R Course: Data Visualization

Fritz Günther

Note to myself: Activate all Animations before loading (search for multiinclude)

Topics

1. R - Some Basics
2. Discrete Data
   - Frequencies and Distributions
3. Continuous Data
   - Frequencies and Distributions
   - Relations between Continuous Variables
4. Plotting Data vs. Analyses
5. Stepwise Plotting
6. Controlling Graphical Parameters
7. Exporting Plots
8. Colors

R - Some basics

- Most of this course will focus on the base R plotting functions
- Other options are the packages `lattice` and `ggplot2`
- We can have a look at these later

R - Some basics

- Set your working directory with
  `setwd("C:/Users/fritz.guenther/Documents/R_course")`
- Check your current working directory with
  `getwd()`
- Check the files in your current working directory with
  `dir()`

R - Some basics

- Read a text table (here called `datfile.txt`) in your current working directory with
  ```
  read.table("datfile.txt")
  ```
- Read a text table in some other directory with
  ```
  read.table("C:/otherdir/datfile.txt")
  ```
- Read a .csv file with
  ```
  read.csv("datfile.csv")
  ```
  or
  ```
  read.csv2("datfile.csv")
  ```
  ,
  depending on the .csv format (, vs. ;)

R - Some basics

- Save the data in a variable

```
dat <- read.table("datfile.txt")
```

- Inspect the data

```
View(dat)
head(dat)
```

- Look at the data structure

```
str(dat)
summary(dat)
names(dat)
```

R - Some basics

- Extract a column by name (here: the column named `freq`)

```
dat$freq
dat[,"freq"]
```

- Extract a column by position (here: the second column)

```
dat[,2]
```

- Extract a row by position (here: the third row)

```
dat[3,]
```

R - Some basics

- If you don't know how a function works, use
  `?func`
  (with `func` being the name of the function)

Discrete Data:
Frequencies and Distributions

Discrete Data

- *Discrete Data* refers to cases where we have a finite, countable number of possible values
- Examples: native language, Yes/No-answers, one of X different sentence arrangements; strictly speaking, also error rates
- In a sense, also rating scales (for example rating 1–5 or 1–7) are also discrete data; however, these typically have *ordinal* structure

Discrete Data

- Our token data set: Sentence fragment arrangement
- Participants are given some sentence fragments (A, B, C) and have to arrange their order

Read the Data

```
dat <- read.table("sentence_arrangement.txt",header=T)
```

- `header = T` tells R that the first row contains the variable names
- Table of the response patterns
  `table(dat$arrangement)`

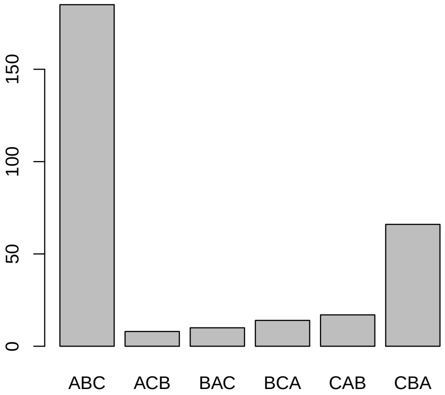
Inspect the Data

```
str(dat)
```

- We have 3 `conditions` à 10 `participants`, as well as their response patterns (`arrangement`)
- `condition` is not a number, but an experimental factor. Therefore: `dat$condition <- as.factor(dat$condition)`
- We further have their response times (`RT`) – when they started arranging the fragments – and their finishing times (`FT`) – when they completed the arrangements
- Within each condition, we have data for two different `time` points (pre and post)
- We also have participant answers whether the sentence is `true`

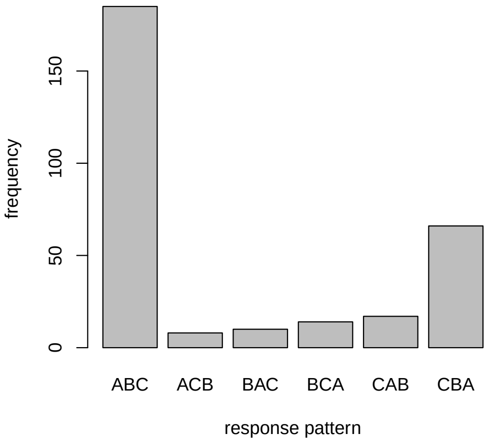
Bar plot of the response patterns

Rbarplot(table(dat$arrangement))



Bar plot: Customizing

```
barplot(table(dat$arrangement),
xlab="response pattern",ylab="frequency")
```

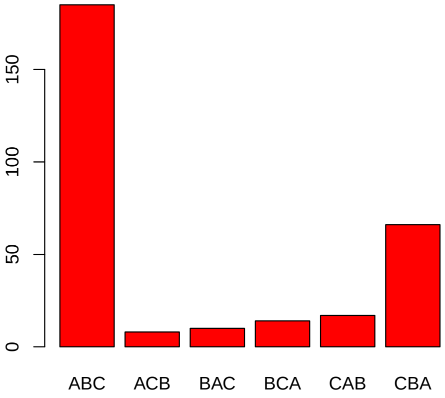


R Basics

- Strings in quotation marks ("red") are characters
- Strings without quotation marks (colors) are variable names (i.e., program code)

Bar plot: Customizing

```
barplot(table(dat$arrangement),
col="red")
```

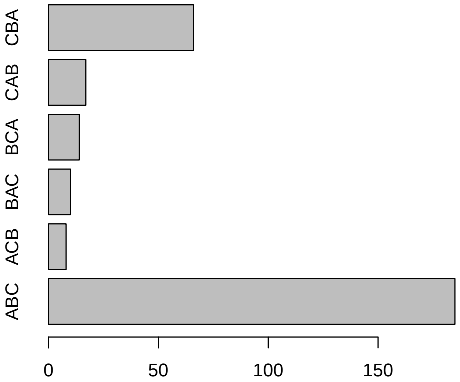


Colors in R: Colors with names

http://research.stowers.org/mcm/efg/R/Color/Chart/ColorChart.pdf

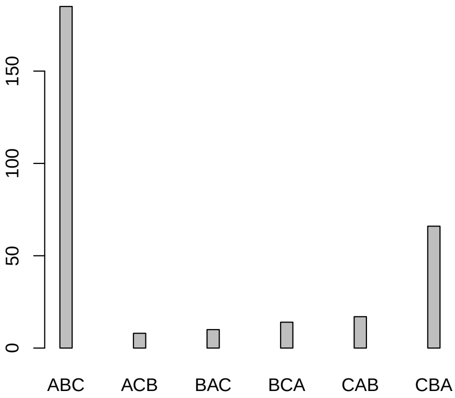
Bar plot: Customizing

```
barplot(table(dat$arrangement),
horiz=T)
```



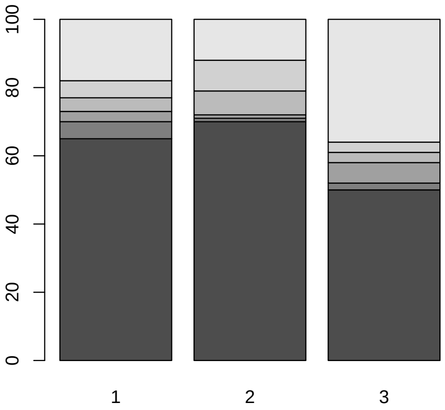
Bar plot: Customizing

```
barplot(table(dat$arrangement),
space=5)
```



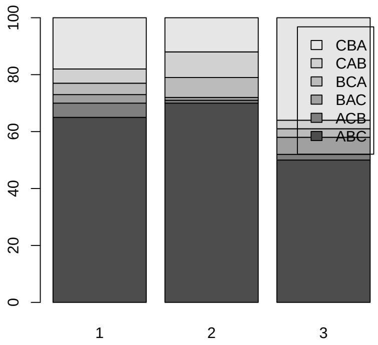
Bar plot by condition

```
barplot(table(dat$arrangement,dat$condition))
```



Bar plot by condition: Customizing

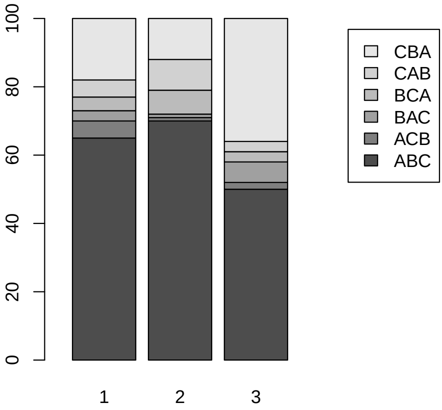
```
barplot(table(dat$arrangement,dat$condition),
legend=T)
```



Look crappy, let's position the legend somewhere else

Bar plot by condition: Customizing

```
barplot(table(dat$arrangement,dat$condition),
legend=T,xlim=c(0,6),args.legend=list(x=6))
```



R Basics

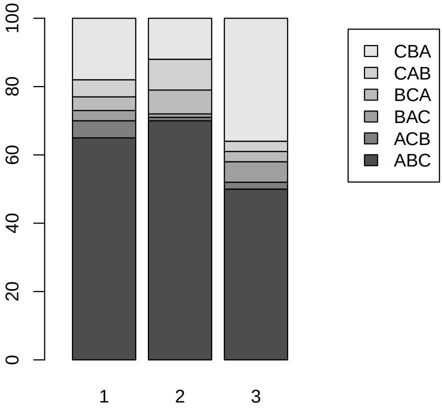
- Create a vector of elements

```
colors <- c("black","red")
values <- c(0,6)
```

Bar plot by condition: Customizing

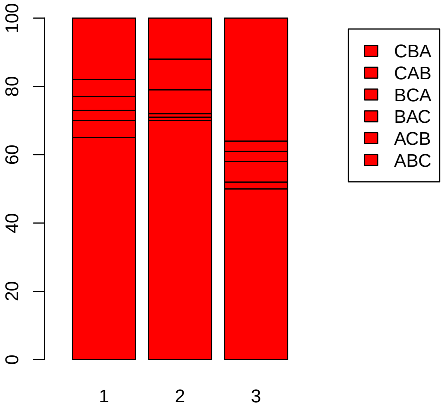
- More flexibility

```
len <- length(unique(dat$condition))
barplot(table(dat$arrangement,dat$condition),
legend=T,xlim=c(0,len+3),args.legend=list(x=len+3))
```



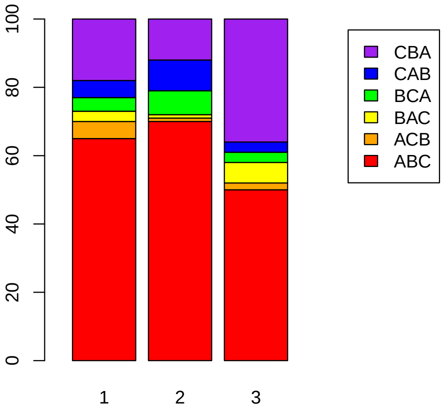
Bar plot by condition: Customizing

```
len <- length(unique(dat$condition))
barplot(table(dat$arrangement,dat$condition),
legend=T,xlim=c(0,len+3),args.legend=list(x=len+3),
col="red")
```



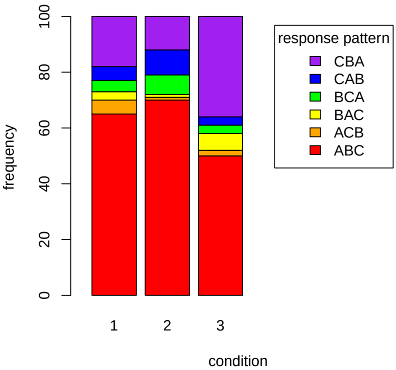
Bar plot by condition: Customizing

```
len <- length(unique(dat$condition))
barplot(table(dat$arrangement,dat$condition),
legend=T,xlim=c(0,len+3),args.legend=list(x=len+3),
col=c("red","orange","yellow","green","blue","purple"))
```



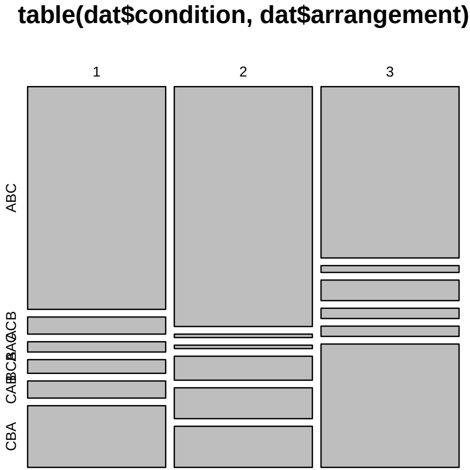
Bar plot by condition: Customizing

```
len <- length(unique(dat$condition))
barplot(table(dat$arrangement,dat$condition),
legend=T,xlim=c(0,len+4),
args.legend=list(x=len+4,title="response pattern"),
col=c("red","orange","yellow","green","blue","purple"),
xlab="condition",ylab="frequency")
```



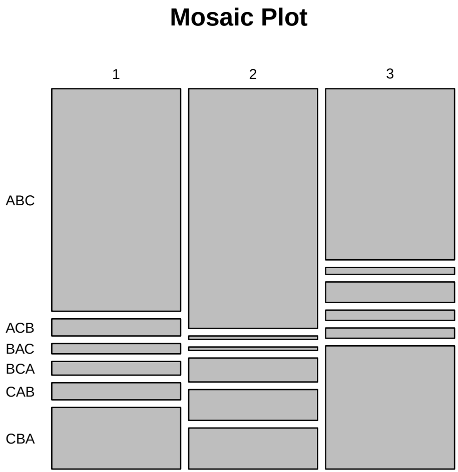
Mosaic Plot

```
mosaicplot(table(dat$condition,dat$arrangement))
```



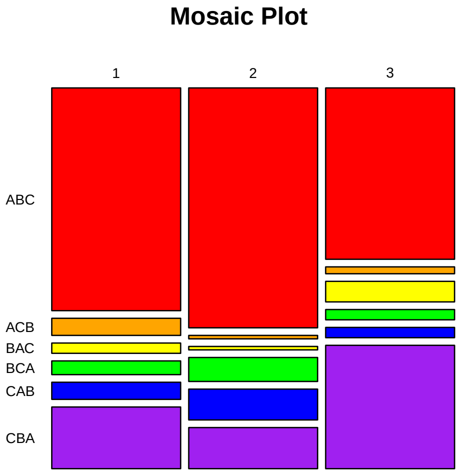
Mosaic Plot: Prettier

```
mosaicplot(table(dat$condition,dat$arrangement),
main="Mosaic Plot",las=1)
```



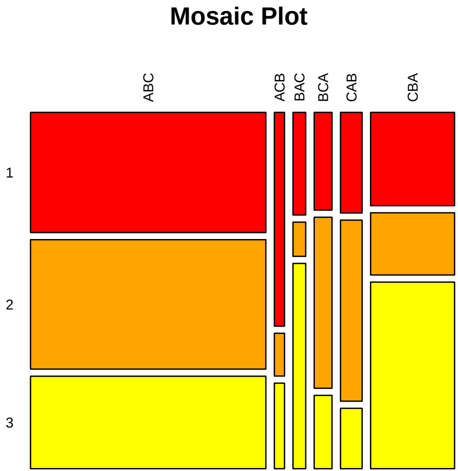
Mosaic Plot: Customizing

```
mosaicplot(table(dat$condition,dat$arrangement),
main="Mosaic Plot",las=1,
col=c("red","orange","yellow","green","blue","purple"))
```



Mosaic Plot: Turning it around

```
mosaicplot(table(dat$arrangement,dat$condition),
main="Mosaic Plot",las=2,
col=c("red","orange","yellow","green","blue","purple"))
```



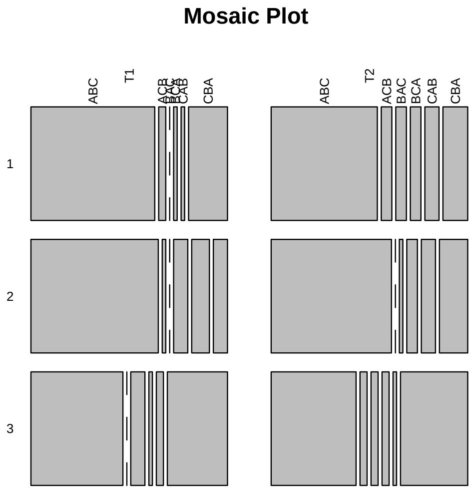
Mosaic Plot

- Mosaic Plots are nice for visualising multi-dimensional frequency data
- Let's include the time (pre vs. post) first

Mosaic Plot: More Dimensions

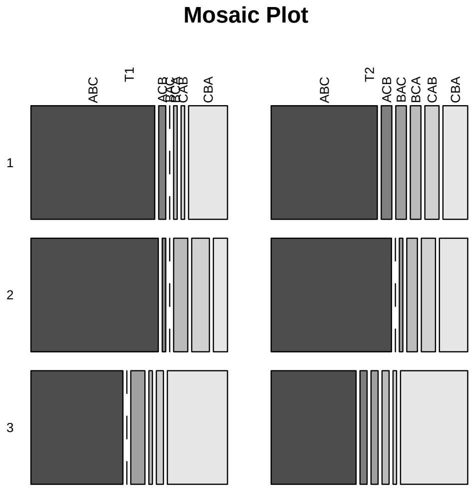
- Mosaic Plot including Time

```
mosaicplot(table(dat$time,dat$arrangement,dat$condition),
main="Mosaic Plot",las=2)
```



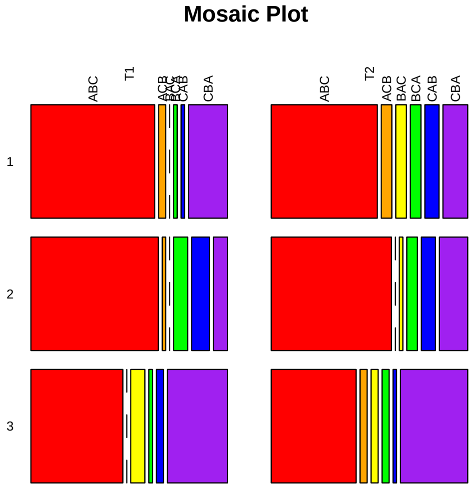
Mosaic Plot: Customizing

- `mosaicplot(table(dat$time,dat$arrangement,dat$condition), main="Mosaic Plot",las=2,col=TRUE)`



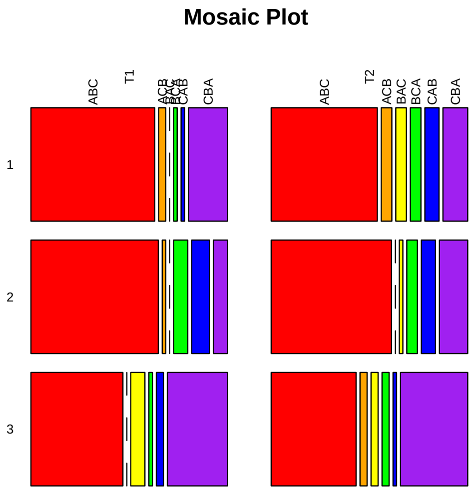
Mosaic Plot: Customizing

- `mosaicplot(table(dat$time,dat$arrangement,dat$condition), main="Mosaic Plot",las=2, col=c("red","orange","yellow","green","blue","purple"))`



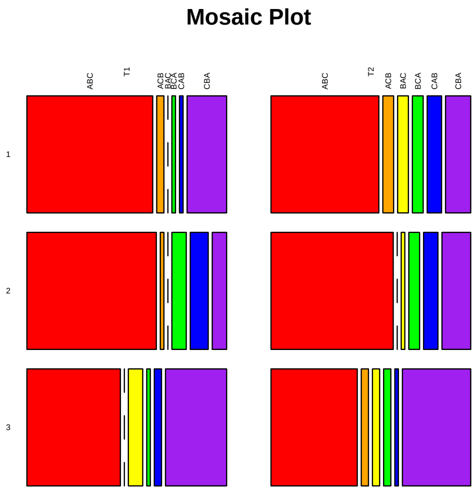
Mosaic Plot: Customizing

- `mosaicplot(table(dat$time,dat$arrangement,dat$condition), main="Mosaic Plot",las=2, col=c("red","orange","yellow","green","blue","purple"))`



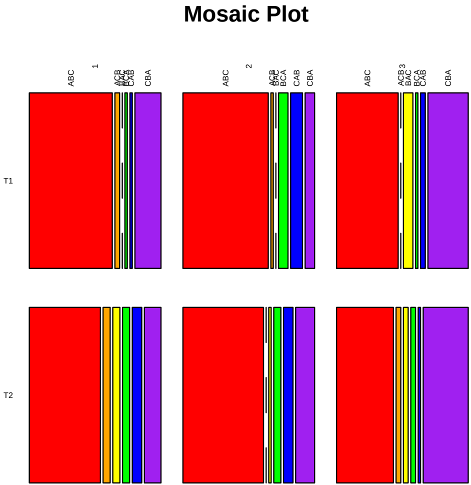
Mosaic Plot: Customizing

- `mosaicplot(table(dat$time,dat$arrangement,dat$condition), main="Mosaic Plot",las=2,cex=.4, col=c("red","orange","yellow","green","blue","purple"))`



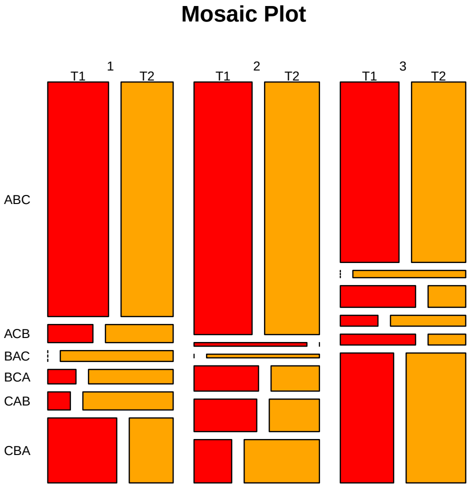
Mosaic Plot: Re-Order Variables

```
mosaicplot(table(dat$arrangement,dat$time,dat$condition),
main="Mosaic Plot",las=2,cex=.4,
col=c("red","orange","yellow","green","blue","purple"))
```



