matrices:

```
> Obs - Ex
            Sm        Med         Lg
  0 -2.0091743  1.5504587  0.4587156
  1  2.0091743 -1.5504587 -0.4587156
```

Their difference squared

```
> (Obs - Ex)^2
          Sm      Med       Lg
  0 4.036781 2.403922 0.210420
  1 4.036781 2.403922 0.210420
```

Their difference squared, then divided by expected

```
> ( (Obs - Ex)^2 ) / Ex
            Sm        Med         Lg
  0 0.80587761 0.32269399 0.13652250
  1 0.11876091 0.04755490 0.02011911
```

All summed…

```
> sum( ( (Obs - Ex)^2 ) / Ex )
[1] 1.451529
```

That's our Chi-square test statistic!

Is this test statistic significant? We can use the built in chi-square distribution, keeping in mind that we want to know how likely it is that we would get a value of `1.451529` *or higher* by random chance from a chi-square distribution with (i-1)(j-1) = (2-1)(3-1) = 2 degrees of freedom:

```
> 1 - pchisq(1.451529, df = 2)
[1] 0.4839544
```

At $p = 0.484$ we would reject the hypothesis that there are difference in mortality among the size classes.

So now we have all of the parts of a chi-square test. Let's see if we can't make a function to do this for us. But let's start a little simpler just to see how they work.

```
> xsquared <- function(x) {x^2}
```

We just made a function, which we assigned to "`xsquared`," that takes an argument, `x`. We can then call our function just like we would any other function. It just returns the square of whatever `x` we provide.

```
> xsquared(4)
[1] 16
> a = 3
> xsquared(a)
[1] 9
```

Easy, right? Well, obviously they can get more complicated. We had a lot of steps to get our chi-square test statistic, for instance. But since we have all of the pieces we need, it's just a matter putting them together in the right way.

First, fire up a text editor. (Something that works well with plain text, like wordpad, but not Word… there are also a lot of code editors that will color-code your bits of text and keep track of parentheses and such. I'd recommend finding one you like and learning how it well.)

Type in the basic structure. We want our function to take two vectors, the first with one factor (e.g., `Dead`) and the second with another (e.g., `SizeClass`). We can use whatever place-holder names we want instead of x and y inside the parentheses, but whatever names we use here will be what we use inside the function's curly brackets.

```
X2test <- function(x, y) {
}
```

Working in the same order as we did before, we first want to create our table of observations:

```
X2test <- function(x, y) {
      Obs <- table(x,y)
}
```

And then calculate a few useful values, like the number of rows and columns and some sums

```
X2test <- function(x, y) {

	Obs <- table(x,y)

	nr <- nrow(Obs)
	nc <- ncol(Obs)

	sr <- rowSums(Obs)
	sc <- colSums(Obs)
	n <- sum(Obs)
}
```

Then we want to calculate our expected matrix

```
X2test <- function(x, y) {

	Obs <- table(x,y)

	nr <- nrow(Obs)
	nc <- ncol(Obs)

	sr <- rowSums(Obs)
	sc <- colSums(Obs)
	n <- sum(Obs)

	Ex <- matrix(ncol = nc, nrow = nr) #Matrix of right size

	#Multiply the column sums by the row sums and divide by
	#n to create the matrix of expected values
	for(i in 1:nr) { #Loop through the i rows of the matrix

		for(j in 1:nc) { #Loop through the j columns

			Ex[i,j] = (sr[i] * sc[j]) / n

		} #End of j loop

	} #end of i loop
}
```

We can then have the function calculate the chi-square test statistic with

```
ChiStat <- sum( ( (Obs - Ex)^2 ) / Ex )
```

And calculate the degrees of freedom and the p-value

```
DF <- (nr-1)*(nc-1)
Pval <- 1 - pchisq(ChiStat, df = DF)
```

Then we just need the function to print out the results:

```
X2test <- function(x, y) {

      Obs <- table(x,y)

      nr <- nrow(Obs)
      nc <- ncol(Obs)

      sr <- rowSums(Obs)
      sc <- colSums(Obs)
      n <- sum(Obs)

      Ex <- matrix(ncol = nc, nrow = nr) #Matrix of right size

      #Multiply the column sums by the row sums and divide by
      #n to create the matrix of expected values
      for(i in 1:nr) { #Loop through the i rows of the matrix

            for(j in 1:nc) { #Loop through the j columns

                  Ex[i,j] = (sr[i] * sc[j]) / n

            } #End of j loop

      } #end of i loop

      ChiStat <- sum( ( (Obs - Ex)^2 ) / Ex )
      DF <- (nr-1)*(nc-1)
      Pval <- 1 - pchisq(ChiStat, df = DF)

      cat("Probability of observing a test statistic ≥", ChiStat,
      "by chance from a Chi-square distribution with", DF,
      "degrees of freedom =", Pval)


}
```

`cat()`, `print()`, `format()`, and `paste()` all concatenate & print text (and variables) in slightly different ways.