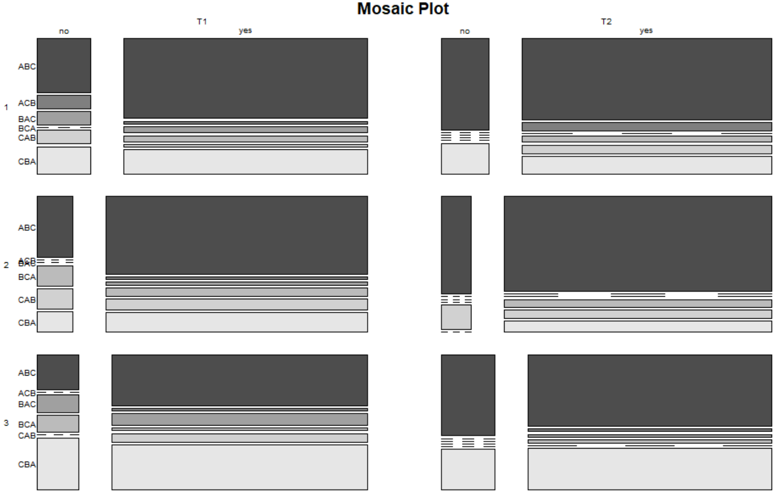
Mosaic Plot: Re-Order Variables

```
mosaicplot(table(dat$arrangement,dat$condition,dat$time),
main="Mosaic Plot",las=1,
col=c("red","orange","yellow","green","blue","purple"))
```



Mosaic Plot: Even more dimensions

```
mosaicplot(
table(dat$time,dat$condition,dat$true,dat$arrangement),
main="Mosaic Plot",las=1,cex=.6,col=T)
```



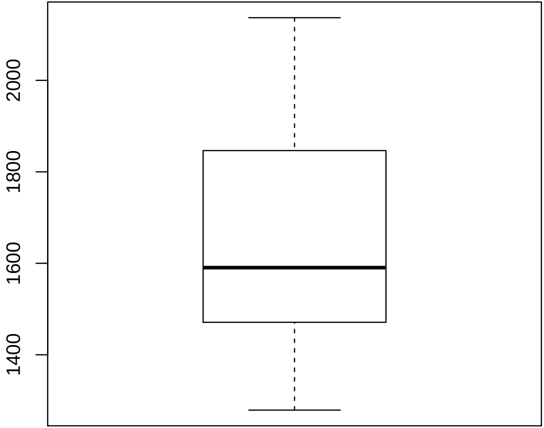
Continuous (Metric) Data:
Frequencies and Distributions

Continuous Data

- *Discrete Data* refers to cases where we have an infinite, non-countable number of possible values
- Examples: response times, N400-amplitudes, gaze durations
- In practice (but not from a theoretical point of view!), the line between discrete and continuous data can become blurry: ratings on a 1–100 scale, error rates computed from a large number of trials

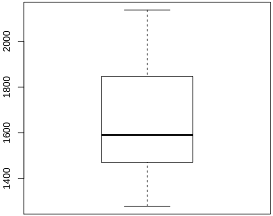
Box Plot of response times

```
boxplot(dat$RT)
```



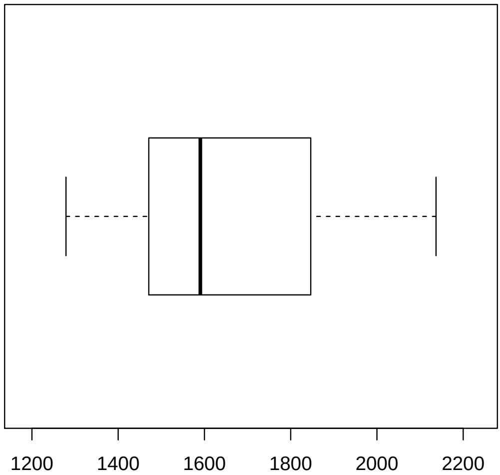
Box Plot of response times

- What can I see in a box plot?
- Outer lines: minimum and maximum value
- Thick middle line: median (50% of values below this point)
- Outer edges of the box: 1st and 3rd quartile (25% / 75% of values below these points)



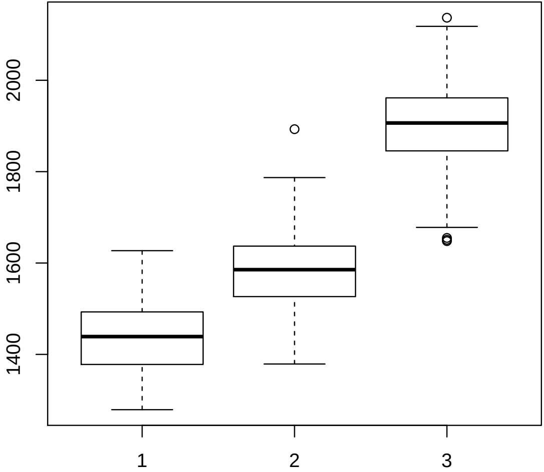
Box Plot: Turning it around

```
boxplot(dat$RT,horizontal=T)
```



Box Plot by condition

```
boxplot(RT ~ condition,dat)
```

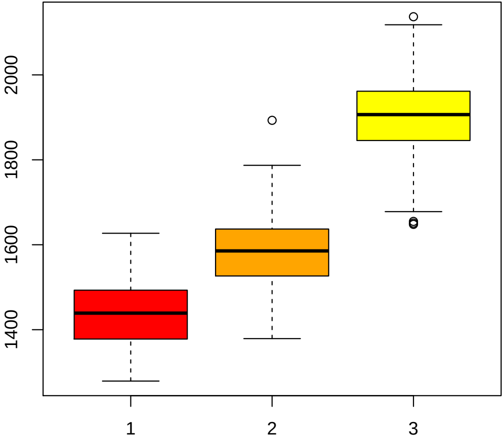


R Basics

- The $\sim$ symbol ("tilde") is used in a `formula` object
- Read
  `RT ~ condition`
  as "`RT` predicted by `condition`"

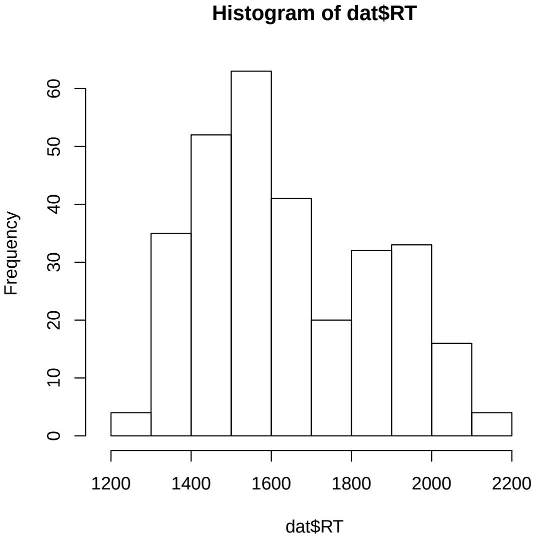
Box Plot: Customizing

```
boxplot(RT ~ condition,dat,
col= c("red","orange","yellow"))
```



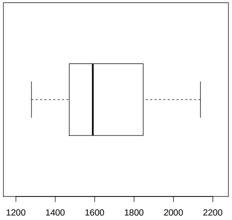
Histogram of response times

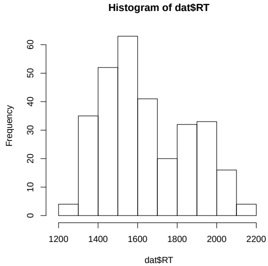
```
hist(dat$RT)
```



Histogram and Box Plot

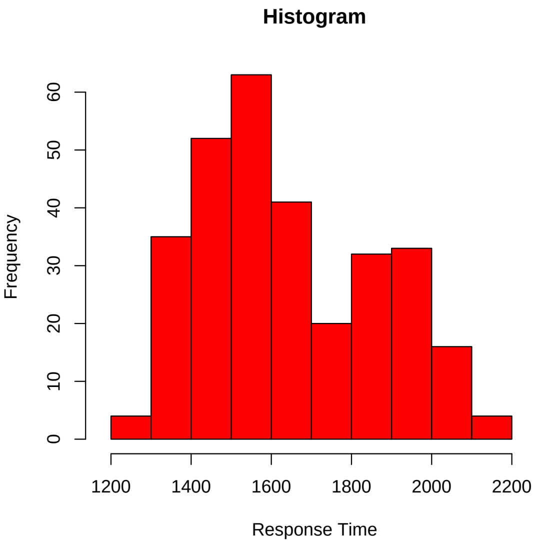
- A box plot is a histogram "as seen from above"





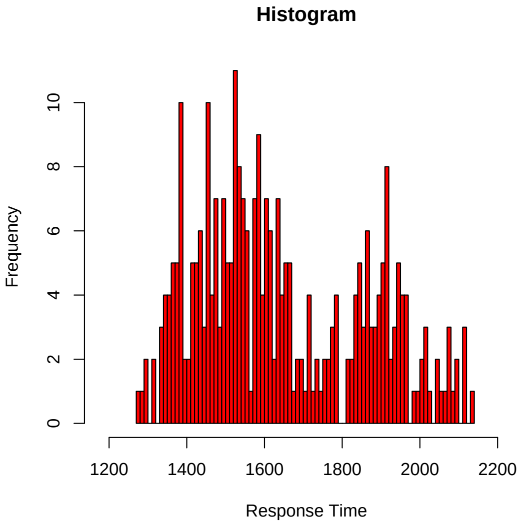
Histogram: Customizing

```
hist(dat$RT,main="Histogram",xlab="Response Time", col="red")
```



Histogram: Customizing

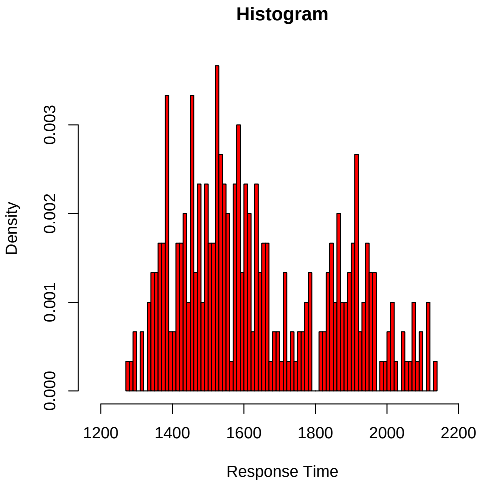
```
hist(dat$RT,main="Histogram",xlab="Response Time",
col="red",breaks=100)
```



Histogram: Customizing

- Density instead of frequency

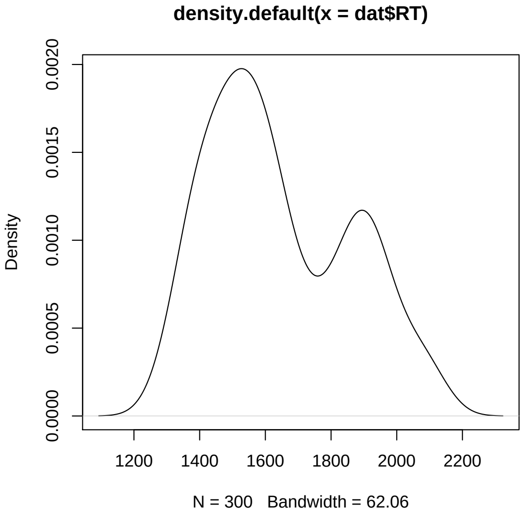
```
hist(dat$RT,main="Histogram",xlab="Response Time",
col="red",breaks=100,freq=F)
```



Kernel Density Plot

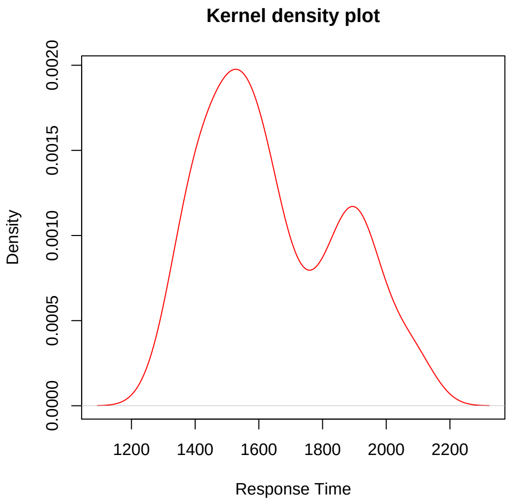
- ("Smoothed Histograms")

```
plot(density(dat$RT))
```



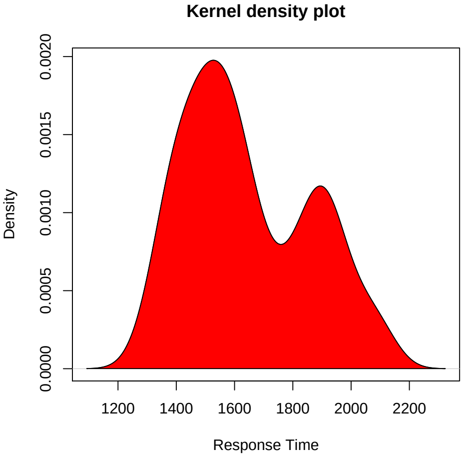
Kernel Density Plot: Customizing

```
plot(density(dat$RT),
main="Kernel density plot",xlab="Response Time",col="red")
```



Kernel Density Plot: Customizing

```
d <- density(dat$RT)
plot(d,main="Kernel density plot",xlab="Response Time")
polygon(d,col="red")
```

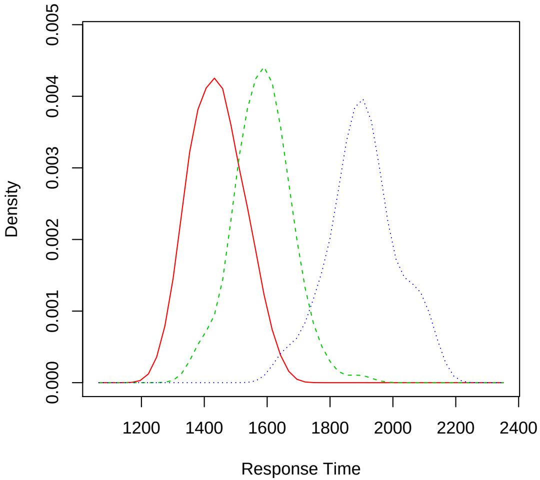


Kernel Density Plot by condition

- First install the sm package
  ```
  install.packages("sm")
  library(sm)
  ```
- If you don't know which functions a package includes, use
  ```
  help(package="sm")
  ```

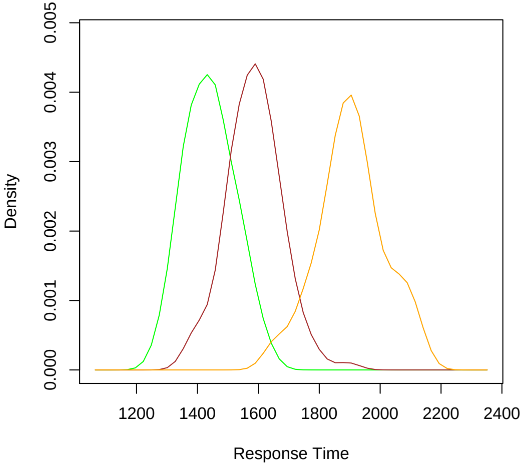
Kernel Density Plot by condition

```
sm.density.compare(dat$RT, dat$condition,xlab="Response
Time")
```



Kernel Density Plot by condition

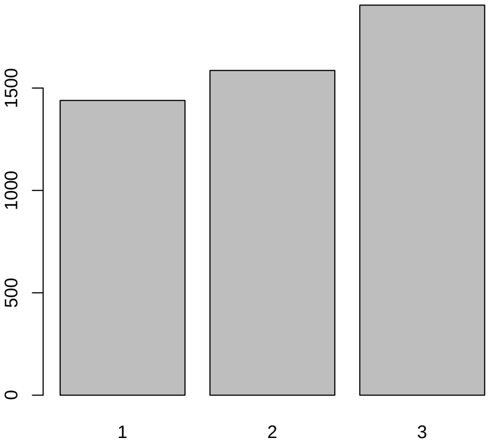
```
sm.density.compare(dat$RT, dat$condition,xlab="Response
Time",
lty=c(1,1,1),col=c("green","brown","orange"))
```



Continuous (Metric) Data:
Means and Deviations

Bar Plot of means

```
m <- aggregate(RT ~ condition,dat,mean)
barplot(m$RT,names.arg=m$condition)
```

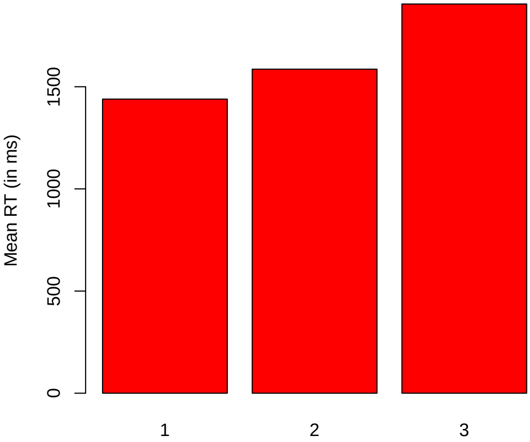


R Basics

- The aggregate() splits the data into subsets and performs a given operation on all subsets individually
- `aggregate(RT ~ condition,dat,mean)` splits dat by condition, and then applies the mean() function to the RT column
- The data can be split over several variables at the same time: `aggregate(RT ~ condition + time,dat,mean)`

Bar Plot of means: Customizing

```
m <- aggregate(RT ~ condition,dat,mean)
barplot(m$RT,names.arg=m$condition,
col="red",xlab="Condition",ylab="Mean RT (in ms)")
```



Bar Plot of means: Error Bars

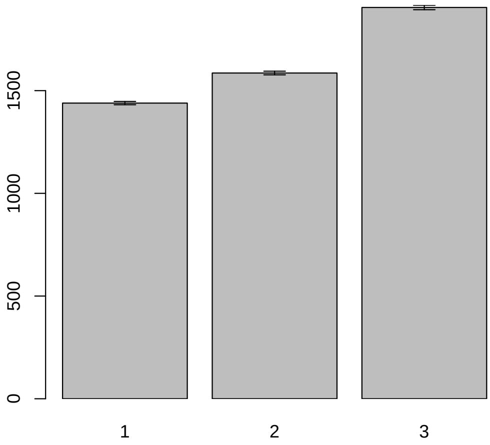
- Installing and loading the `sciplot` package

```
install.packages("sciplot")
library(sciplot)
```

- Package included the `bargraph.CI()` function

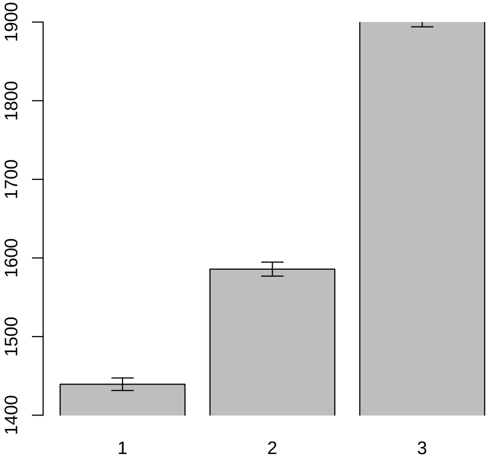
Bar Plot of means: Error Bars

```
bargraph.CI(x.factor=dat$condition,response=dat$RT)
```



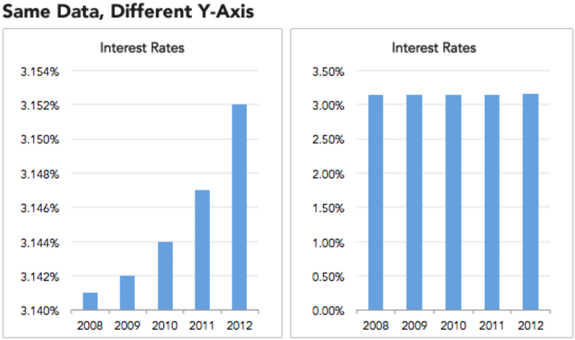
Bar Plot of means: Adjusting the y-axis

```
bargraph.CI(x.factor=dat$condition,response=dat$RT,
ylim=c(1400,1900))
```



Adjusting the y-axis

- Adjusting the y-axis is a great way to misrepresent your data and mislead your audience: https://heap.io/blog/data-stories/how-to-lie-with-data-visualization



Bar Plot of means: Error Bars

- One main purpose of error bars is to provide at leat *some* reference frame

Bar Plot of means: Error Bars

- One main purpose of error bars is to provide at leat *some* reference frame
- Another purpose is of course to indicate the variability of data, which is critical when it comes to the statistical testing for effects

Bar Plot of means: Error Bars

- One main purpose of error bars is to provide at leat *some* reference frame
- Another purpose is of course to indicate the variability of data, which is critical when it comes to the statistical testing for effects
- However, in many cases, it's not completely clear *which* error bars should be used

Bar Plot of means: Error Bars

- Moreover, errors bars are also criticized: http://biostat.mc.vanderbilt.edu/wiki/Main/DynamitePlots

Bar Plot of means: Error Bars

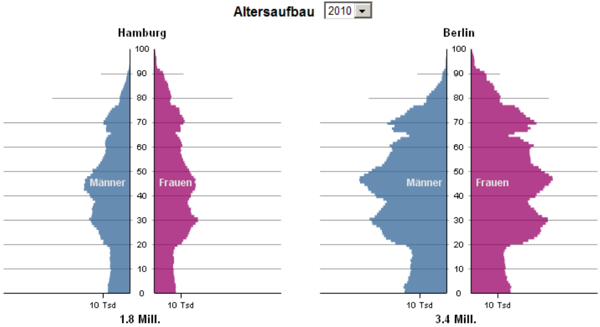
- Moreover, errors bars are also criticized: http://biostat.mc.vanderbilt.edu/wiki/Main/DynamitePlots
- We will deal with these issues later

Bar Plot of means: Error Bars

- At this point, the Box Plot by conditions might be one of the most "honest" ways to display the data

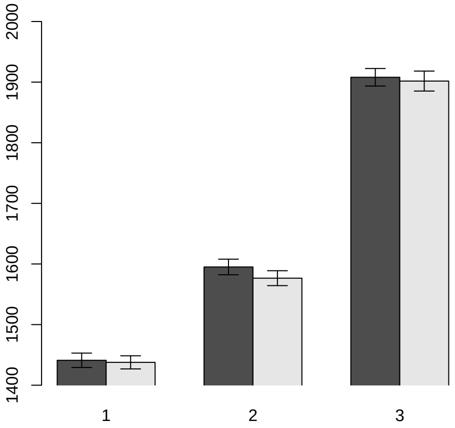
Bar Plot of means: Error Bars

- At this point, the Box Plot by conditions might be one of the most “honest” ways to display the data
- Something like vertical histograms might be even better, but they need some coding in R (which is why we won’t deal with them here)



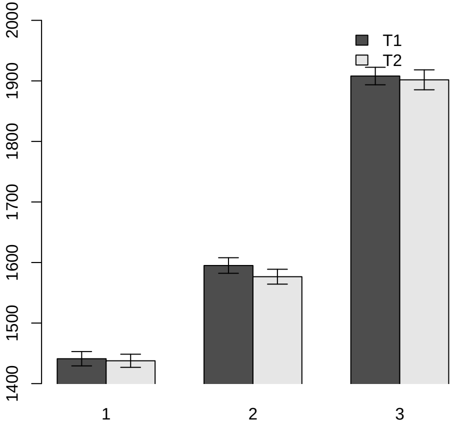
Bar Plot of means: Two-factorial

- Include a second factor in the plots:
  `bargraph.CI(x.factor=dat$condition,group=dat$time, response=dat$RT,ylim=c(1400,2000))`



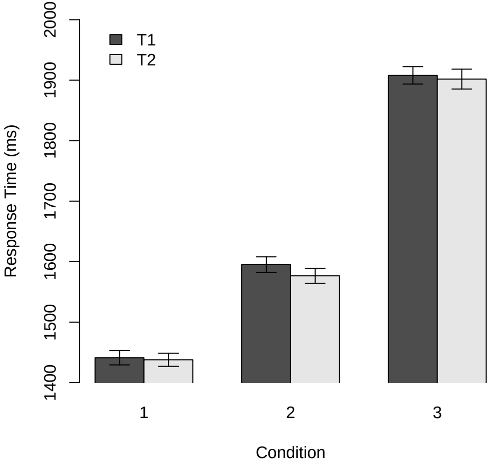
Bar Plot of means: Two-factorial with legend

```
bargraph.CI(x.factor=dat$condition,group=dat$time,
response=dat$RT,ylim=c(1400,2000),legend=T)
```



Bar Plot of means: Customize

```
bargraph.CI(x.factor=dat$condition,group=dat$time,
response=dat$RT,ylim=c(1400,2000),legend=T,
x.leg=1,xlab="Condition",ylab="Response Time (ms)")
```



Line Plot of means

- In many publications, you will see Line Plots instead of Bar Plots to display the mean values and standard error per condition

Line Plot of means

- In many publications, you will see Line Plots instead of Bar Plots to display the mean values and standard error per condition
- This is mostly convention, but it can be justified

Line Plot of means

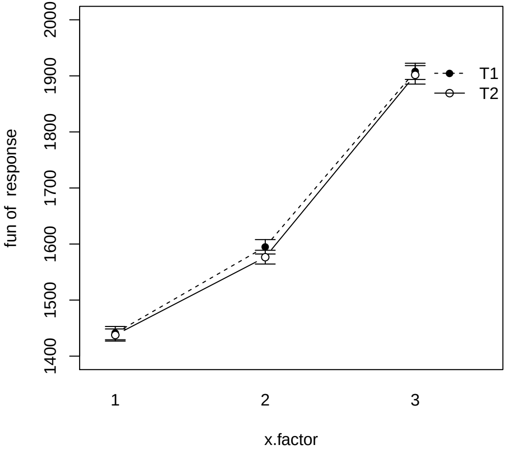
- In many publications, you will see Line Plots instead of Bar Plots to display the mean values and standard error per condition
- This is mostly convention, but it can be justified
- The only thing that matters for a Bar Plot is their height; however, there are more (unnecessary) dimensions on display (width, area)

Line Plot of means

- In many publications, you will see Line Plots instead of Bar Plots to display the mean values and standard error per condition
- This is mostly convention, but it can be justified
- The only thing that matters for a Bar Plot is their height; however, there are more (unnecessary) dimensions on display (width, area)
- Sometimes, the area can be informative, and here it can get confusing

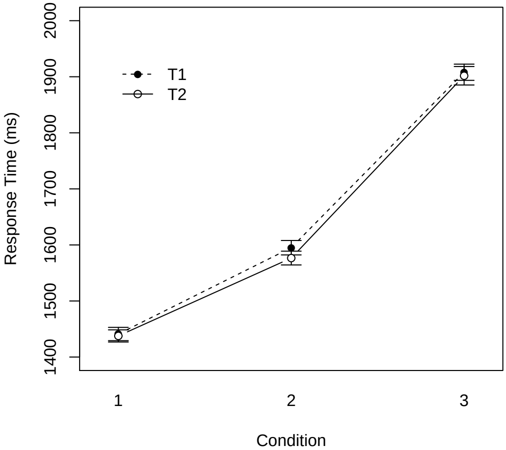
Line Plot of means

```
lineplot.CI(x.factor=dat$condition,group=dat$time,
response=dat$RT,ylim=c(1400,2000))
```



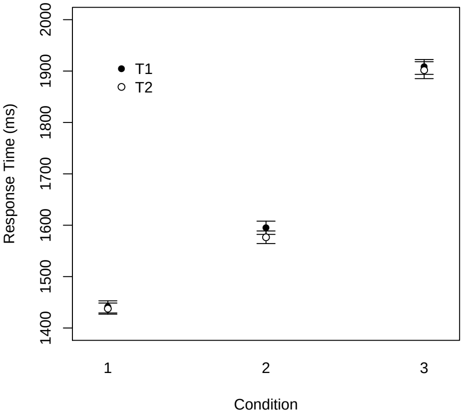
Line Plot of means: Customize

```
lineplot.CI(x.factor=dat$condition,group=dat$time,
response=dat$RT,ylim=c(1400,2000),legend=T,
x.leg=1,xlab="Condition",ylab="Response Time (ms)")
```



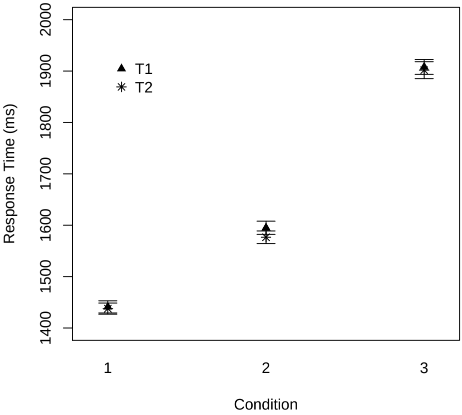
Line Plot of means: Customize

```
lineplot.CI(x.factor=dat$condition,group=dat$time,
response=dat$RT,ylim=c(1400,2000),legend=T,
x.leg=1,xlab="Condition",ylab="Response Time (ms)",
type="p")
```

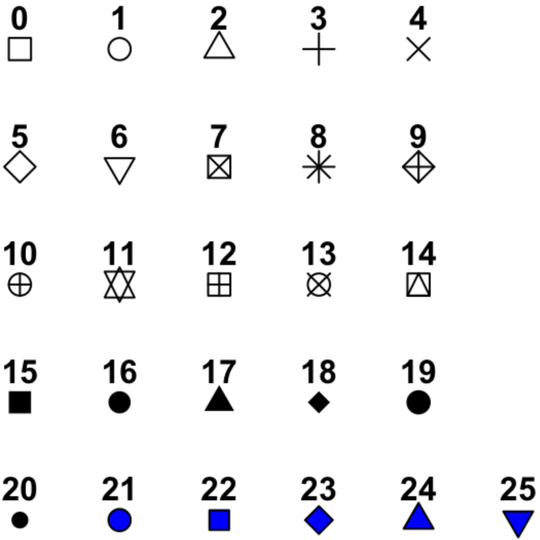


Line Plot of means: Customize

```
lineplot.CI(x.factor=dat$condition,group=dat$time,
response=dat$RT,ylim=c(1400,2000),legend=T,
x.leg=1,xlab="Condition",ylab="Response Time (ms)",
type="p",pch=c(17,8))
```



Points in R: The `pch` option

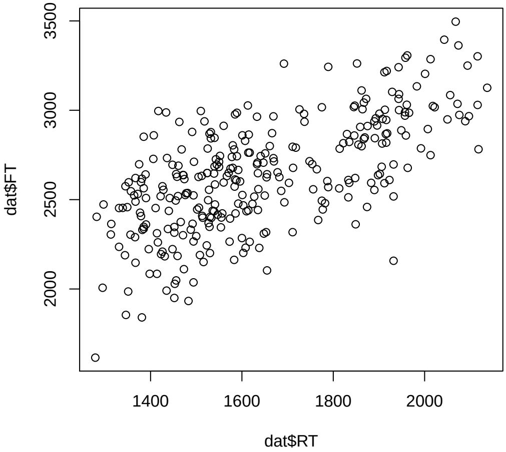


Relations between Variables

- We have discussed plots of multi-dimensional data before:
  - Multiple discrete variables: stacked Bar Plots, Mosaic Plots, overlapping Kernel Density Plots
  - Multiple discrete + 1 continuous variable: Bar/Line Plots by condition
- Now we turn to cases with multiple continuous variables

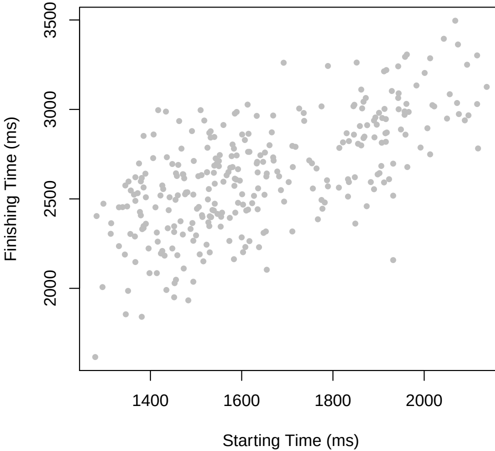
Scatter Plot

```
plot(dat$RT,dat$FT)
```



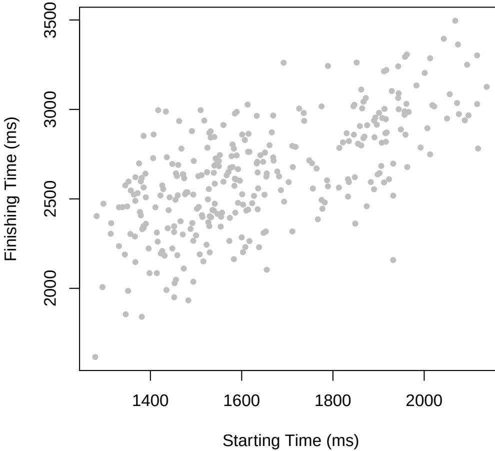
Scatter Plot: Customize

```
plot(dat$RT,dat$FT,
xlab="Starting Time (ms)",ylab="Finishing Time (ms)",
pch=20,col="grey")
```



Scatter Plot: Alternative command

```
plot(FT ~ RT, data = dat,
xlab="Starting Time (ms)",ylab="Finishing Time (ms)",
pch=20,col="grey")
```



Scatter Plot by Condition

- We now make a first step in the direction of step-wise plotting
- General procedure: Create a plot containing the points for one condition, then add the points for the other conditions in a different color

R Basics: Indexing

- See Introduction: data frames can be indexed using the `[,]` square brackets
  `dat[1,]` extracts the first row

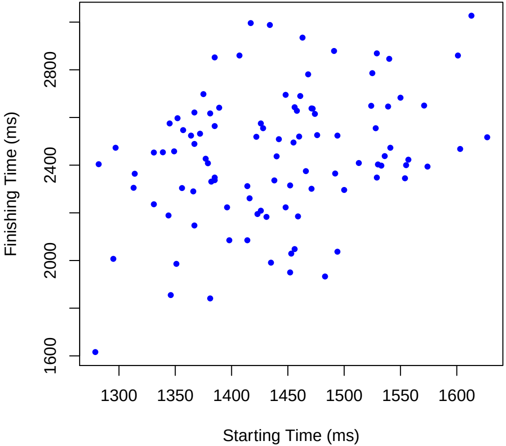
- Create an index that only extracts a certain factor level:
  `dat[dat$condition == 1,]`

- Logical operators in R:

`==`	equal to
`!=`	not equal to
`<` or `>`	smaller/greater than
`<=` or `>=`	smaller/greater or equal
`&`	element-wise AND
`&&`	AND
`|`	element-wise OR
`||`	OR
`%in%`	included in
`!(X)` (where X is another statement)	NOT

Scatter Plot by condition

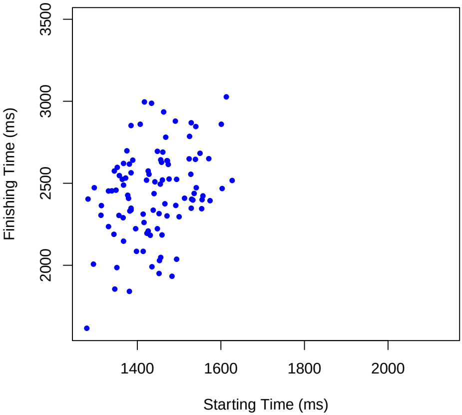
```
plot(FT ~ RT, data = dat[dat$condition == 1,],
xlab="Starting Time (ms)",ylab="Finishing Time (ms)",
pch=20,col="blue")
```



Scatter Plot by condition

- Ensure that the axes are sufficiently long to display all data

```
plot(FT ~ RT, data = dat[dat$condition == 1,],
xlab="Starting Time (ms)",ylab="Finishing Time (ms)",
pch=20,col="blue",xlim=range(dat$RT),ylim=range(dat$FT))
```

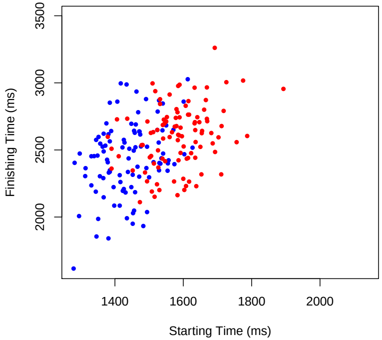


Scatter Plot by condition

- Add the points for condition 2

```
plot(FT ~ RT, data = dat[dat$condition == 1,],
xlab="Starting Time (ms)",ylab="Finishing Time (ms)",
pch=20,col="blue",ylim=range(dat$FT))
```

```
points(FT ~ RT,data=dat[dat$condition==2,],
pch=20,col="red")
```



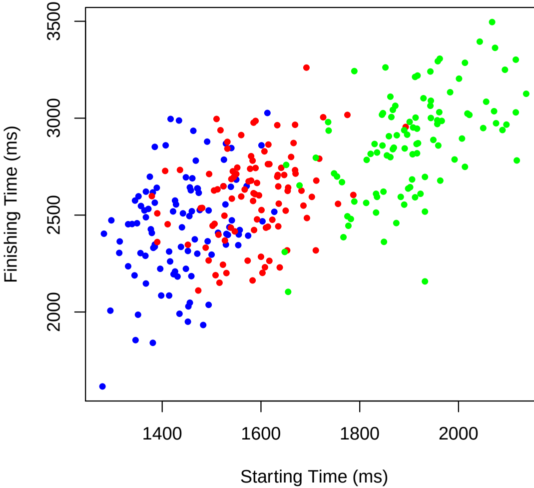
Scatter Plot by condition

- Add the points for condition 3

```
plot(FT ~ RT, data = dat[dat$condition == 1,],
xlab="Starting Time (ms)",ylab="Finishing Time (ms)",
pch=20,col="blue",ylim=range(dat$FT))

points(FT ~ RT,data=dat[dat$condition==2,],
pch=20,col="red")

points(FT ~ RT,data=dat[dat$condition==3,],
pch=20,col="green")
```

Scatter Plot by condition

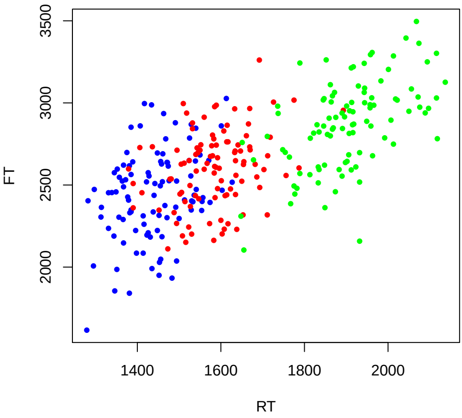
- Add the points for condition 3



Scatter Plot by condition

- Another (maybe simpler) method:

```
cols <- c("blue","red","green")
cols2 <- cols[as.numeric(dat$condition)]
plot(FT ~ RT,data=dat,col=cols2,pch=20)
```



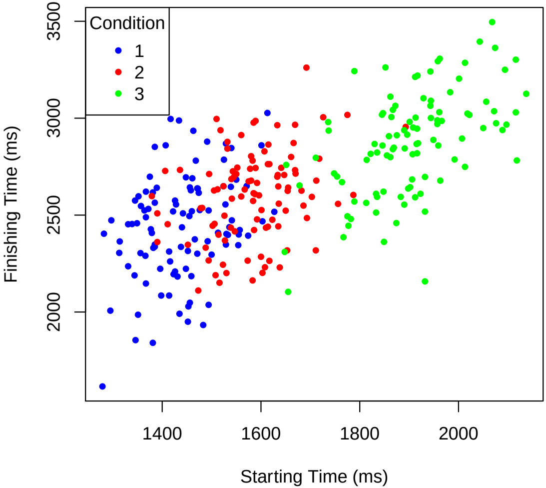
Scatter Plot by condition

- Add a legend

```
plot(FT ~ RT, data = dat[dat$condition == 1,],
xlab="Starting Time (ms)",ylab="Finishing Time (ms)",
pch=20,col="blue",ylim=range(dat$FT))

points(FT ~ RT,data=dat[dat$condition==2,],
pch=20,col="red")

points(FT ~ RT,data=dat[dat$condition==3,],
pch=20,col="green")

legend(x ="topleft",legend=c(1,2,3),
col=c("blue","red","green"),pch=20,title="Condition")
```

Scatter Plot by condition

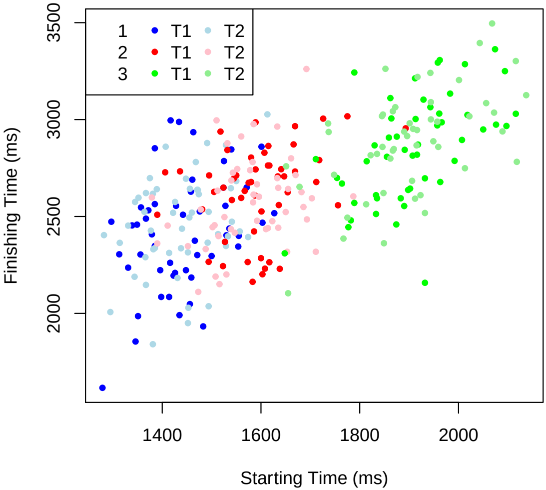
- Add a legend



Scatter Plot by two conditions: An example

```
plot(FT ~ RT, data=dat[dat$condition==1 & dat$time=="T1",],
xlab="Starting Time (ms)",ylab="Finishing Time (ms)",
pch=20,col="blue",ylim=range(dat$FT),xlim=range(dat$RT))
points(FT ~ RT, data=dat[dat$condition==1 &
dat$time=="T2",],pch=20,col="lightblue")

points(FT ~ RT,data=dat[dat$condition==2 &
dat$time=="T1",],pch=20,col="red")
points(FT ~ RT,data=dat[dat$condition==2 &
dat$time=="T2",],pch=20,col="pink")

points(FT ~ RT,data=dat[dat$condition==3 &
dat$time=="T1",],pch=20,col="green")
points(FT ~ RT,data=dat[dat$condition==3 &
dat$time=="T2",],pch=20,col="lightgreen")

legend(x="topleft",legend=c(1,2,3,rep("T1",3),rep("T2",3))
,col=c(rep("white",3),"blue","red","green",
"lightblue","pink","lightgreen"),pch=20,ncol=3)
```

Scatter Plot by two conditions: An example

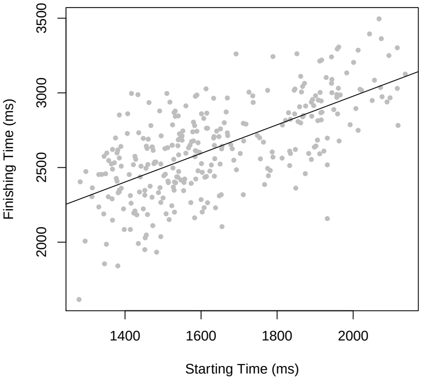


Linear Regression

- Regression: Predict one value with another value (or a set of other values)
- Linear Regression: $y = b \cdot x + a + \epsilon$,
  with ϵ being an unsystematic error
- Estimate a and b by minimizing the deviation between predicted and actual values