Copy and paste this into [R], and after fixing your inevitable typos, try it:

```
> X2test(Sal.Exp$Died, Sal.Exp$SizeClass)

Probability of observing a test statistic ≥ 1.451529 by chance from a
Chi-square distribution with 2 degrees of freedom = 0.4839544
```

This is exactly what built-in chi-square test does, only they've made it a bit more flexible and have a slightly prettier output. ☺

```
> chisq.test(Sal.Exp$Died, Sal.Exp$SizeClass)

      Pearson's Chi-squared test

data:  Sal.Exp$Died and Sal.Exp$SizeClass
X-squared = 1.4515, df = 2, p-value = 0.484

Warning message:
In chisq.test(Sal.Exp$Died, Sal.Exp$SizeClass) :
  Chi-squared approximation may be incorrect
```

We got it right! (The warning is because we have < 5 observations in two cells.)

We had used a set of loops to calculate the table of expected values:

```
Exp <- matrix(ncol = nc, nrow = nr)
for(i in 1:nr) {
      for(j in 1:nc) {
            E[i,j] = (sr[i] * sc[j]) / sum(z)
      }
}
```

We could have used a function called `outer()` to do this in one step:

```
Exp <- outer(sr, sc, "*")/n
```

This function does vector multiplication on the two vectors, which is their "outer product." Sometimes it pays to search around for pre-made functions, but there is *always* a simple, if perhaps less elegant way to do it.

**Important considerations on writing functions:**
If you want your function to return something (rather than, say, just print the results) use the statement, `return(ObjectToReturn)`. [R] will *implicitly* return the value of simple statements, like in our `squarex()` function, and in more complex cases when you are calculating many things it will return the last expression evaluated, but to be clear it is best to use `return()`.

When you assignments made inside a function apply only *within* that function. For instance

```
> x <- 0
> x
[1] 0
> xplus <- function(x) {(x <- x + 1)}
> xplus(x)
[1] 1 #This is the output from within the function. In here x = 1
> x
[1] 0 #x at the command prompt is unchanged. x = 0 still.
```

If we want to act on the variable `x` that is *outside* of the function (i.e., in the "global environment") we can use the `<<-` assignment operator.

```
> xplus <- function(x) {(x <<- x + 1)}
> xplus(x)
[1] 1 #Output from within the function
> x
[1] 1 #The function changed the value of x in the global environment
```

Errors are a big part of writing functions. Often we don't know where in the function we are getting stuck. The `traceback()` function (entered just like that at the command prompt) will tell you which loop or function you were in when the error was called. A classic way to see how things look inside your function is to put `print()` statements in strategic places. It's hackey, but it works. There are other tools you might try if get more advanced. Let me know if you take the time to figure them out.

**Getting data into the right format:**
I have this dataset (Mice-Ramapo.csv at http://www.esf.edu/efb/brunner/Rworkshop.htm), with the locations of mice captured on an 8 by 8 trapping grid over a summer in 2006. I'm interested in whether there are "hot-spots" of mouse captures.

```
> Mice <- read.csv("/Users/brunner/Desktop/Rworkshop/Mice-Ramapo.csv")
> str(Mice)
'data.frame':     95 obs. of  9 variables:
 $ Year       : int  2006 2006 2006 2006 2006 2006 2006 2006 2006 2006 ...
 $ Month      : int  5 5 5 5 6 5 6 6 6 6 ...
 $ Day        : int  30 30 30 30 13 30 13 13 13 13 ...
 $ Location   : Factor w/ 52 levels "","A1","A2","A3",..: 42 4 6 51 37 ...
 $ Primary_Tag: Factor w/ 89 levels "","D151","G214",..: 2 3 4 5 5 6 6 ...
 $ Sex        : Factor w/ 3 levels "","F","M": 3 3 3 2 2 2 2 2 2 3 ...
 $ Age        : Factor w/ 4 levels "","A","J","S": 2 4 4 2 2 4 2 4 4 2 ...
 $ Weight     : int  15 13 15 16 20 22 21 15 13 19 ...
 $ Ticks      : int  6 1 0 4 0 0 9 1 0 7 ...
```

```
> tail(Mice) #Can you guess what head() will show you?
   Year Month Day Location Primary_Tag Sex Age Weight Ticks
90 2006     7  25       C6        X661   M   A     21     1
91 2006     7   4       D4        X662   M   S     16     1
92 2006     7   4       A3        X997   F   A     22     4
93 2006     7   4       E3        X998   M   A     21     0
94 2006     7   4       F3        X999   F   A     17     0
95   NA    NA  NA                                  NA    NA
```