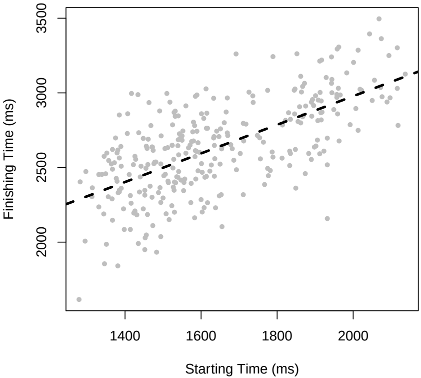
Linear Regression

```
plot(FT ~ RT,data=dat,
xlab="Starting Time (ms)",ylab="Finishing Time (ms)",
pch=20,col="grey")
regr <- lm(FT ~ RT,data=dat)
abline(regr)
```

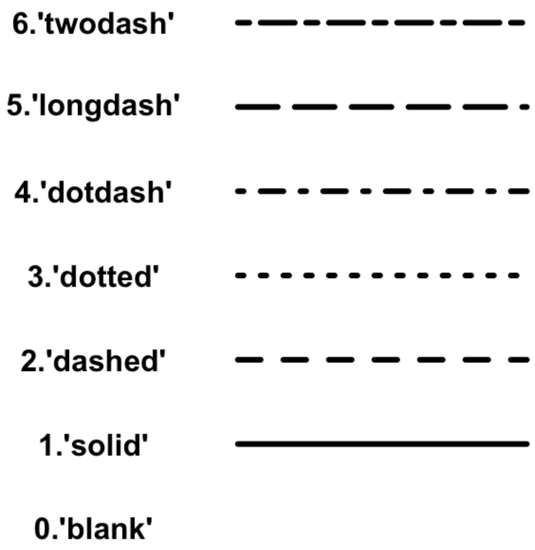


Linear Regression: Customize

```
plot(FT ~ RT,data=dat,
xlab="Starting Time (ms)",ylab="Finishing Time (ms)",
pch=20,col="grey")
regr <- lm(FT ~ RT,data=dat)
abline(regr,lty=2,lwd=3)
```



Lines in R: The `lty` option



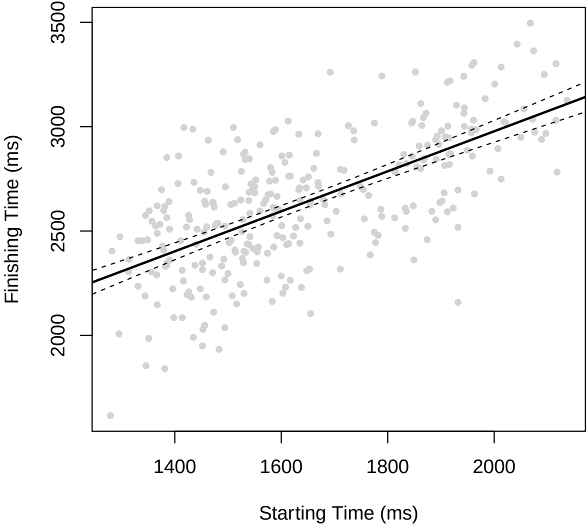
Linear Regression: Confidence Intervals (the long way)

- You might want to add some indication about the confidence of your prediction: A confidence interval around the predicted values

- Long script:

```
plot(FT ~ RT,data=dat,
xlab="Starting Time (ms)",ylab="Finishing Time (ms)",
pch=20,col="lightgrey")
regr <- lm(FT ~ RT,data=dat)
abline(regr,lwd=2)
newdat <- seq(min(dat$RT)-50,max(dat$RT)+50,length.out=10000)
CI <- predict(regr, newdata=data.frame(RT=newdat),
interval="confidence", level = 0.95)
matlines(newdat, CI[,2:3], lty=2,col="black")
```

Linear Regression: Confidence Intervals (the long way)

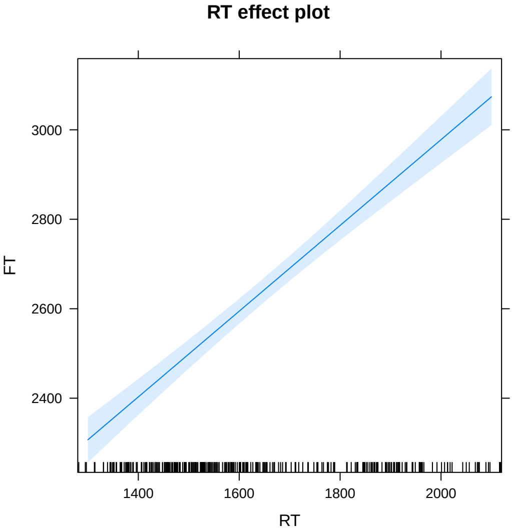


Linear Regression: Confidence Intervals (the short way)

- Use the `effects` package `install.packages("effect")` `library(effects)`

Linear Regression: Confidence Intervals (the short way)

```
regr <- lm(FT ~ RT,data=dat)
plot(effect("RT",regr))
```

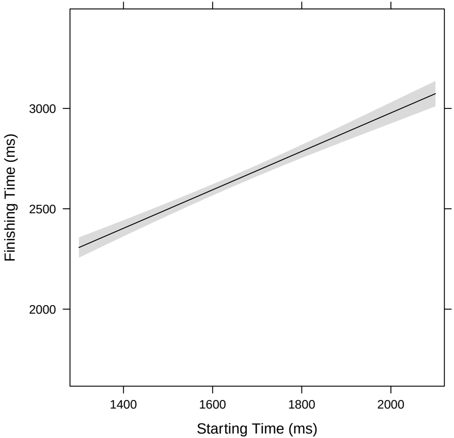


Linear Regression: Customize

- The `plot.effect` command (called when using `plot(effect(...))`) has *a lot* of options
- These are arranged into several clusters, and each cluster can be specified using a `list`
- See the help function at `?plot.effect`

Linear Regression: Customize

```
regr <- lm(FT ~ RT,data=dat)
plot(effect("RT",regr),ylim=range(dat$FT),
xlab="Starting Time (ms)",ylab="Finishing Time (ms)",main="",
lines=list(col="black"),axes=list(ylim=range(dat$FT)),rug=F)
```



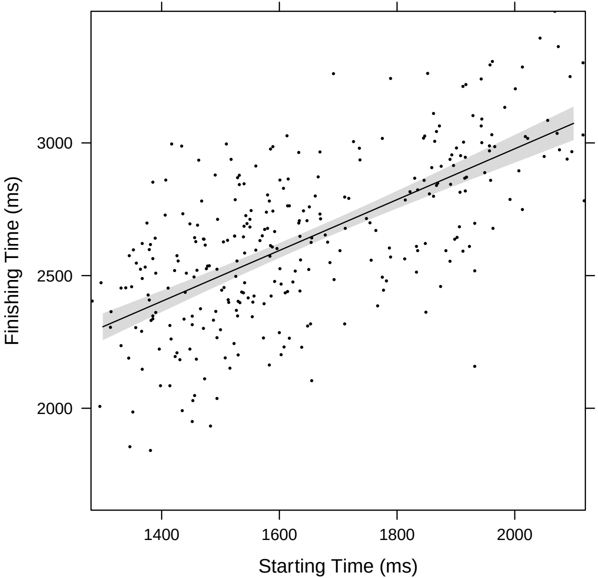
Linear Regression: Customize

- Adding points takes a bit of a workaround with the `lattice` package

```
install.packages("lattice")
library(lattice)

regr <- lm(FT   RT,data=dat)
plot(effect("RT",regr),ylim=range(dat$FT),
xlab="Starting Time (ms)",ylab="Finishing Time (ms)",main="",
lines=list(col="black"),axes=list(ylim=range(dat$FT)),rug=F)
trellis.focus("panel", 1, 1, highlight=F)
panel.points(dat$RT, dat$FT,pch=20,col="black",cex=.3)
trellis.unfocus()
```

- There are simpler options using the ggplot2 package

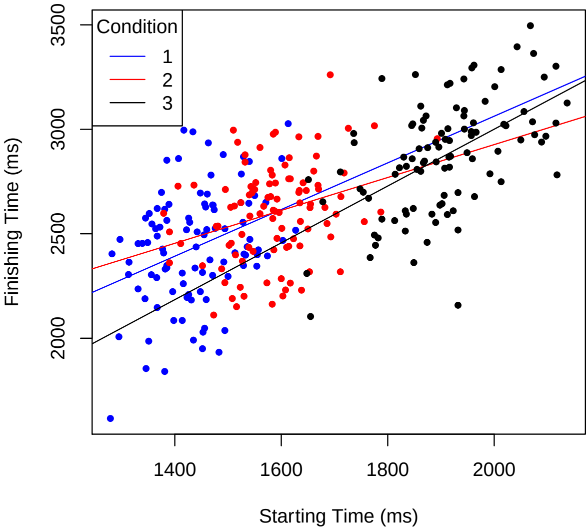
Linear Regression: Customize



Linear Regression by condition

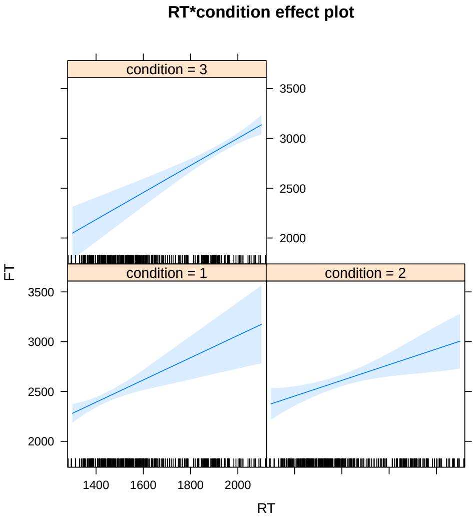
```
plot(FT ~ RT,data=dat[dat$condition==1,],
xlab="Starting Time (ms)",ylab="Finishing Time (ms)",
pch=20,col="blue",ylim=range(dat$FT),xlim=range(dat$RT))
abline(lm(FT ~ RT,data=dat[dat$condition==1,]),col="blue")

points(FT ~ RT,data=dat[dat$condition==2,],pch=20,col="red")
abline(lm(FT ~ RT,data=dat[dat$condition==2,]),col="red")

points(FT ~ RT,data=dat[dat$condition==3,],
pch=20,col="black")
abline(lm(FT ~ RT,data=dat[dat$condition==3,]),col="black")

legend(x ="topleft",legend=c(1,2,3),
col=c("blue","red","black"),lty=1,title="Condition")
```

Linear Regression by condition



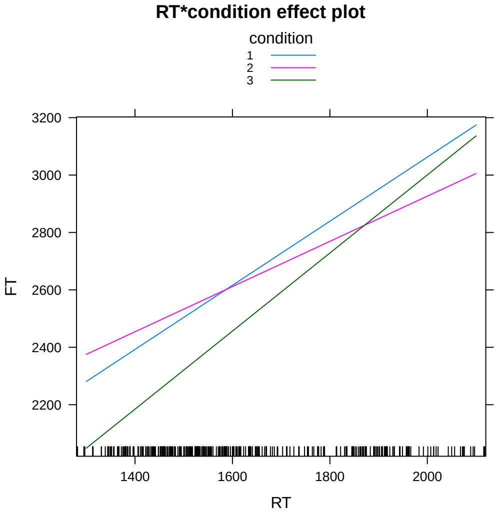
Linear Regression by condition

```
regr <- lm(FT ~ RT*condition,data=dat)
plot(effect("RT*condition",regr))
```



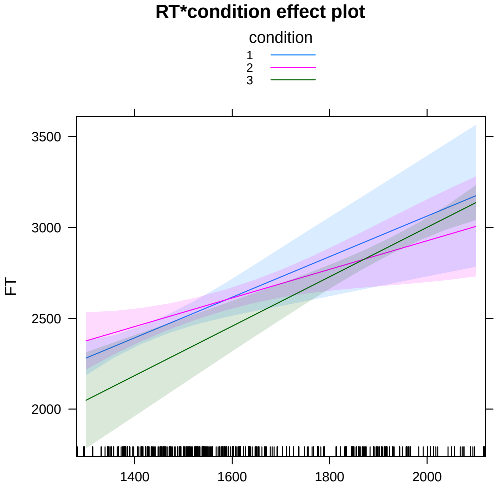
Linear Regression by condition

```
regr <- lm(FT ~ RT*condition,data=dat)
plot(effect("RT*condition",regr),lines=list(multiline=TRUE))
```



Linear Regression by condition

```
regr <- lm(FT ~ RT*condition,data=dat)
plot(effect("RT*condition",regr),lines=list(multiline=TRUE),
confint = list(style="bands"))
```



Non-linear Regression

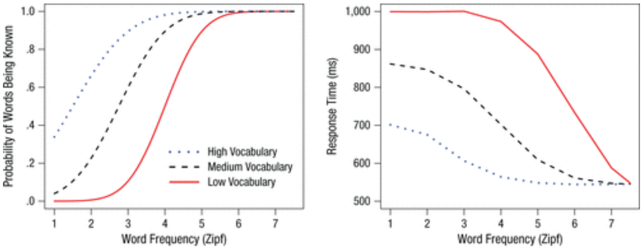
- Sometimes, the relation between two variables is not linear
- In these cases, a non-linear regression design can be helpful
- Be careful: This can increase the degrees of freedom of your analysis substantially!
- Do you have a reason to expect non-linear effects?
  (On the other hand, why should linear be the default?)

Non-linear Regression

- Sometimes, the relation between two variables is not linear
- In these cases, a non-linear regression design can be helpful
- Be careful: This can increase the degrees of freedom of your analysis substantially!
- Do you have a reason to expect non-linear effects?
(On the other hand, why should linear be the default?)

Non-linear Regression

- Example: Word Frequency Effect (Brysbaert, Mandera & Keuleers, 2017)

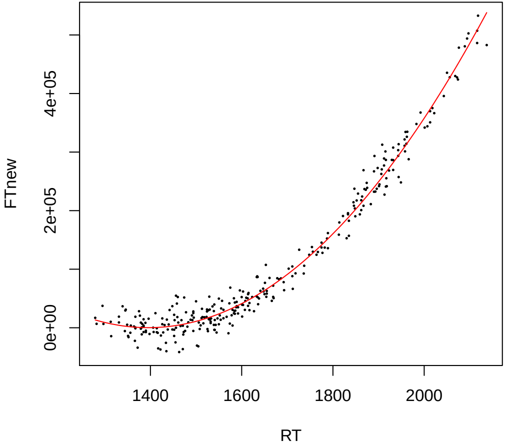


Non-linear Regression: Quadratic Regression

- Create a new column `FTnew` in your data frame which is based on `RT` raised to the power of 2, plus some noise
  ```
  dat$FTnew <- (dat$RT-1400)^2 + rnorm(nrow(dat),0,20000)
  ```
- Fit a new regression model
  ```
  regr2 <- lm(FTnew ~ poly(RT,2),data=dat)
  ```
- Also allows the use of higher-order polynomials

Non-linear Regression: Quadratic Regression

```
regr2 <- lm(FTnew ~ poly(RT,2),data=dat)
plot(FTnew ~ RT,data=dat,pch=20,cex=.3)
lines(sort(dat$RT),fitted(regr2)[order(dat$RT)],col="red")
```

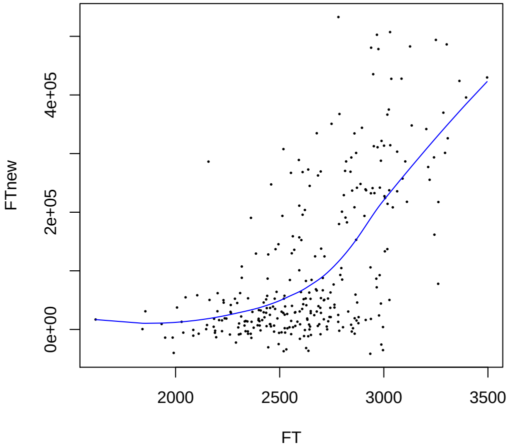


Non-linear Regression: Generalized

- With a quadratic regression (i.e., polynomial degree 2), we are committing to a specific shape of relation
- We can relax this assumption by considering generalized non-linear effects
- Fit a non-linear regression model:
  ```
  regr3 <- loess(FTnew ~ FT, data=dat)
  ```
- This function relies on local polynomial fitting

Non-linear Regression: Generalized

```
plot(FTnew   FT,data=dat,pch=20,cex=.3)
regr3 <- loess(FTnew ~ FT,data=dat)
lines(sort(dat$FT),fitted(regr3)[order(dat$FT)],col="blue")
```



Non-linear Regression: Generalized

- Another option for non-linear effects are Generalized Additive Models (GAMs) as implemented in the `mgcv` package:
  ```
  install.packages("mgcv")
  library(mgcv)
  ```

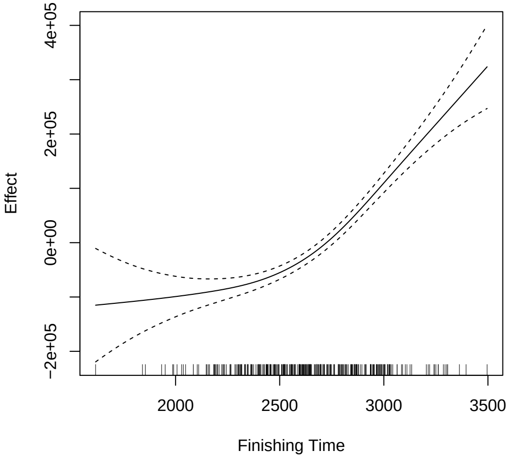
- Again, be a bit careful with non-linear effects

- Fit a GAM:
  regr4 ¡- gam(FTnew   s(FT),data=dat)

- `s()` to include a non-linear effect

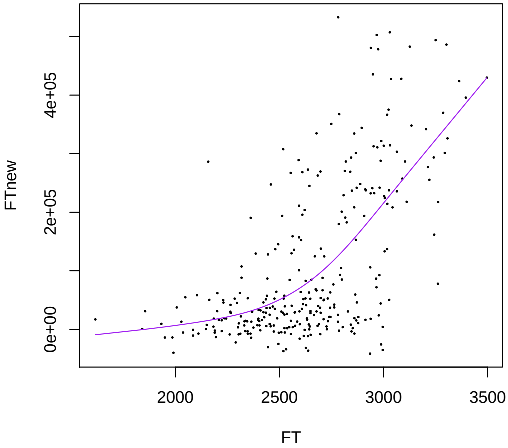
Non-linear Regression: Generalized

```
regr4 <- gam(FTnew ~ s(FT),data=dat)
plot(regr4,xlab="Finishing Time",ylab="Effect")
```



Non-linear Regression: Generalized

```
plot(FTnew ~ FT,data=dat,pch=20,cex=.3)
regr4 <- gam(FTnew ~ s(FT),data=dat)
lines(sort(dat$FT),fitted(regr4)[order(dat$FT)],col="purple")
```



Linear Regression: Continuous interactions

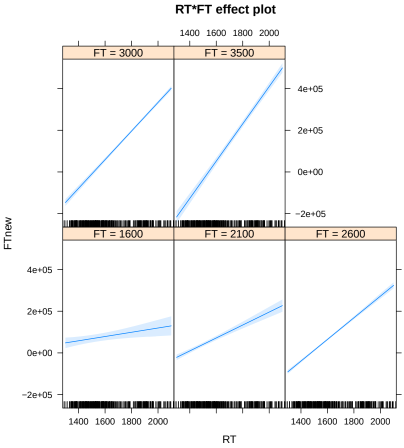
- Fit a regression model predicting FTnew from a linear interaction between RT and FT
  ```
  creg <- lm(FTnew ~ RT*FT,data=dat)
  ```
- With an interaction, the effect of one of these predictors on the outcome depends on the value of the other predictor

Linear Regression: Continuous interactions

- Option 1: "Splitting" one of the variables into discrete levels
- The easiest way of doing this employs the `effects` package

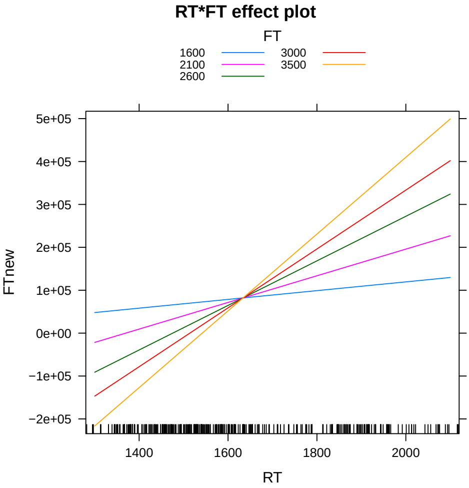
Linear Regression: Continuous interactions

```
creg1 <- lm(FTnew ~ RT*FT,data=dat)
plot(effect("RT*FT",creg1))
```



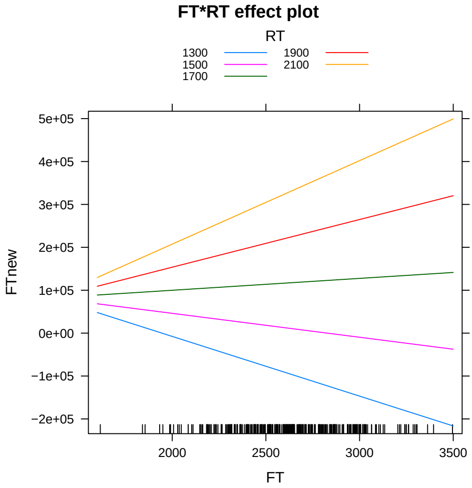
Linear Regression: Continuous interactions

```
creg1 <- lm(FTnew ~ RT*FT,data=dat)
plot(effect("RT*FT",creg1),lines=list(multilines=TRUE))
```

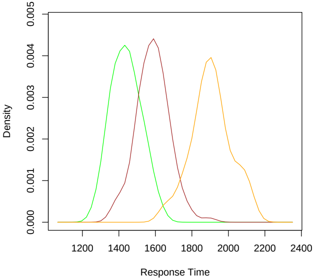


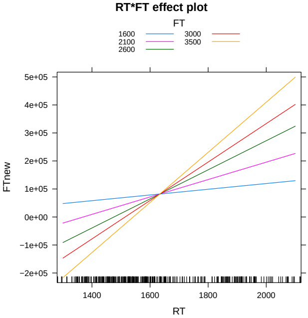
Linear Regression: Continuous interactions

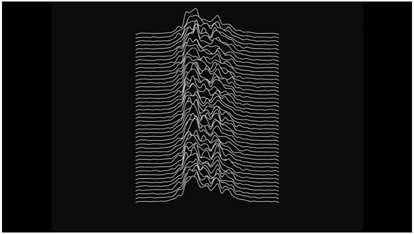
```
creg2 <- lm(FTnew ~ FT*RT,data=dat)
plot(effect("FT*RT",creg2),lines=list(multilines=TRUE))
```



3D-Plots


Density
0.000
0.001
0.002
0.003
0.004
0.005
1200 1400 1600 1800 2000 2200 2400
Response Time


RT*FT effect plot
FT
1600
2100
2600
3000
3500
FTnew
5e+05
4e+05
3e+05
2e+05
1e+05
0e+00
−1e+05
−2e+05
1400 1600 1800 2000
RT



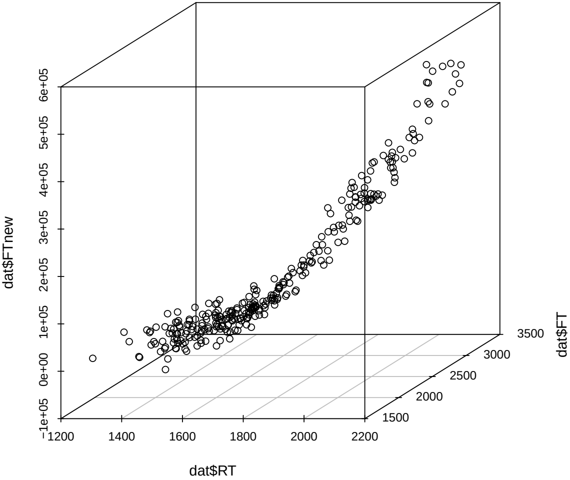
3D Scatter Plots

- Very similar to the usual Scatter Plot, just with a "second x-axis"
- Option 1: The `scatterplot3d` package
- Load the package

```
install.packages("scatterplot3d")
load(scatterplot3d)
```

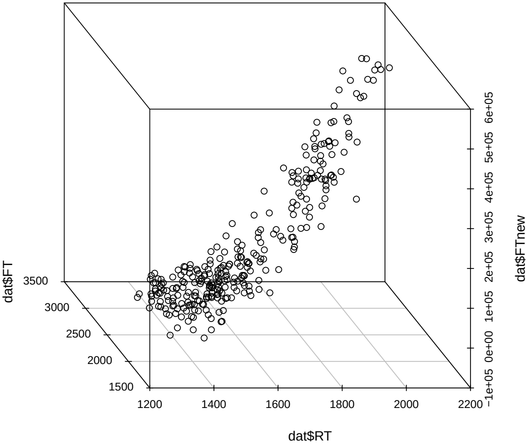
3D Scatter Plots

```
scatterplot3d(x=dat$RT,y=dat$FT,z=dat$FTnew)
```



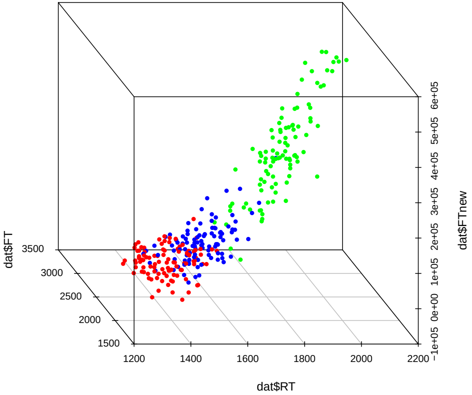
3D Scatter Plots: Customize

```
scatterplot3d(x=dat$RT,y=dat$FT,z=dat$FTnew,
angle=120)
```



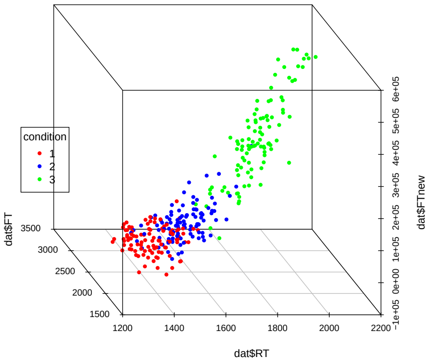
3D Scatter Plots: Customize

```
colors <- c("red","blue","green")
colors <- colors[as.numeric(dat$condition)]
scatterplot3d(x=dat$RT,y=dat$FT,z=dat$FTnew,
angle=120,color=colors,pch=20)
```



3D Scatter Plots: Customize

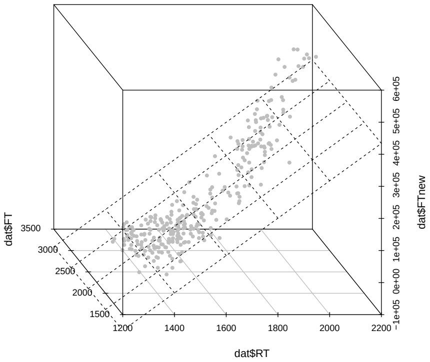
```
colors <- c("red","blue","green")
colors <- colors[as.numeric(dat$condition)]
scatterplot3d(x=dat$RT,y=dat$FT,z=dat$FTnew,angle=120,
color=colors,pch=20)
legend("left", legend = levels(dat$condition),
title="condition", col=c("red","blue","green"), pch=20)
```



3D Scatter Plots: Regression Plane

- (Does not work with interaction terms)

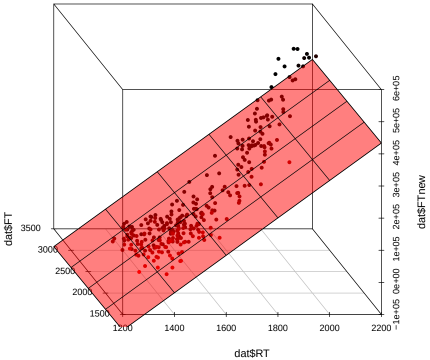
```
s3d <- scatterplot3d(x=dat$RT,y=dat$FT,z=dat$FTnew,
angle=120,pch=20,color="grey")
reg3d <- lm(FTnew ~ RT + FT,data=dat)
s3d$plane3d(reg3d)
```



3D Scatter Plots: Customize Regression Plane

- (Does not work with interaction terms)

```
s3d <- scatterplot3d(x=dat$RT,y=dat$FT,z=dat$FTnew,
angle=120,pch=20,color="grey")
reg3d <- lm(FTnew ~ RT + FT,data=dat)
s3d$plane3d(reg3d,lty=1,
draw_polygon=T,polygon_args=list(col=rgb(1,0,0,0.5)))
```



3D Scatter Plots

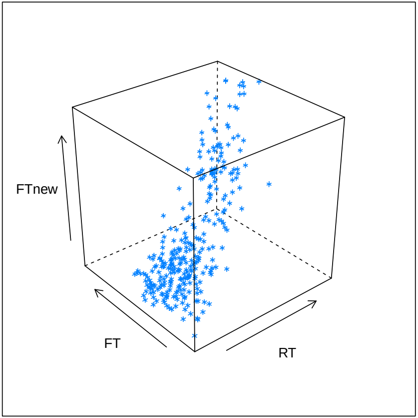
- For a tutorial on 3D Scatter plots, see http://www.sthda.com/english/wiki/scatterplot3d-3d-graphics-r-software-and-data-visualization

3D Scatter Plots

- Option 2: The `lattice` package
- Load the package (we have installed it before):
  `library(lattice)`

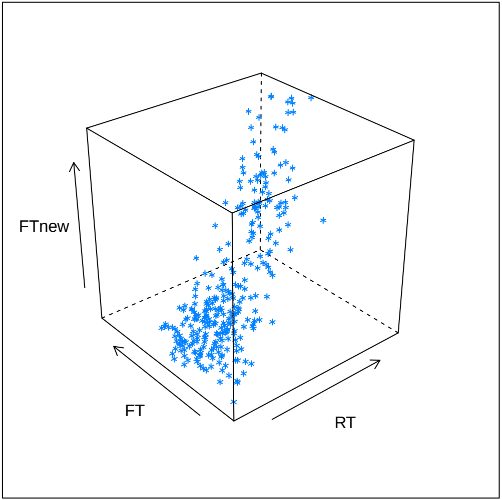
3D Scatter Plots

```
cloud(FTnew ~ RT + FT,data=dat)
```



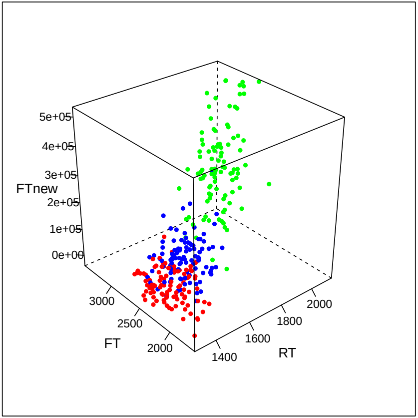
3D Scatter Plots: Customize

```
cloud(FTnew ~ RT + FT,data=dat)
```



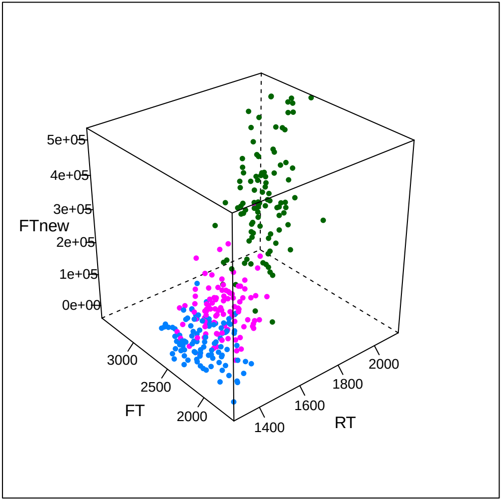
3D Scatter Plots: Customize

```
colors <- c("red","blue","green")
colors <- colors[as.numeric(dat$condition)]
cloud(FTnew ~ RT + FT,
data=dat,col=colors,scales=list(arrows=F),pch=20)
```



3D Scatter Plots: Customize

```
cloud(FTnew ~ RT + FT,
data=dat,group=condition,scales=list(arrows=F),pch=20)
```



3D Scatter Plots: Customize

- For a third option using the packages `plot3D` and `plot3Drgl`, see this tutorial:

  http://www.sthda.com/english/wiki/impressive-package-for-3d-and-4d-graph-r-software-and-data-visualization

Going fancy: The `rgl` package

- Install the `rgl` package:

```
install.packages("rgl")
library(rgl)
```

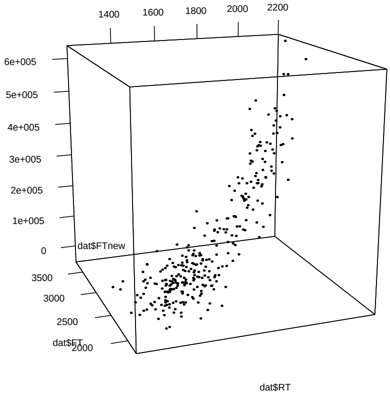
Going fancy: The rgl package

- Install the rgl package:

```
install.packages("rgl")
library(rgl)
```

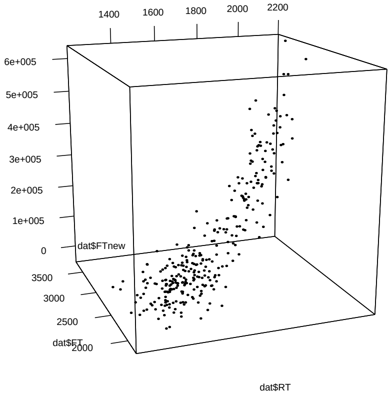
3D Scatter Plots: rgl

```
plot3d(x=dat$RT,y=dat$FT,z=dat$FTnew)
```



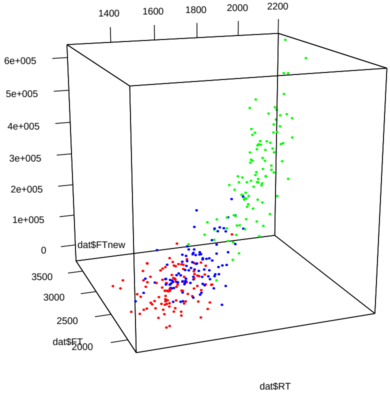
3D Scatter Plots: rgl

```
plot3d(x=dat$RT,y=dat$FT,z=dat$FTnew)
```



3D Scatter Plots: rgl

```
colors <- c("red","blue","green")
colors <- colors[as.numeric(dat$condition)]
plot3d(x=dat$RT,y=dat$FT,z=dat$FTnew,col=colors)
```



3D Scatter Plots: rgl animation

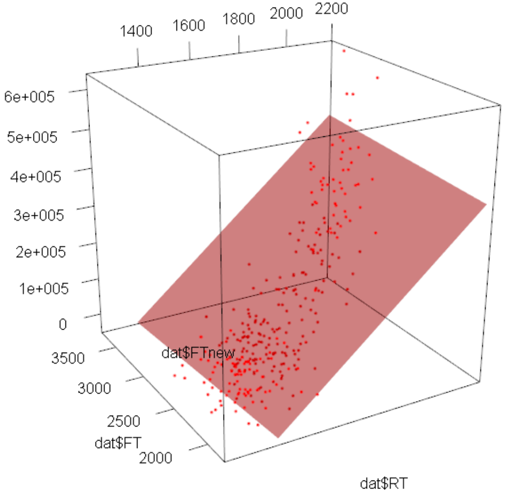
```
colors <- c("red","blue","green")
colors <- colors[as.numeric(dat$condition)]
plot3d(x=dat$RT,y=dat$FT,z=dat$FTnew,col=colors)
play3d(spin3d(), duration=12)
```

rgl plots: Regression Plane

```
plot3d(x=dat$RT,y=dat$FT,z=dat$FTnew,col="red")
reg3d <- lm(FTnew ~ RT + FT,data=dat)
coefs <- coef(reg3d)
a <- coefs["RT"]
b <- coefs["FT"]
c <- -1
d <- coefs["(Intercept)"]
planes3d(a, b, c, d, alpha=0.5,col="red")
```

- Use `c <- -1` for every data set

rgl plots: Regression Plane

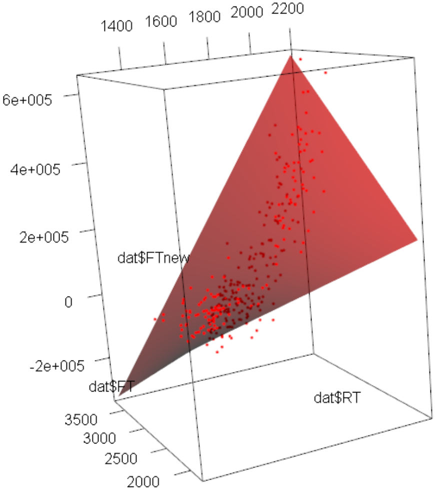


rgl plots: Regression "Plane" with interaction

```
plot3d(x=dat$RT,y=dat$FT,z=dat$FTnew,col="red")

reg3d2 <- lm(FTnew ~ RT*FT,data=dat)

grd <- expand.grid(RT=sort(unique(dat$RT)),
FT=sort(unique(dat$FT)))

grd$pred <- predict(reg3d2, newdata=grd)

persp3d(x=unique(grd$RT), y=unique(grd$FT),
z=matrix(grd$pred,length(unique(grd$RT)),length(unique(grd$FT))),
add=TRUE,col="red",alpha=.7)
```

- There is a function calles `persp` to create surface plots as "normal", static plots, but I find the `rgl` version simpler (can easily be added to a Scatter Plot)

rgl plots: Regression "Plane" with interaction



`rgl` plots: Regression "Plane" with interaction

- By combining previous approaches, we could also plot interaction planes by condition
- Use the commands `plot3d`, `points3d` and `persp3d` in combination with `dat[dat$condition == 1,]` and so on
- Due to the amount of coding involved, this will be omitted from this course

`rgl` plots: Surface Plots

- Difference between Scatter Plots and Surface Plots: Surface Plots need exactly *one* value of z for *each* value of x and y

`rgl` plots: Surface Plots

- Difference between Scatter Plots and Surface Plots: Surface Plots need exactly *one* value of z for *each* value of x and y
- They are plots of *functions*

`rgl` plots: Surface Plots

- Difference between Scatter Plots and Surface Plots: Surface Plots need exactly *one* value of z for *each* value of x and y
- They are plots of *functions*
- Other examples: Histograms, Regression Lines

rgl plots: Surface Plots

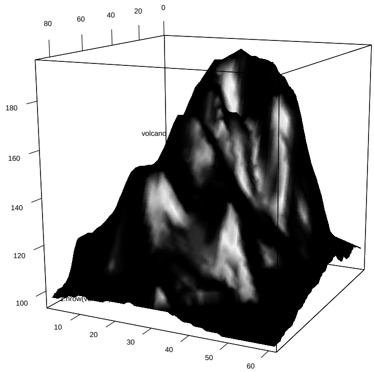
- Another example: Load the data set `volcano`:

```
data(volcano)
```

rgl plots: Surface Plots

- Plot this matrix as a surface plot:
  `persp3d(x=1:nrow(volcano),y=1:ncol(volcano), z=volcano)`
- `nrow(volcano)` gives the number of rows of this matrix, `ncol(volcano)` gives the number of columns
- `1:nrow(volcano)` gives all integer values from 1 to the number of rows

rgl plots: Surface Plots

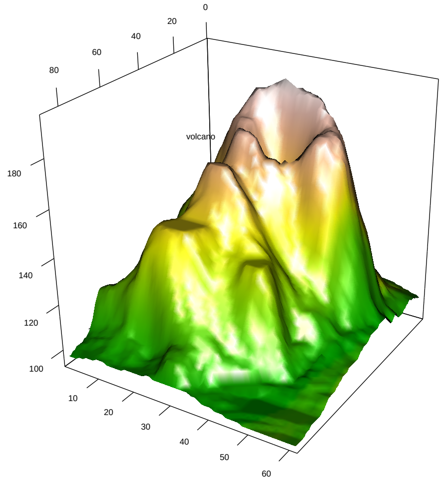


rgl plots: Surface Plots

- Use terrain colors:
  `zlim <- range(volcano) zlen <- zlim[2] - zlim[1] + 1 colors <- terrain.colors(zlen,alpha=0) col2 <- colors[volcano-min(volcano)]`
- We have now created a color palette! This is actually very similar to something like `cols = c("red","blue","green")`
- We will have a closer look at color palettes later

rgl plots: Surface Plots

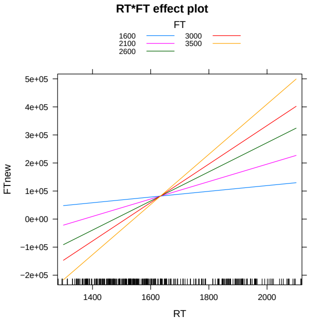
```
persp3d(x=1:nrow(volcano),y=1:ncol(volcano),
z=volcano,col=col2)
```

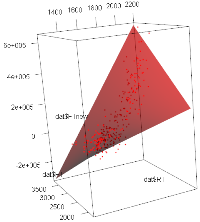


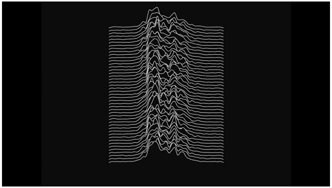
`rgl` plots: Surface Plots

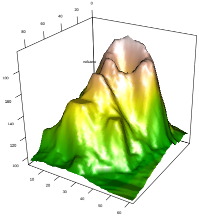
- So why exactly have we started plotting volcanoes now?