Oops, we have an extra row in our dataset… let's get rid of it:

```
> Mice <- Mice[-95, ]
```

OK, we're set. But wait, the data are in these long columns. What we'd really like is a matrix of the number of mice caught that mirrors the actual trapping grid. This can be the tricky part about [R] once the data are in the right format things work easily, but getting data into that format can be really, really frustrating. So here is one way of converting these lists of trap locations into a grid.

First, we need to extract rows and columns from the trap `Location`

```
> Mice$Row <- substr(Mice$Location, start = 1, stop = 1) #substr() extracts the characters beginning at start and ending at stop
> Mice$Col <- as.numeric( substr(Mice$Location, start = 2, stop = 2) )
#here we're using as.numeric() to convert the character version of each number into the numeric version
```

```
> Grid <- table(Mice$Row, Mice$Col)
   1 2 3 4 5 6 7 8
A 1 1 4 0 4 4 2 1
B 1 1 2 0 2 0 2 1
C 0 3 0 0 1 2 1 1
D 1 1 0 1 0 1 2 1
E 3 0 2 0 1 1 1 1
F 1 1 1 0 3 4 1 1
G 1 3 1 3 0 1 0 1
H 2 4 4 1 2 3 5 1
```

OK, that was pretty easy. But what if we wanted something different, like the mean number of ticks on the animals caught at each trapping station? `tapply()` to the rescue!

```
> Grid.meanTick <- tapply(Mice$Ticks, INDEX= list(Mice$Row, Mice$Col), FUN
= mean)
```

This function takes the variable you want to work on first, and then it needs the factor(s) that you want it to group the data by (called `INDEX`). Since we want to sort by the combination of `Row` and `Col` we need to give `INDEX` a list. Lastly we need to give it the function we want it to apply to each group of data. We can use the pre-made functions, like `mean()`, or our own.

```
> Grid.meanTick
     1         2    3        4    5        6    7  8
A  4.0 12.000000 2.75       NA 21.5 10.00000  5.0  3
B 29.0  8.000000 6.50       NA 12.5       NA  2.0  9
C   NA 14.333333   NA       NA  1.0  4.00000  5.0 23
D  3.0 14.000000   NA 1.000000   NA 31.00000 10.5  6
E 11.0        NA 6.00       NA  1.0 32.00000 27.0  0
F  9.0  9.000000 0.00       NA 12.0  7.00000  0.0  3
G 36.0  6.666667 9.00 5.333333   NA 25.00000   NA  1
H 26.5 14.500000 8.25 8.000000 23.0 10.33333 11.8  3
```

You probably noticed all of the NAs in the grid, where no animals were caught. We might want to leave them, but some functions or statistics will choke on them, so let's replace them with zeros.