Surface Plots and Continuous Interactions

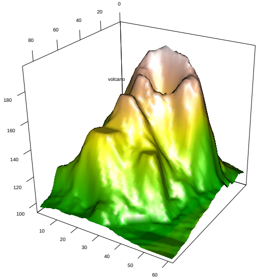


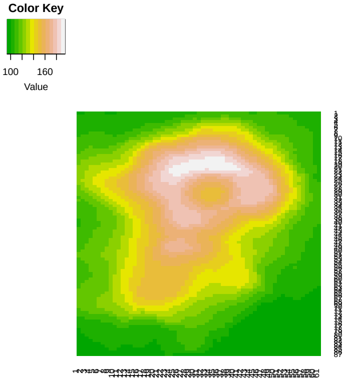




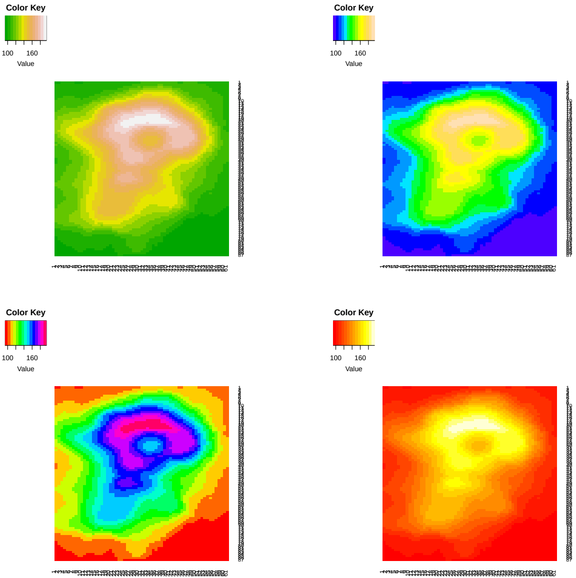


Surface Plots and Heat Maps



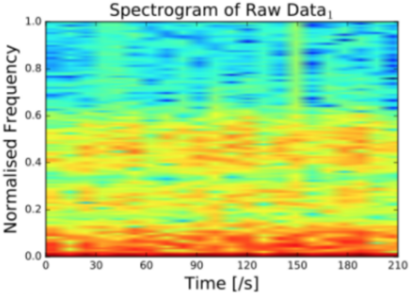


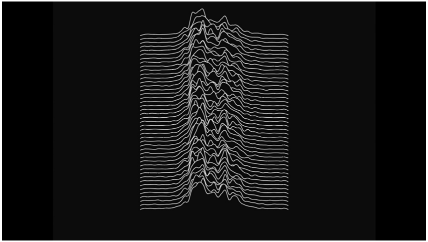
Heat Maps


Color Key
100 160
Value
Color Key
100 160
Value
Color Key
100 160
Value
Color Key
100 160
Value

Heat Maps: Common applications

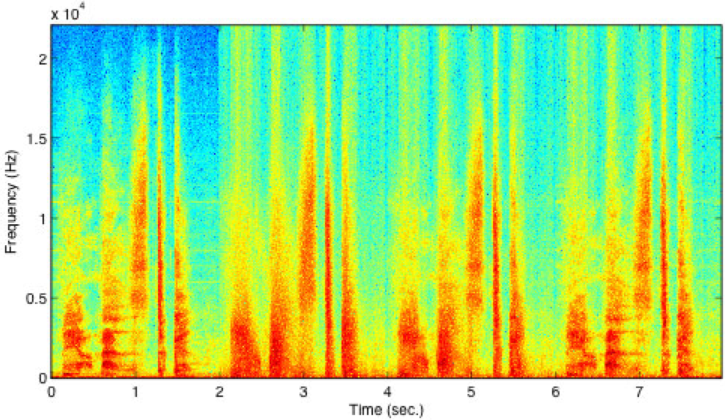
- EEG frequency bands





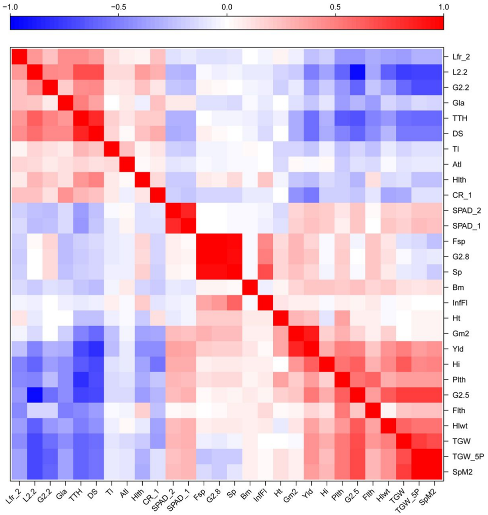
Heat Maps: Common applications

- Speech frequency bands



Heat Maps: Common applications

- Correlation Matrices between many different variables



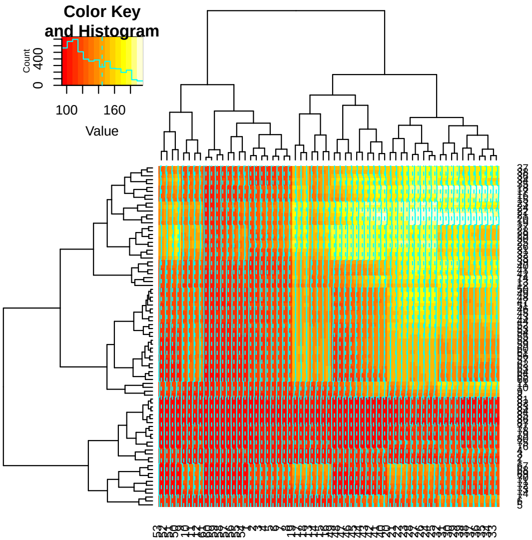
Heat Maps

- Standard R contains a `heatmap()` function
- But `heatmap.2()`, included in the `gplots` package, comes with more options
- Load the package

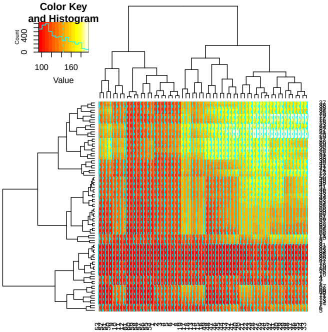
```
install.packages("gplots")
library(gplots)
```

Heat Maps

```
heatmap.2(volcano)
```



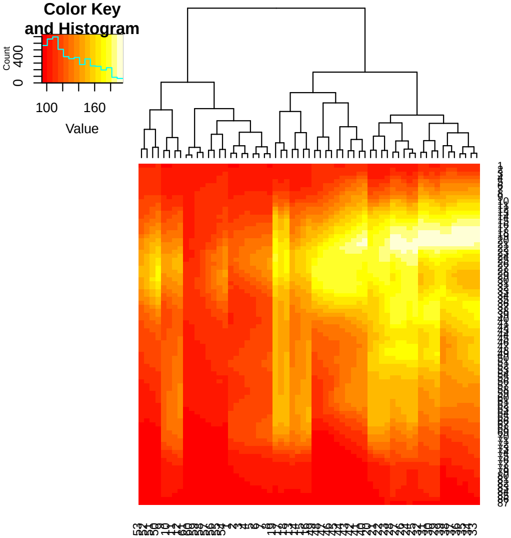
Heat Maps



- This is not really what we want:
  - Factor-separating lines in the plot
  - Rows and columns are clustered (as indicated by dendrograms at the side) and re-ordered
  - (This re-ordering is useful to plot correlation clusters)

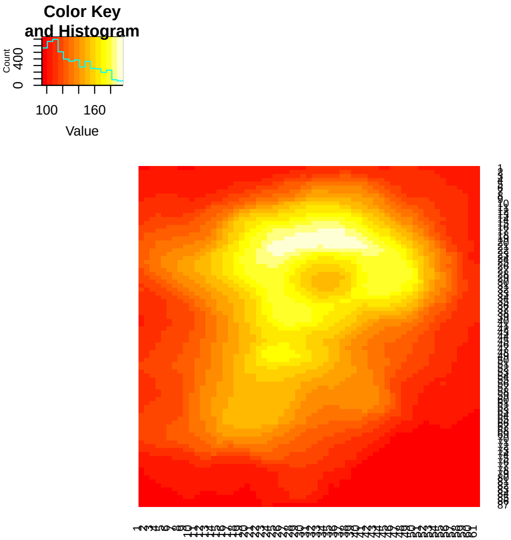
Heat Maps

```
heatmap.2(volcano,trace="none",Rowv=F)
```



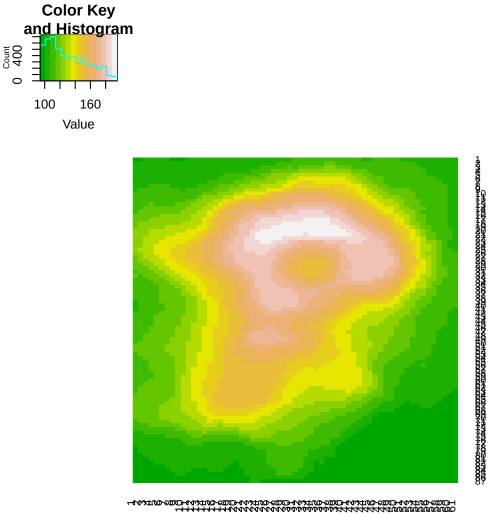
Heat Maps

```
heatmap.2(volcano,trace="none",Rowv=F,Colv=F)
```



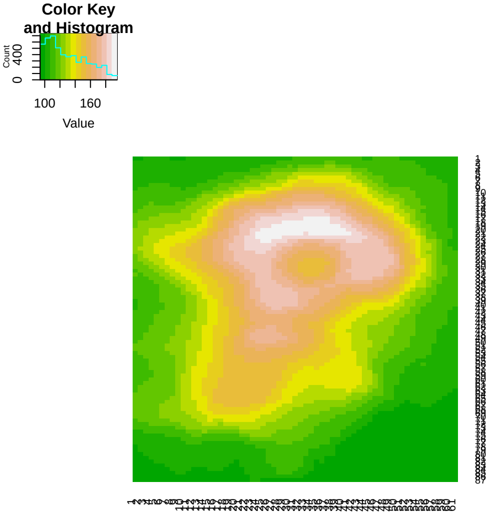
Heat Maps: Change Color

```
heatmap.2(volcano,trace="none",Rowv=F,Colv=F,
col="terrain.colors")
```



Heat Maps: Erase everything but the plot

```
heatmap.2(volcano,trace="none",Rowv=F,Colv=F,
col="terrain.colors")
```



Heat Maps: Erase everything but the plot

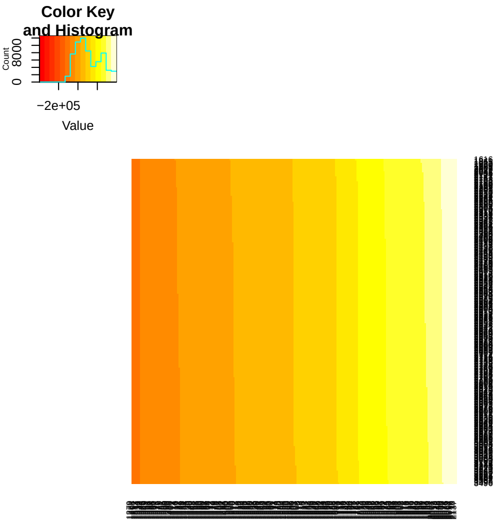
- The `heatmap.2` function comes with many, many options specifying
  - The heatmap itself
  - The dendograms
  - The axes
  - The legend
  - The general plot
- To get an overview, call the help function:
  `?heatmap.2`

Heat Maps: Multiple Regression

- Back to our original data set:

```
heatreg1 <- lm(FTnew ~ RT + FT,data=dat)
grd <- expand.grid(RT=sort(unique(dat$RT)),
FT=sort(unique(dat$FT)))
grd$pred <- predict(heatreg1, newdata=grd)
grd2 <- xtabs(pred ~ FT + RT,grd)
heatmap.2(grd2,Rowv=F,Colv=F,trace="none")
```

Heat Maps: Multiple Regression

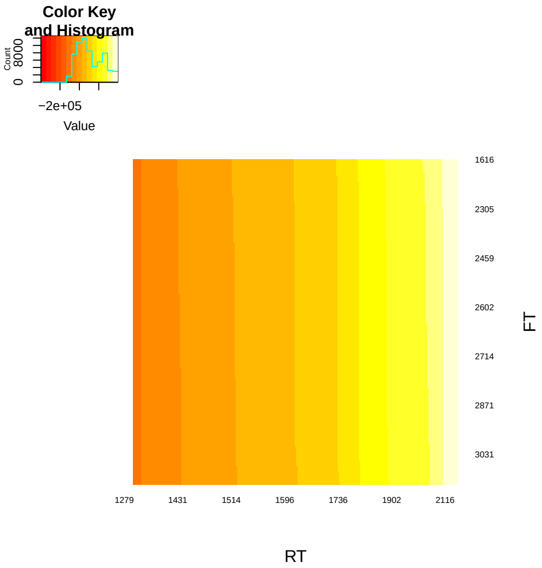


Heat Maps: Multiple Regression

- Workaround for sensible non-factorial axes

```
heatreg1 <- lm(FTnew ~ RT + FT,data=dat)
grd <- expand.grid(RT=sort(unique(dat$RT)),
FT=sort(unique(dat$FT)))
grd$pred <- round(predict(heatreg1, newdata=grd),0)
grd2 <- xtabs(pred ~ FT + RT,grd)
heatrows <- rep("",300)
heatrows[seq(1,300,40)] <- rownames(grd2)[seq(1,300,40)]
heatcols <- rep("",300)
heatcols[seq(1,300,40)] <- colnames(grd2)[seq(1,300,40)]
heatmap.2(grd2,Rowv=F,Colv=F,trace="none",
labRow=heatrows,labCol=heatcols,srtCol=0,
xlab="RT",ylab="FT")
```

Heat Maps: Multiple Regression

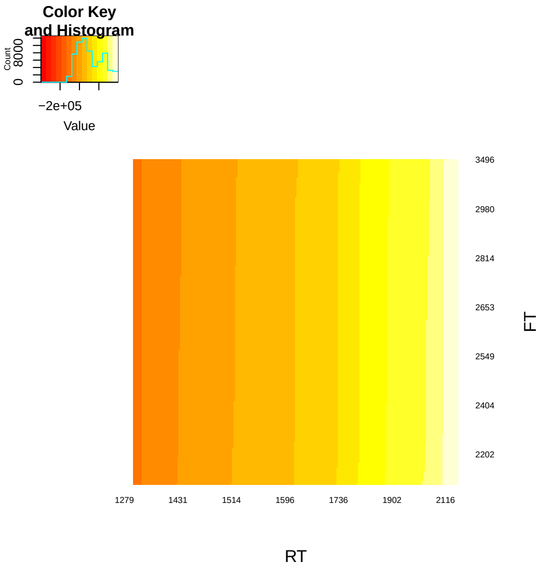


Heat Maps: Multiple Regression

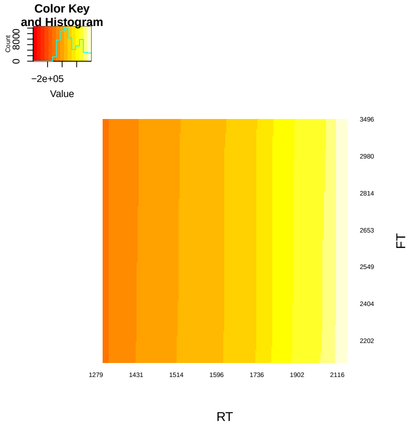
- Make the y-axis increasing instead of decreasing

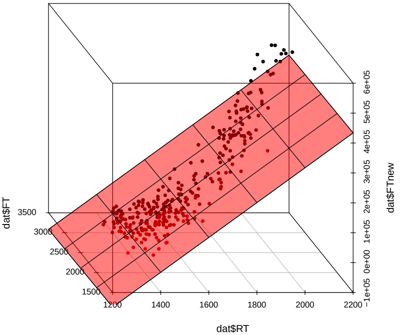
```
heatreg1 <- lm(FTnew ~ RT + FT,data=dat)
grd <- expand.grid(RT=sort(unique(dat$RT)),
FT=sort(unique(dat$FT)))
grd$pred <- round(predict(heatreg1, newdata=grd),0)
grd2 <- xtabs(pred ~ FT + RT,grd)
grd2 <- grd2[order(rownames(grd2),decreasing=T),]
heatrows <- rep("",300)
heatrows[seq(1,300,40)] <- rownames(grd2)[seq(1,300,40)]
heatcols <- rep("",300)
heatcols[seq(1,300,40)] <- colnames(grd2)[seq(1,300,40)]
heatmap.2(grd2,Rowv=F,Colv=F,trace="none",
labRow=heatrows,labCol=heatcols,srtCol=0,
xlab="RT",ylab="FT")
```

Heat Maps: Multiple Regression



Heat Maps: Multiple Regression



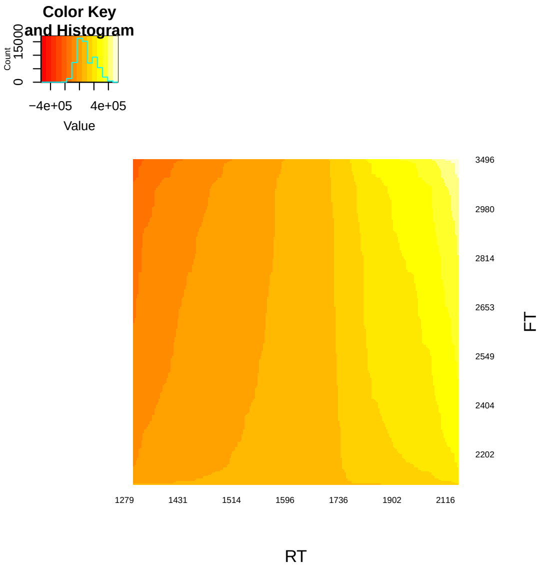


Heat Maps: Continuous interactions

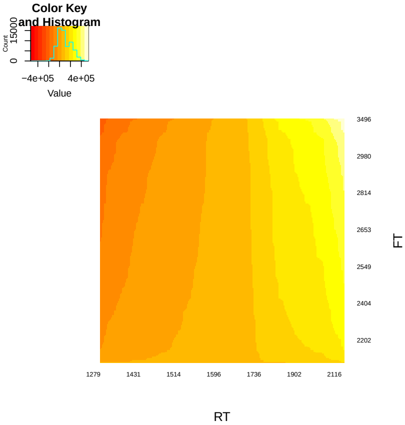
- Just change the regression model: + to *

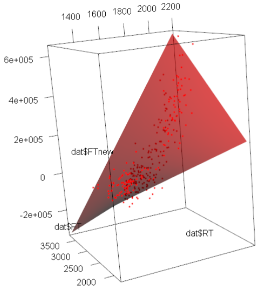
```
heatreg1 <- lm(FTnew ~ RT * FT,data=dat)
grd <- expand.grid(RT=sort(unique(dat$RT)),
FT=sort(unique(dat$FT)))
grd$pred <- round(predict(heatreg1, newdata=grd),0)
grd2 <- xtabs(pred ~ FT + RT,grd)
grd2 <- grd2[order(rownames(grd2),decreasing=T),]
heatrows <- rep("",300)
heatrows[seq(1,300,40)] <- rownames(grd2)[seq(1,300,40)]
heatcols <- rep("",300)
heatcols[seq(1,300,40)] <- colnames(grd2)[seq(1,300,40)]
heatmap.2(grd2,Rowv=F,Colv=F,trace="none",
labRow=heatrows,labCol=heatcols,srtCol=0,
xlab="RT",ylab="FT")
```

Heat Maps: Continuous interactions



Heat Maps: Continuous interactions



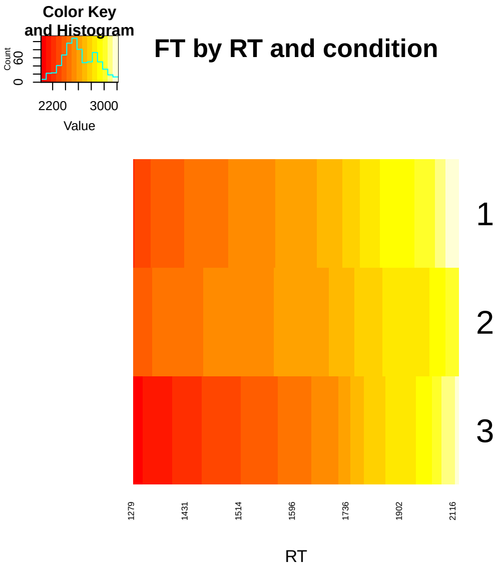


Heat Maps: Multiple Regression

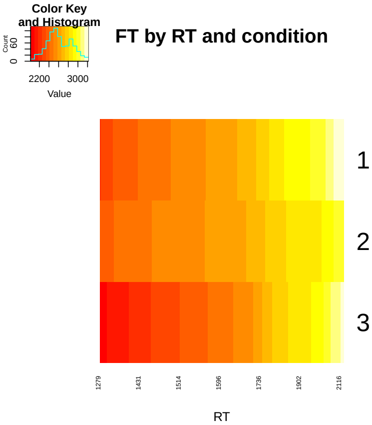
- All these heat maps also work if one or both predictor variables are factors

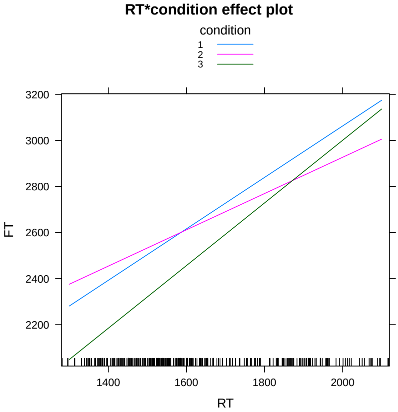
```
heatreg2 <- lm(FT ~ RT*condition,data=dat)
grd <- expand.grid(RT=sort(unique(dat$RT)),
condition=levels(dat$condition))
grd$pred <- round(predict(heatreg2, newdata=grd),0)
grd2 <- xtabs(pred ~ condition + RT,grd)
heatrows <- rep("",300)
heatcols[seq(1,300,40)] <- colnames(grd2)[seq(1,300,40)]
heatmap.2(grd2,Rowv=F,Colv=F,trace="none",
labCol = heatcols,xlab="RT",main="FT by RT and condition")
```

Heat Maps: Multiple Regression



Heat Maps: Multiple Regression





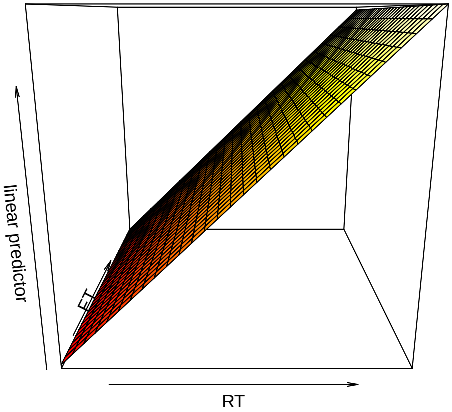
Heat Maps and Surface Plots: The `mgcv` package

- The `mgcv` package makes these plots way easier
- Load the package:

```
library(mgcv)
```

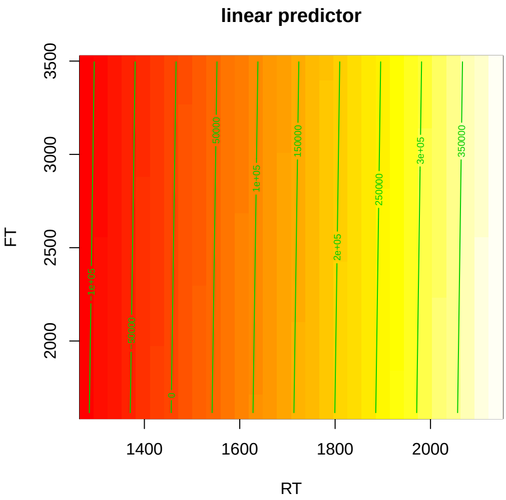
Heat Maps and Surface Plots: The `mgcv` package

```
nlint1 <- gam(FTnew ~ RT + FT ,data=dat)
vis.gam(nlint1)
```



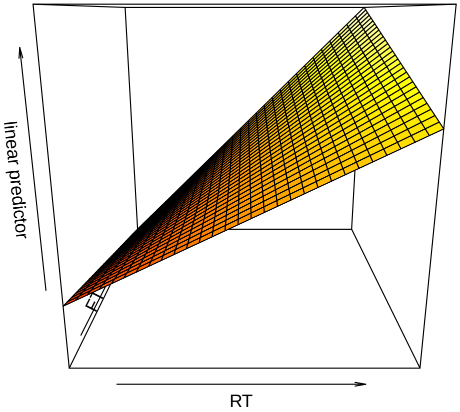
Heat Maps and Surface Plots: The mgcv package

```
nlint1 <- gam(FTnew ~ RT + FT ,data=dat)
vis.gam(nlint1,plot.type="contour")
```



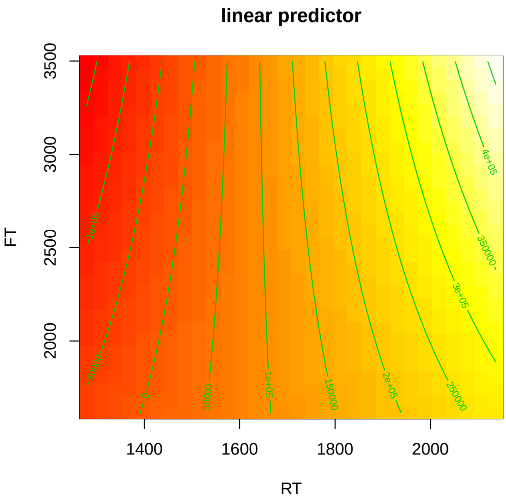
Heat Maps and Surface Plots: The `mgcv` package

```
nlint2 <- gam(FTnew ~ RT * FT ,data=dat)
vis.gam(nlint2)
```



Heat Maps and Surface Plots: The mgcv package

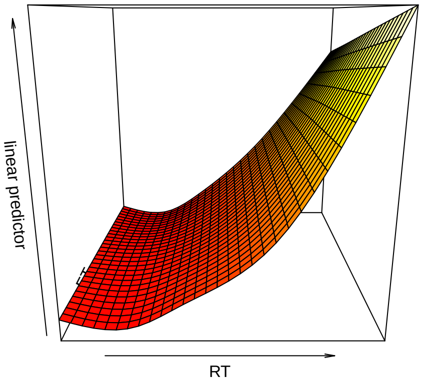
```
nlint2 <- gam(FTnew ~ RT * FT ,data=dat)
vis.gam(nlint2,plot.type="contour")
```



Heat Maps and Surface Plots: The mgcv package

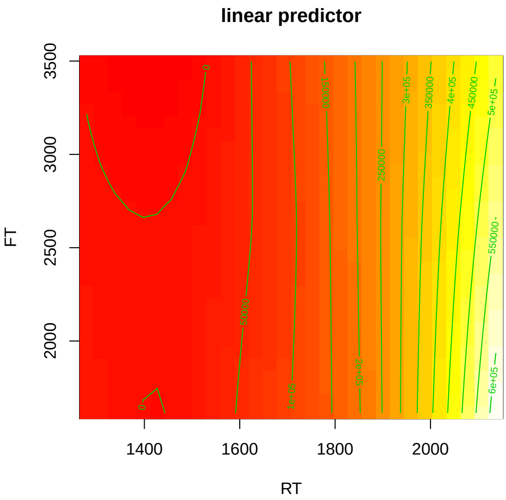
- With the mgcv package, we can even plot non-linear interactions:

```
nlint3 <- gam(FTnew ~ te(RT,FT) ,data=dat)
vis.gam(nlint3)
```



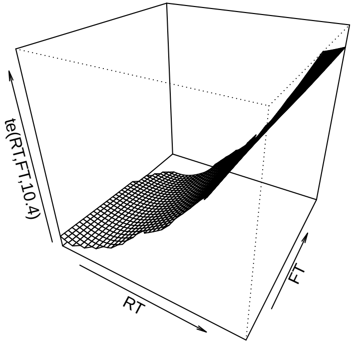
Heat Maps and Surface Plots: The mgcv package

```
nlint3 <- gam(FTnew ~ te(RT,FT) ,data=dat)
vis.gam(nlint3,plot.type="contour")
```



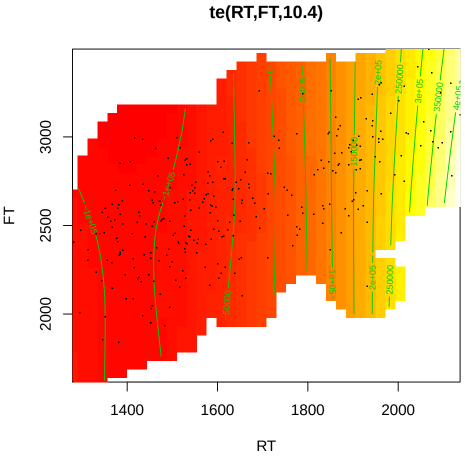
Heat Maps and Surface Plots: The mgcv package

- Alternative option: plot() on the gam() object `nlint3 <- gam(FTnew ~ te(RT,FT) ,data=dat)`
  `plot(nlint3,scheme=1)`



Heat Maps and Surface Plots: The mgcv package

```
nlint3 <- gam(FTnew ~ te(RT,FT) ,data=dat)
plot(nlint3,scheme=2)
```



- Call the help function ?plot.gam() for more options

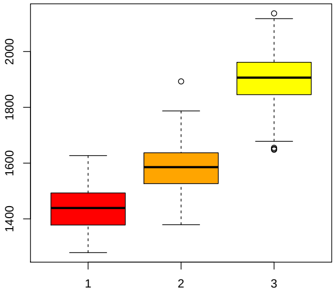
Multiple Regression: More complex models

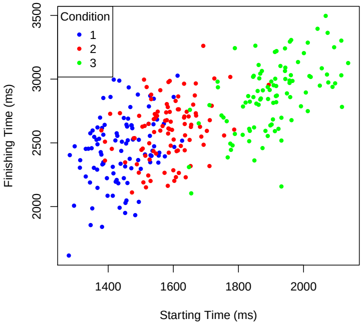
- Sometimes, regression models include a higher number of terms, such as
  `RT ~ pred1*pred2 + pred2*pred3 + pred4 + pred5`
- All of the plotting functions presented can handle these cases and "pick out" the effects of interest, for example by
  - Specifying `term` in the `effect()` function
  - Specifying `view` in the `vis.gam()` function
  - Specifying `select` in the `plot()` function for `gam()` objects
  - ...

Plotting Data vs. Analyses

Plotting Data vs. Analyses

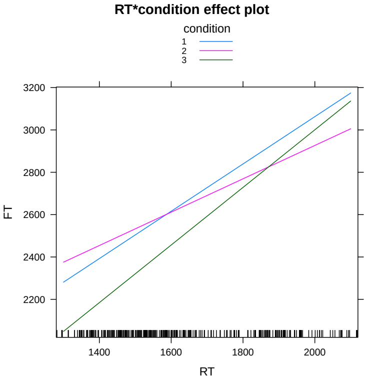
- In some plots, we are plotting descriptive summaries of the data

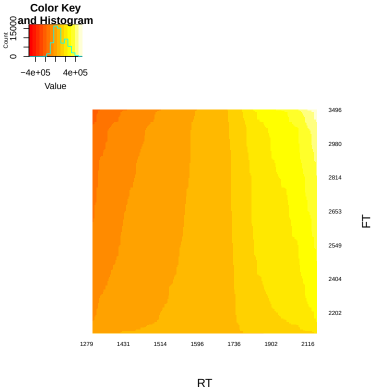




Plotting Data vs. Analyses

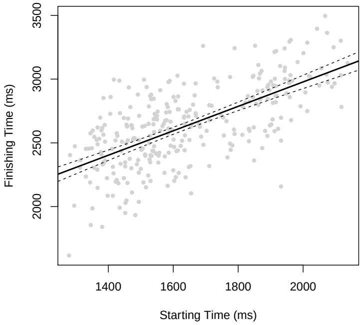
- In some plots, we are plotting the results of analyses

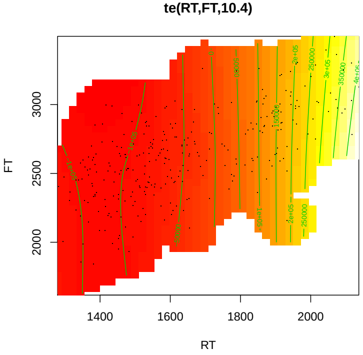




Plotting Data vs. Analyses

- In some plots, we are plotting both



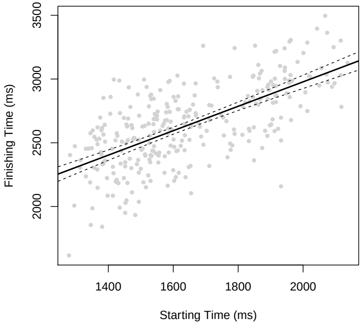


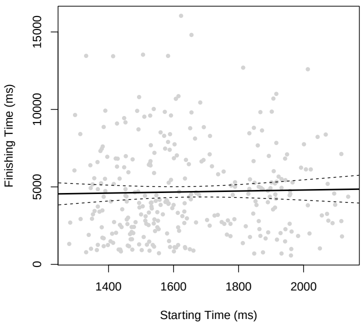
Plotting Data vs. Analyses

- Every plot should serve a purpose, so you have to choose between these options in every case

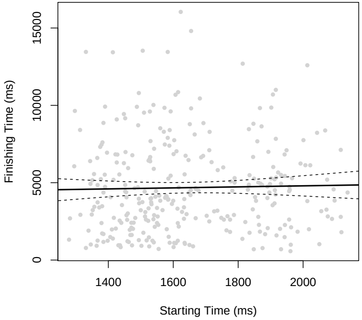
Plotting Data vs. Analyses

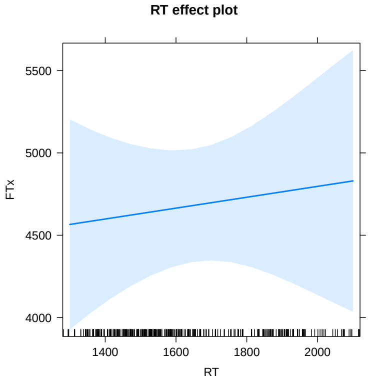
- Every plot should serve a purpose, so you have to choose between these options in every case
- Although plotting both data and analyses seems the overall best way, the data sometimes makes this difficult:





Plotting Data vs. Analyses





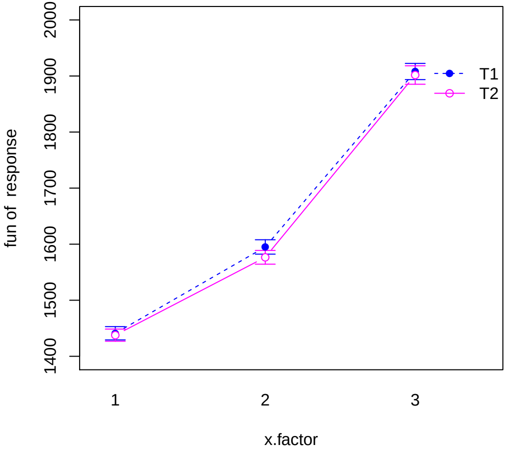
- Always tell your reader/audience what they are seeing
- This is what figure captions are for

Error Bars

- A prime example for this issue are standard factorial designs with a continuous dependent variable (for example 2x2 design for RTs)
- Let's look at different ways to plot this

Error Bars

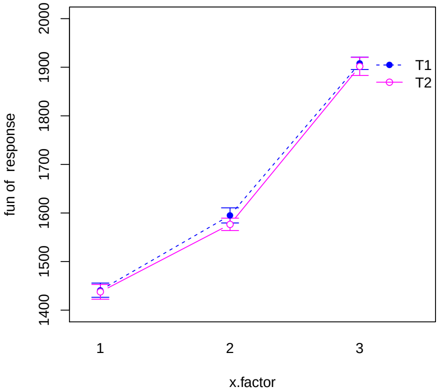
- Line Plot of means:
  `bargraph.CI(x.factor=dat$condition,group=dat$time, response=dat$RT,ylim=c(1400,2000),col=c("blue","magenta"))`



Error Bars

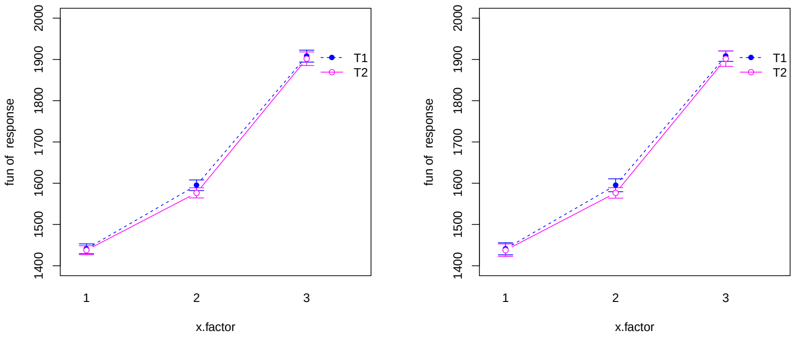
- Line Plot of means (aggregated data):

```
agg <- aggregate(RT ~ condition + time + participant,
data = dat,mean)
lineplot.CI(x.factor=agg$condition,group=agg$time,
response=agg$RT,ylim=c(1400,2000),col=c("blue","magenta"))
```



Error Bars

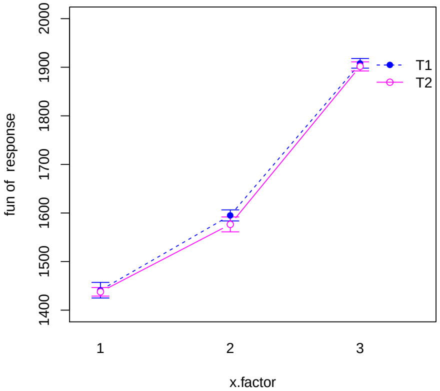
- Line Plot of means (raw vs. aggregated data):



Error Bars

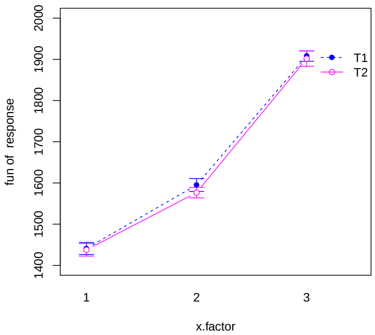
- Line Plot of means (data aggregated over items):

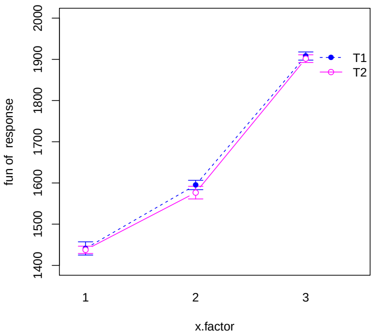
```
agg2 <- aggregate(RT ~ condition + time + item,
data = dat,mean)
lineplot.CI(x.factor=agg2$condition,group=agg2$time,
response=agg2$RT,ylim=c(1400,2000),col=c("blue","magenta"))
```



Error Bars

- Line Plot of means (aggregated over participants vs. items):

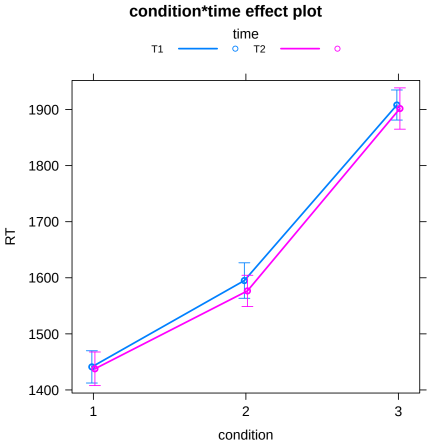




Error Bars

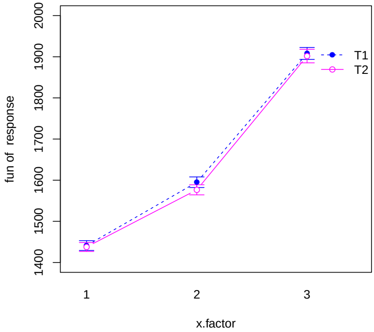
- Line Plot of model predictions:

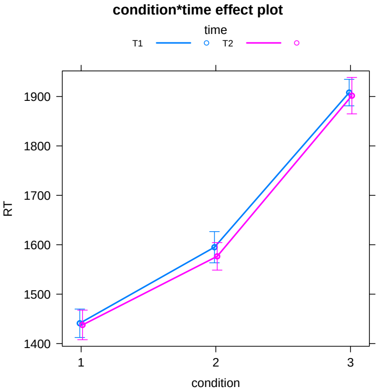
```
model1 <- lmer(RT ~ condition*time +
(condition*time|participant), data=dat)
plot(effect("condition*time",model1),lines=list(multiline=TRUE)
confint=list(style="bars"))
```



Error Bars

- Line Plot of means (raw data vs. model predictions):





Error Bars

- In this case, the plots are all very similar, but they display different things!
- Be clear about that

Error Bars

- For repeated-measures designs (one participant/item in more than one condition), adjustments to the error bars have been suggested:
- See for example

  Loftus, G. R., & Masson, M. E. (1994). Using confidence intervals in within-subject designs. *Psychonomic Bulletin & Review, 1,* 476-490.

  Cousineau, D. (2005). Confidence intervals in within-subject designs: A simpler solution to Loftus and Masson's method. *Tutorials in Quantitative Methods for Psychology, 1,* 42-45.

Stepwise Plotting

Stepwise Plotting

- Up to now, most plots were handled in a single command, and then adjusted in the options
- Let's check the alternative way: Starting from an empty plot and add everything piece by piece
- Takes longer, but gives most control

Stepwise Plotting: Line Plot with Error Bars

- Empty Plot

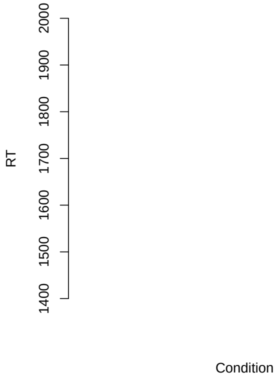
```
plot(0,type="n",axes=F,xlim=c(0.5,3.5),ylim=c(1400,2000),
xlab="Condition",ylab="RT")
```

- `xlim` and `ylim` options are very important here: They define the window to be plotted

Stepwise Plotting: Line Plot with Error Bars

- Add the y-axis:

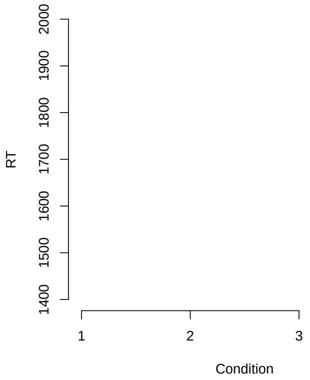
```
axis(2)
```



Stepwise Plotting: Line Plot with Error Bars

- Add the x-axis:

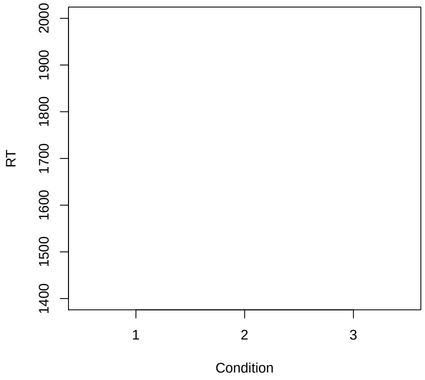
```
axis(1,at=1:3,labels=c("1","2","3"))
```



Stepwise Plotting: Line Plot with Error Bars

- Add a box:

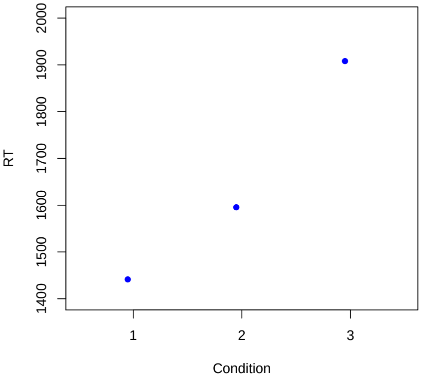
```
box()
```



Stepwise Plotting: Line Plot with Error Bars

- Add points for T1:

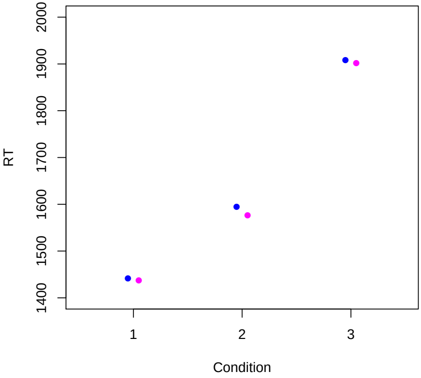
```
means <- aggregate(RT   condition + time,data=dat,mean)
e <- .05
points(x=(1:3 - e),y=means[means$time == "T1",]$RT,
pch=16,col="blue")
```



Stepwise Plotting: Line Plot with Error Bars

- Add points for T2:

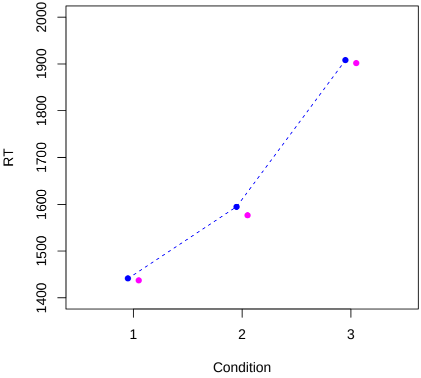
```
points(x=(1:3 + e),y=means[means$time == "T2",]$RT,
pch=16,col="magenta")
```