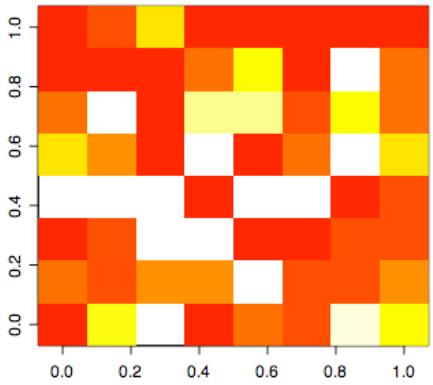
```
> is.na(Grid.meanTick)  #is.na() returns TRUE if the cell has an NA. What will
!is.na() return? Hint: the exclamation mark (!) means "not"
      1     2     3     4     5     6     7     8
A FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
B FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE
C  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE
D FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE
E FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE
F FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
G FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE
H FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

> Grid.meanTick[is.na(Grid.meanTick)] = 0 #Everywhere there is a TRUE (i.e., there is
an NA) we set it equal to zero
> Grid.meanTick
     1         2    3        4    5        6    7  8
A  4.0 12.000000 2.75 0.000000 21.5 10.00000  5.0  3
B 29.0  8.000000 6.50 0.000000 12.5  0.00000  2.0  9
C  0.0 14.333333 0.00 0.000000  1.0  4.00000  5.0 23
D  3.0 14.000000 0.00 1.000000  0.0 31.00000 10.5  6
E 11.0  0.000000 6.00 0.000000  1.0 32.00000 27.0  0
F  9.0  9.000000 0.00 0.000000 12.0  7.00000  0.0  3
G 36.0  6.666667 9.00 5.333333  0.0 25.00000  0.0  1
H 26.5 14.500000 8.25 8.000000 23.0 10.33333 11.8  3
```

Actually, for this next part it will be easier if we leave the missing values, so let me reset things:

```
> Grid.meanTick <- tapply(Mice$Ticks, list(Mice$Row, Mice$Col), mean)
```

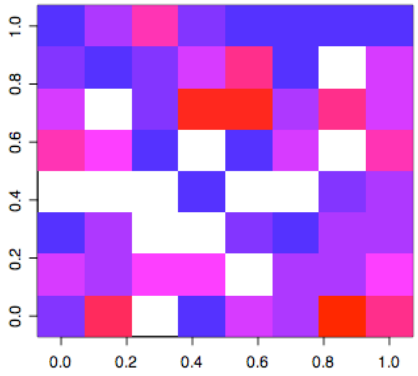
Let's plot it.

```
> image(Grid.meanTick) #There are actually many ways to plot "heat maps", but image() is pretty basic and flexible.
```




That's kind of cool, but the axes are in the wrong places, plus they don't really mean anything, and the colors are a bit hard to interpret (red is low numbers and white is large numbers, but white is also NA. Confusing!). Let's deal with the colors first:

```
> image(Grid.meanTick, col = rainbow(n = 12, start = 0.7, end = 0))
#rainbow() is one of many functions that will create color gradients (others are heat.colors() [default for image()], terrain.colors(), topo.colors(), cm.colors() ). Most of these functions need the number of colors (n) you want it to generate and some want a start and stop position along the color ramp. Just experiment. Another very useful, but tricky function is rgb(), which allows you to control the intensity of each color as well as the transparency or alpha level.
```

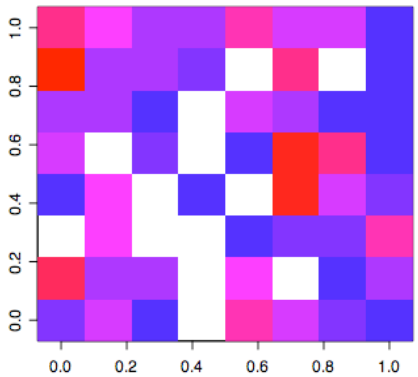



```
> round(Grid.meanTick)
   1  2  3  4  5  6  7  8
A  4 12  3 NA 22 10  5  3
B 29  8  6 NA 12 NA  2  9
C NA 14 NA NA  1  4  5 23
D  3 14 NA  1 NA 31 10  6
E 11 NA  6 NA  1 32 27  0
F  9  9  0 NA 12  7  0  3
G 36  7  9  5 NA 25 NA  1
H 26 14  8  8 23 10 12  3
```

It's garish, sure, but at least it is informative. Blue is low and red is big; NA is white. Notice that the A "row" is actually the first *column* in this figure and that column 1 is running along the bottom. (It's always a good idea to check your figures against the actual data.)