Stepwise Plotting: Line Plot with Error Bars

- Add lines for T1:

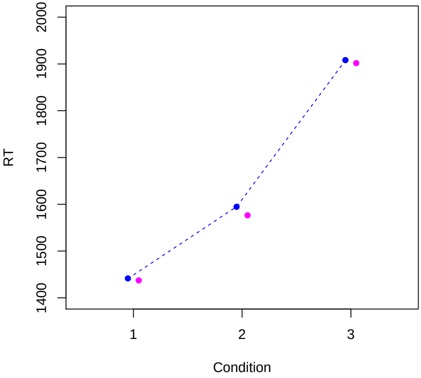
```
lines(x=(1:3 - e),y=means[means$time == "T1",]$RT,
lty=2,col="blue")
```



Stepwise Plotting: Line Plot with Error Bars

- Add lines for T1:

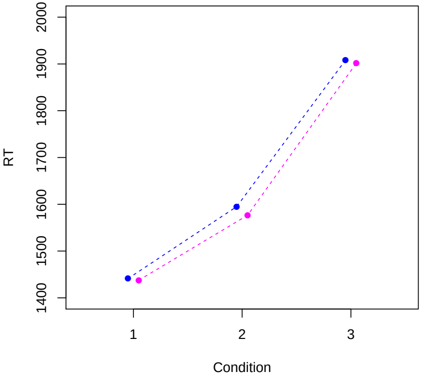
```
lines(x=(1:3 - e),y=means[means$time == "T1",]$RT,
lty=2,col="blue")
```



Stepwise Plotting: Line Plot with Error Bars

- Add lines for T2:

```
lines(x=(1:3 + e),y=means[means$time == "T2",]$RT,
lty=2,col="magenta")
```



Stepwise Plotting: Line Plot with Error Bars

- Compute standard errors (function `se` is part of the `sciplot` package) and attach them to the object containing the means:

```
means$se <- aggregate(RT ~ condition + time,data=dat,se)$RT
```

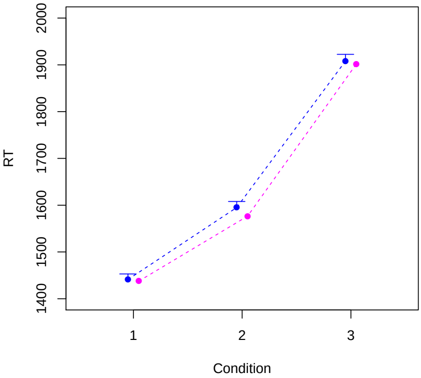
- Compute $M + SE$ and $M - SE$:

```
means$seplus <- means$RT + means$se
means$seminus <- means$RT - means$se
```

Stepwise Plotting: Line Plot with Error Bars

- Draw error bars as arrows:

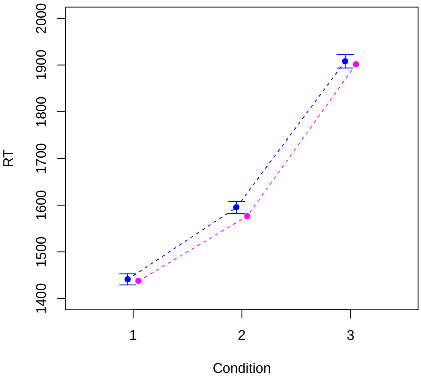
```
arrows(x0=(1:3-e),x1=(1:3-e),
y0=means[means$time == "T1",]$RT,
y1=means[means$time == "T1",]$seplus,
col="blue",angle=90,length=.1)
```



Stepwise Plotting: Line Plot with Error Bars

- Draw error bars as arrows:

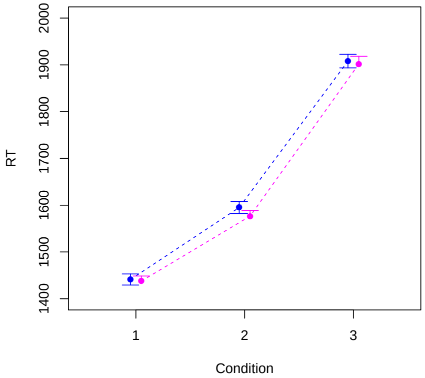
```
arrows(x0=(1:3-e),x1=(1:3-e),
y0=means[means$time == "T1",]$RT,
y1=means[means$time == "T1",]$seminus,
col="blue",angle=90,length=.1)
```



Stepwise Plotting: Line Plot with Error Bars

- Draw error bars as arrows:

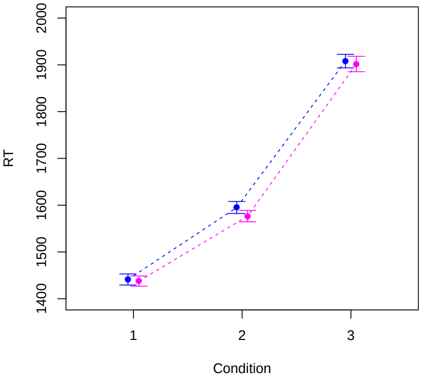
```
arrows(x0=(1:3+e),x1=(1:3+e),
y0=means[means$time == "T2",]$RT,
y1=means[means$time == "T2",]$seplus,
col="magenta",angle=90,length=.1)
```



Stepwise Plotting: Line Plot with Error Bars

- Draw error bars as arrows:

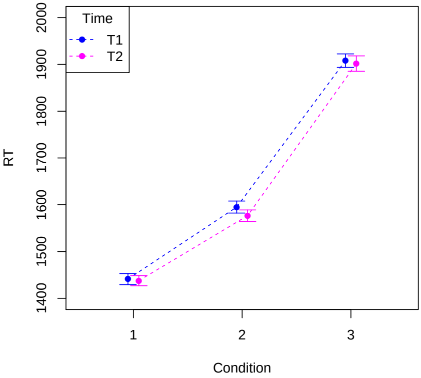
```
arrows(x0=(1:3+e),x1=(1:3+e),
y0=means[means$time == "T2",]$RT,
y1=means[means$time == "T2",]$seminus,
col="magenta",angle=90,length=.1)
```



Stepwise Plotting: Line Plot with Error Bars

- Add a legend:

```
legend("topleft",pch=16,lty=2,
col=c("blue","magenta"),legend = c("T1","T2"),title = "Time")
```



Stepwise Plotting: Other graphical elements

`segments()`	Line segments between pairs of points
`abline()`	A line with slope and intercept
`rect()`	Rectangles (can be used for Bar Plots)
`polygon()`	Polygons

Controlling Graphical Parameters

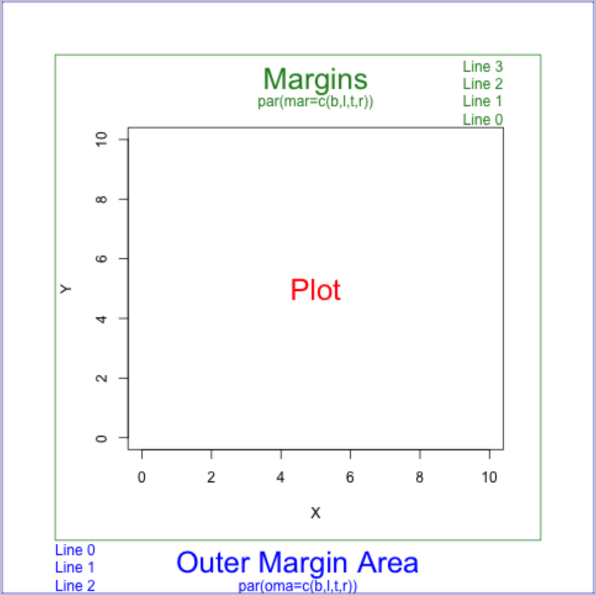
Controlling Graphical Parameters

- Graphical parameters are adjusted globally, using the `par()` function
- They will affect every subsequent plot
- To reset `par()` to "factory settings", use the function `dev.off()` (without argument), which will close the plotting device

Controlling Graphical Parameters

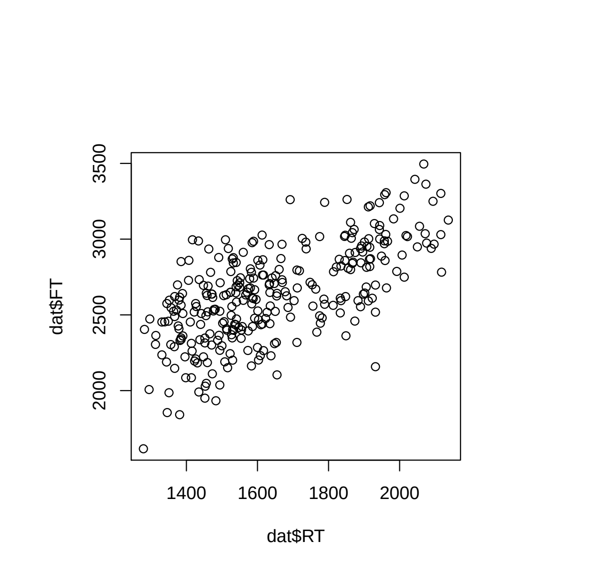
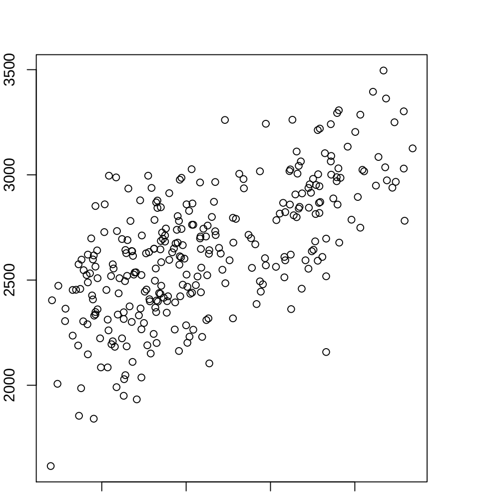
- There are many, many graphical parameters that can be changed
- See `?par`
- We will only deal with the most common ones here

Controlling Graphical Parameters: Margins



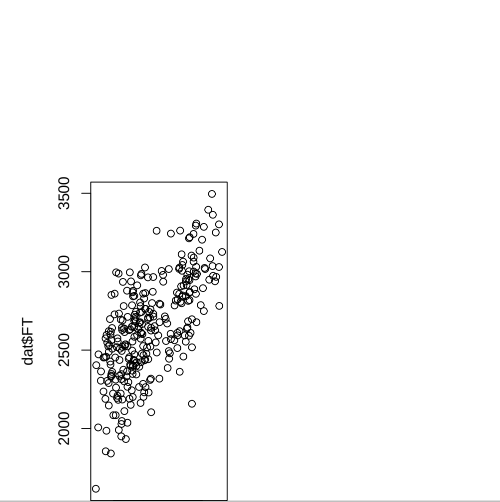
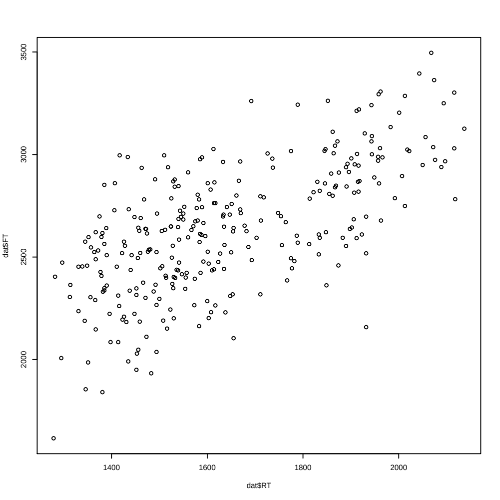
Controlling Graphical Parameters: Margins

- ```
  par(oma=c(1,2,3,4))
  plot(dat$RT,dat$FT)
  ```


  ```

Controlling Graphical Parameters: Margins

- ```
  par(mar=c(1,2,3,4))
  plot(dat$RT,dat$FT)
  ```


  ```

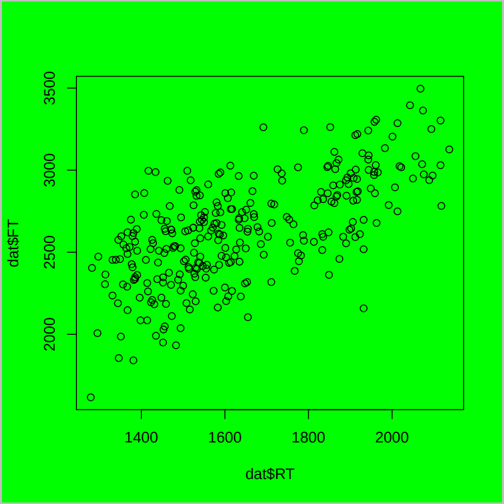
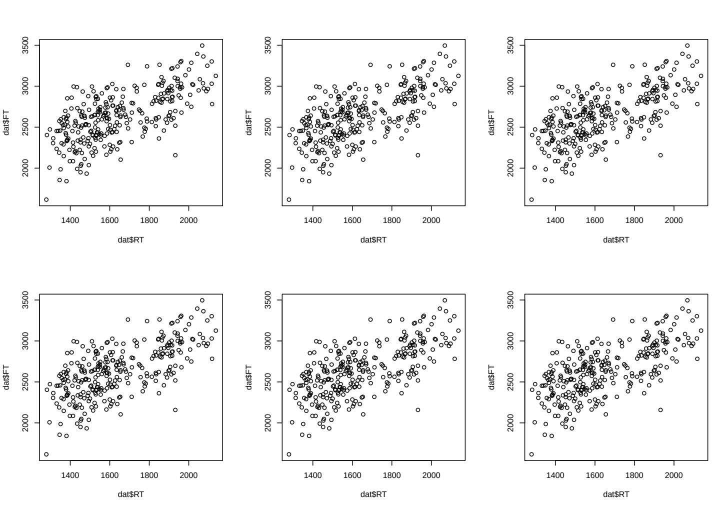
Controlling Graphical Parameters: Margins

- ```
  par(mai=c(0,1,2,3))
  plot(dat$RT,dat$FT)
  ```


  ```

Controlling Graphical Parameters: Character Size

- ```
  par(cex=.5)
  plot(dat$RT,dat$FT)
  ```


  ```

Controlling Graphical Parameters: Background Color

- ```
  par(bg="green")
  plot(dat$RT,dat$FT)
  ```


  ```

Controlling Graphical Parameters: Multiple Graphs

- ```
  par(mfrow=c(2,3))
  for(i in 1:6)plot(dat$RT,dat$FT)
  ```


  ```

Multiple Graphs: More fine-tuning

- Define the Frame:

```
zones=matrix(c(2,0,1,3), ncol=2, byrow=TRUE)
layout(zones, widths=c(.75,.25), heights=c(.25,.75))
par(oma=c(1,1,1,1))
par(mar=c(1,1,1,1))
```

- Inspect `zones`

```
zones
      [,1]  [,2]
 [1,]   2     0
 [2,]   1     3
```

Multiple Graphs: More fine-tuning

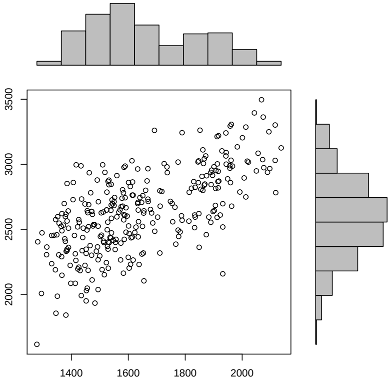
- Prepare two histograms:

```
xhist <- hist(dat$RT,plot=FALSE)
yhist <- hist(dat$FT,plot=FALSE)
top <- max(c(xhist$counts, yhist$counts))
```

Multiple Graphs: More fine-tuning

- Plot all three graphs:

```
plot(dat$RT,dat$FT)
barplot(xhist$counts, axes=FALSE, ylim=c(0, top), space=0)
barplot(yhist$counts, axes=FALSE, xlim=c(0, top), space=0,
horiz=TRUE)
```



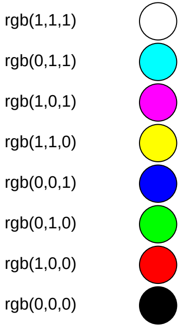
Colors

Colors

- As we have seen throughout the course, there are a lot of standard colors that can be accessed by name
- For an overview, see http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf

Colors

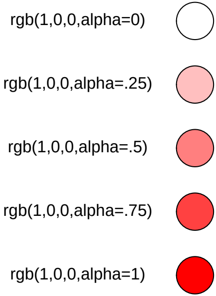
- Additional colors can be customized using the `rgb()` function



- use `rgb(...,maxValue=255)` for the standard 255 scale

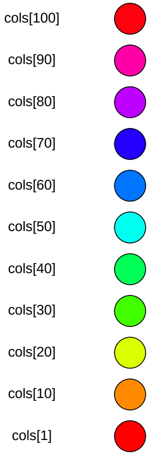
Colors

- Use the `alpha` option to adjust parameters



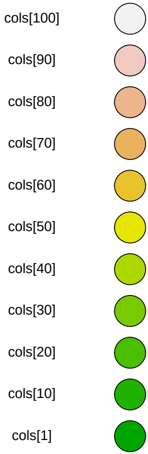
Colors

- Use a pre-defined color palette:
  `cols <- rainbow(100)`



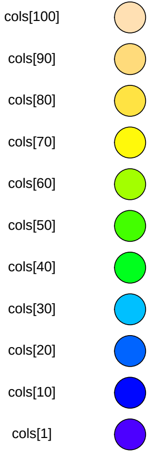
Colors

- Use a pre-defined color palette:
  `cols <- terrain.colors(100)`



Colors

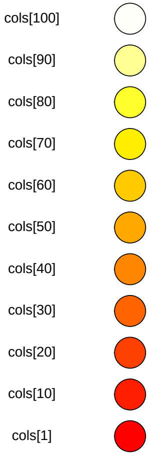
- Use a pre-defined color palette:
  `cols <- topo.colors(100)`



Colors

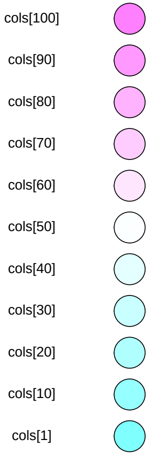
- Use a pre-defined color palette:

```
cols <- heat.colors(100)
```



Colors

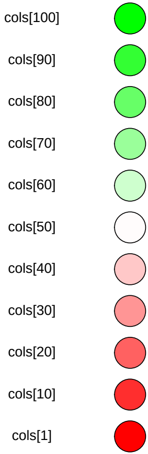
- Use a pre-defined color palette:
  ```
  cols <- cm.colors(100)
  ```



Colors

- Create your own color palette:

```
cols <- colorRampPalette(c("red","white","green"))(100)
```



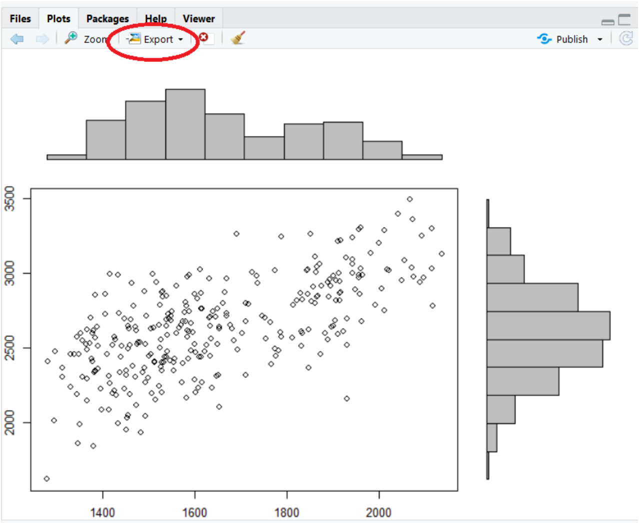
Colors

- For more information (also on the `RColorBrewer` package), see https://www.stat.ubc.ca/~jenny/STAT545A/block14_colors.html

Exporting Plots

Exporting Plots

- In RStudio, plots can be exported by clicking on "Export"



Exporting Plots

- Plots can also be exported using R commands:

```
pdf("C:/User/Documents/myplot.pdf")
plot(dat$RT,dat$FT)
dev.off()
```

- Everything between opening the device with `pdf()` and closing it with `dev.off()` is exported

Exporting Plots

- Adjusting the size of the plot:

```
pdf("C:/User/Documents/myplot.pdf",width=5,height=5)
plot(dat$RT,dat$FT)
dev.off()
```

- The size of characters and symbols will depend on the figure size (smaller symbols with larger sizes)

Exporting Plots

- There are many other options that can be specified while exporting: font style, point size, background and foreground color, ...
- And also other file formats:

  Raster images

  - `png("myplot.png")`
  - `jpeg("myplot.jpeg")`
  - `bmp("myplot.bmp")`

  Vector Graphics

  - `pdf("myplot.pdf")`
  - `postscript("myplot.ps")`
  - `win.metafile("myplot.wmf")`

Exporting `rgl` graphs

- Rotatable 3D-Plots created with the `rgl` package are exported as follows:
  - Create the rgl graph
    ```
    data(volcano)
    persp3d(x=1:nrow(volcano),y=1:ncol(volcano),z=volcano)
    ```
  - Turn them to the position you want to export (can also be done using commands, see `?view3d`)
  - Call `rgl.snapshot(filename="snapshot.png")` or `rgl.postscript(filename="rgl2.pdf",fmt="pdf")` (also supports ps, eps, tex, svg, pgf)

Exporting `rgl` graphs

- You can also export animations as .gifs, using commands such as `movie3d(spin3d(),movie="mygif-",duration=12,dir=getwd())`
- This requires the package `magick` to be installed
- To also export all the individual .png files used to create the .gif, use `movie3d(spin3d(),movie="mygif-",duration=12, dir=getwd(),clean=F)`