We can transpose the matrix with the t() function and plot that:

```
> image(t(Grid.meanTick), col = rainbow(12, start = 0.7, end = 0))
```

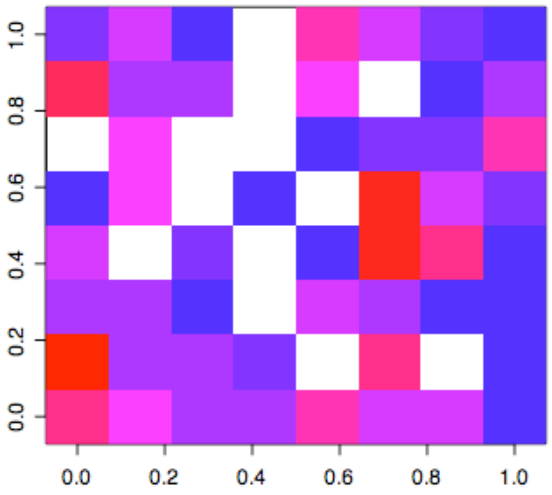



```
> round(Grid.meanTick)
   1  2  3  4  5  6  7  8
A  4 12  3 NA 22 10  5  3
B 29  8  6 NA 12 NA  2  9
C NA 14 NA NA  1  4  5 23
D  3 14 NA  1 NA 31 10  6
E 11 NA  6 NA  1 32 27  0
F  9  9  0 NA 12  7  0  3
G 36  7  9  5 NA 25 NA  1
H 26 14  8  8 23 10 12  3
```

That's closer; column 1 is on the left and column 8 is on the right. The rows, however, are upside down, with A on the bottom.

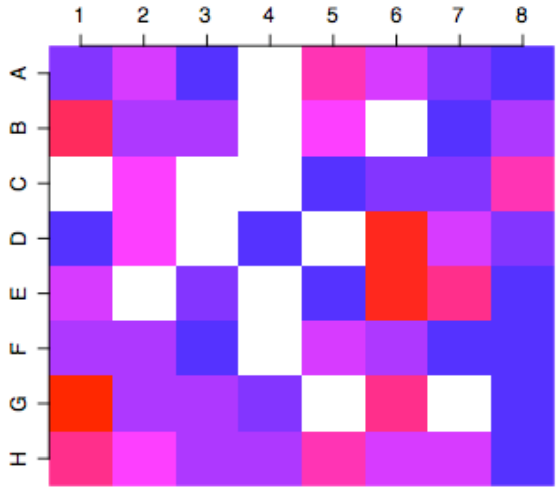
Well, we can use our indexing prowess to switch that around.

```
> image( t(Grid.meanTick[8:1, ]), col = rainbow(12, start = 0.7, end = 0) )
```



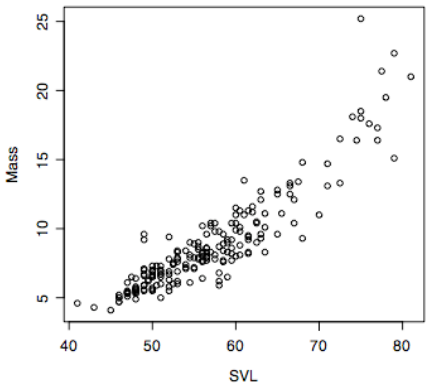

Just right! Now for the axes:

```
> image( t(Grid.meanTick[8:1, ]), col = rainbow(12, start = 0.7, end = 0), axes = F) #Turn off the axes
> axis(side = 3, at = seq(0, 1, length=8), labels = 1:8) #Then add the axes where we want them. The side numbers go clockwise from the bottom. The at tells axis() where to put the tick marks and labels; they have to be in [0,1]
> axis(side = 2, at = seq(0, 1, length=8), labels = rownames(Grid.meanTick)[8:1]) #Getting the names of the rows from the matrix, then need to invert them since like most y-labels this goes from bottom to top.
```



**Plotting with `tapply()`:**
Let's go back to the Sal dataset.

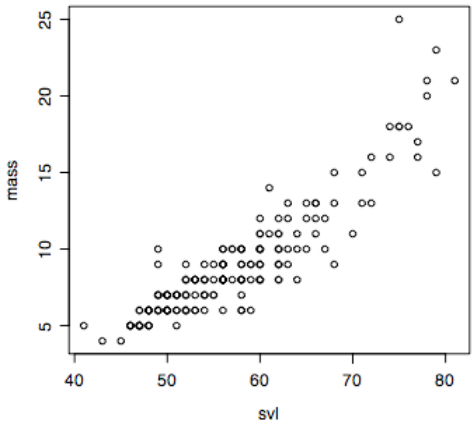
```
> plot(Mass ~ SVL, data = Sal)
```

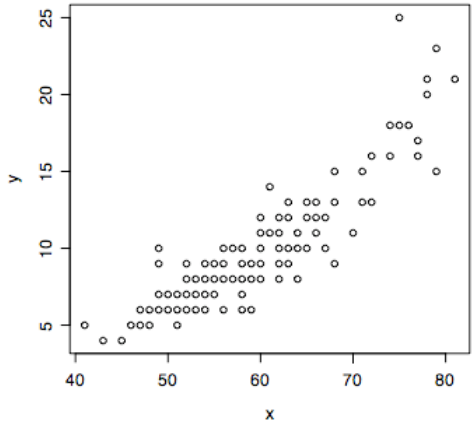



Notice that we have a lot of overlapping points. This is a problem with big datasets. We can use tapply() to instead make one point represent all of the individuals at that SVL and Mass, and the size (really, the areas) of the point to represent the number of individual behind it.

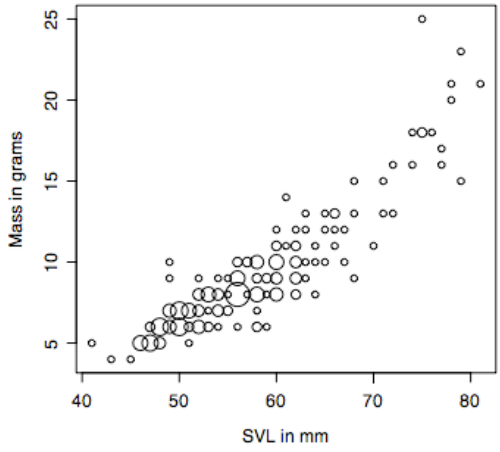
First, we want to group our data a bit courser since we have so many levels of Mass and SVL

```
> mass <- round(Sal$Mass) #defaults to zero digits, but you can change this
> svl <- round(Sal$SVL)
> plot(mass ~ svl)
```




```
> y <- tapply(mass, list(mass, svl), mean) #This makes one y-value per level of mass by svl, e.g., one y-value for all of the animals that weight 10g at 60mm SVL. I used mean because the mean weight of five animals that all weigh 10g is 10g!
> x <- tapply(svl, list(mass, svl), mean) #Same thing for the x-values (i.e., SVL)
> n <- tapply(svl, list(mass, svl), length) #The number of observations at each combination of mass and svl is calculated as the length of the vector of svl's ( mass would have worked, too).
> plot(y ~ x) #Above right. Notice we no longer have multiple points overlapping.

> plot(y ~ x, cex = sqrt(n), ylab = "Mass in grams", xlab = "SVL in mm")
```




`cex` controls the size of the plotted characters. I used square root of `n` so that the area would be proportional to the sample size (rather than the diameter). `xlab` and `ylab` are pretty self-explanatory.

**More on data management:**

Back to the `Mice` dataset. You may have noticed that we had 94 mouse captures on this grid, but a few animals were apparently caught more than once.

How do we figure out how many individuals we have?

```
> length(levels(Mice$Primary_Tag))
[1] 85
```

Wait a second! I happen to know that there were only 84 individuals on the grid. What happened? Well, remember that we had an extra row of empty values when we imported the data? [R] treated that blank row as an extra level of the `Primary_Tag` factor, `""`, i.e., an empty level. To get rid of this we can just re-factor our variable.

```
> Mice$Primary_Tag <- factor(Mice$Primary_Tag)
> length(levels(Mice$Primary_Tag))
[1] 84
```

What if we want to know the location of first capture for each individual? You might think there would be a function called something like unique. There is, but it doesn't work quite the way you might imagine. What it does is remove duplicated rows (across *all* columns). The only way to get the unique tag numbers is this way:

```
> Mice.uniq <- unique(Mice[, 5])
> str(Mice.uniq)
 Factor w/ 84 levels "D151","G214",..: 1 2 3 4 5 6 7 8 9 10 ...
> length(Mice.uniq)
[1] 84
```

That's not terribly helpful. We can get that with `levels()` just fine. A better approach would be to use the `duplicated()` function, which returns a logical (`TRUE/FALSE`) vector indicating whether an entry (here, a tag number) has been seen before. If we want the *first* observation of each animal, then we need to make sure the first entry for each tag is the earliest in the year.

```
> newOrder <- order(Mice$Primary_Tag, Mice$Month, Mice$Day) #order() returns
```

the index of the first argument (here `Mice$Primary_Tag`) sorted in ascending order (by default, use the minus sign in front of the argument to reverse the order), and the breaks ties according to the next argument (`Month`) and then the next (`Day`). There is a `sort()` functon, but it works differently and only on a single vector.

```
> newOrder #Our data weren't too out of order.
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 23
[23] 22 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 44 43
[45] 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66
[67] 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88
[89] 89 90 91 92 93 94
```

Then we'll just rearrange our dataframe

```
> Mice <- Mice[newOrder, ]
```

```
> duplicated(Mice$Primary_Tag)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE
[14] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
[27] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE
[40] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE
[53] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[66] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[79]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[92] FALSE FALSE FALSE

> Mice.uniq <- Mice[!duplicated(Mice$Primary_Tag), ] #Notice the exclamation
  point to reverse the TRUE/FALSE levels… we want to keep the unduplicated entries after all.
> str(Mice.uniq)
'data.frame':     84 obs. of  11 variables:
 $ Year       : int  2006 2006 2006 2006 2006 2006 2006 2006 2006 2006 ...
 $ Month      : int  5 5 5 5 5 6 6 6 6 6 ...
 $ Day        : int  30 30 30 30 30 13 13 13 13 13 ...
 $ Location   : Factor w/ 52 levels "","A1","A2","A3",..: 42 4 6 51 13 ...
 $ Primary_Tag: Factor w/ 84 levels "D151","G214",..: 1 2 3 4 5 6 7 8 9 ...
 $ Sex        : Factor w/ 3 levels "","F","M": 3 3 3 2 2 2 2 3 3 3 ...
 $ Age        : Factor w/ 4 levels "","A","J","S": 2 4 4 2 4 4 4 2 2 2 ...
 $ Weight     : int  15 13 15 16 22 15 13 19 15 20 ...
 $ Ticks      : int  6 1 0 4 0 1 0 7 12 9 ...
 $ Row        : chr  "G" "A" "A" "H" ...
 $ Col        : num  4 3 6 7 7 2 3 3 3 1 ...
```

There we go… *that* is what we wanted.

Lastly, let me introduce you to the `merge()` function. Like it suggests, it can merge two data frames (or matrices) together based on some common set of columns. You can merge very different sorts of dataframes so long as they have a common column or two. They needn't even share a name, but they *do* need to have the same data (hence, "common"). It's worth reading (and re-reading) the help file on this one, but here is a quick example:

Let's say we need the location of first capture for ever row in the `Mice` dataframe. We need to merge the `Mice.uniq` dataframe into the `Mice` dataframe. Note that they are different sizes.

```
> dim(Mice)
[1] 94 11
> dim(Mice.uniq)
[1] 84 11
```

First, we should rename the variable corresponding to the location of first capture:

```
> names(Mice.uniq)
 [1] "Year"        "Month"       "Day"         "Location"
 [5] "Primary_Tag" "Sex"         "Age"         "Weight"
 [9] "Ticks"       "Row"         "Col"
> names(Mice.uniq)[4] <- "LFC"
```

Now, to merge them. It is a very good idea to use temporary names until you get it right… this took me a couple times.

```
> temp <- merge(Mice, Mice.uniq[,4:5], all.x = T)
```
# I only wanted merge by the tag numbers, not dates, etc., which explains the `Mice.uniq[,4:5]`. `all.x = T` means keep all rows in the `x` dataframe (=`Mice`), which implies duplicating some rows in the `y` dataframe (=`Mice.uniq`) when, for instance, a mouse was captured a second time.

```
> dim(temp)
[1] 94 12
```

```
> temp[1:10,]
   Primary_Tag Year Month Day Sex Age Weight Ticks Row Col Location LFC
1         D151 2006     5  30   M   A     15     6   G   4       G4  G4
2         G214 2006     5  30   M   S     13     1   A   3       A3  A3
3         G215 2006     5  30   M   S     15     0   A   6       A6  A6
4         G216 2006     5  30   F   A     16     4   H   7       H7  H7
5         G216 2006     6  13   F   A     20     0   F   7       F7  H7
6         I196 2006     5  30   F   S     22     0   B   7       B7  B7
7         I196 2006     6  13   F   A     21     9   B   8       B8  B7
8         I252 2006     6  13   F   S     15     1   G   2       G2  G2
9         I259 2006     6  13   F   S     13     0   A   3       A3  A3
10        I260 2006     6  13   M   A     19     7   B   3       B3  B3
```

Everything seem OK. So now we can overwrite the `Mice` dataframe

```
> Mice <- temp
```