Monte Carlo Simulation

Paul Johnson, Ben Kite, Terrence Jorgensen[1]

[1] Center for Research Methods and Data Analysis

2018

KU

Outline

KU

Outline

This is a template, not instructions

- All of this is brought to us by R (R Core Team, 2017)

Outline

What is a Monte Carlo Simulation?

- Generating from a known probability model
- Comparing variations among separate samples drawn from the model
- "Monte Carlo Analysis in Academic Research" (Johnson, 2013) gives history and applications doi:10.1093/oxfordhb/9780199934874.013.0022

Goals of a Monte Carlo Simulation

- Consider a statistical procedure (e.g., a t test) that receives data and returns a result – i.e., parameter estimates, sample statistics
- Presumably there is a "true" set of parameters ("population values") that the estimate is supposed to represent
- We wonder
  - Does the procedure yield unbiased (correct "on average") estimates of the "true" parameters?
  - Is an estimator consistent (closer to correct as the sample size grows?)
  - Is the sampling distribution of the estimates normal, symmetric, etc.?
- From estimates with "real data", we can't say if we are "right", because we don't know the true model of the data generator

The Standard "Playbook"

- Specify a data generating process (i.e., a set of parameters)
  - often called a "population" in statistical vernacular
- Draw random samples from it
- Apply the procedure to each sample
  - Save estimates, tests, p-values, etc.
- Evaluate the procedure
  - Compare stats to parameters, check distributions

Goals of Analysis

- Check that a procedure behaves as expected
  - Does the null rejection rate match the nominal Type I error rate?
  - Are estimates unbiased?
- See how a procedure behaves when assumptions are violated
  - Inflated Type I error rates? Robust if minor?
  - Effects of missing data?
  - Effect of sample size?
- Compare 2 procedures – OLS v. WLS; LGCM v. MLM
- Power analysis

Replication is a Priority

- We must be able to regenerate results exactly without saving each data set
- Pseudorandom number generator (PRNG)
  - Algorithm that generates seemingly random streams of integers
  - The "random" numbers you get depend on a random "state" characterized by a "seed"
    - Initial starting condition can be controlled by specifying an single integer, which is commonly referred to as the "seed" (but, technically, it is not)
    - Setting the initial seed makes it possible to replicate draws from random number generators

Outline

Let's Generate Random Numbers in R!

Let's open our R syntax and get started. Here is an outline of today's topics/tasks:

- Generate some simple (pseudo) random numbers
- Generate random samples of data using population parameters
- Design a small-scale Monte Carlo study
  - How are Type I errors affected by between-group differences in N and SD?

R terminology

For most distributions, R offers functions with names like **r**norm, **d**norm, **p**norm and **q**norm

- **r** returns a pseudorandom sample from that distribution
- **d** returns the probability density (or probability mass for discrete distributions)
- **p** returns the cumulative probability distribution (CDF)
- **q** returns the quantile associated with a certain cumulative probability

Help pages for some built-in distributions

```
?rnorm
```

```
Normal                          package:stats                          R Documentation

The Normal Distribution

Description:

     Density, distribution function, quantile function and random
     generation for the normal distribution with mean equal to 'mean'
     and standard deviation equal to 'sd'.

Usage:

     dnorm(x, mean = 0, sd = 1, log = FALSE)
     pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
     qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
     rnorm(n, mean = 0, sd = 1)

Arguments:

    x, q: vector of quantiles.

       p: vector of probabilities.
```

Help pages for some built-in distributions ...

```
      n: number of observations. If 'length(n) > 1', the length is
         taken to be the number required.

   mean: vector of means.

     sd: vector of standard deviations.

log, log.p: logical; if TRUE, probabilities p are given as log(p).

lower.tail: logical; if TRUE (default), probabilities are P[X <= x]
         otherwise, P[X > x].

Details:

     If 'mean' or 'sd' are not specified they assume the default values
     of '0' and '1', respectively.

     The normal distribution has density

             f(x) = 1/(sqrt(2 pi) sigma) e^-((x - mu)^2/(2 sigma^2))

     where mu is the mean of the distribution and sigma the standard
     deviation.

Value:
```

Help pages for some built-in distributions ...

```
     'dnorm' gives the density, 'pnorm' gives the distribution
     function, 'qnorm' gives the quantile function, and 'rnorm'
     generates random deviates.

     The length of the result is determined by 'n' for 'rnorm', and is
     the maximum of the lengths of the numerical arguments for the
     other functions.

     The numerical arguments other than 'n' are recycled to the length
     of the result.  Only the first elements of the logical arguments
     are used.

     For 'sd = 0' this gives the limit as 'sd' decreases to 0, a point
     mass at 'mu'.  'sd < 0' is an error and returns 'NaN'.

Source:

     For 'pnorm', based on

     Cody, W. D. (1993) Algorithm 715: SPECFUN - A portable FORTRAN
     package of special function routines and test drivers.  _ACM
     Transactions on Mathematical Software_ *19*, 22-32.

     For 'qnorm', the code is a C translation of
```

Help pages for some built-in distributions ...

```
     Wichura, M. J. (1988) Algorithm AS 241: The percentage points of
     the normal distribution.  _Applied Statistics_, *37*, 477–484.

     which provides precise results up to about 16 digits.

     For 'rnorm', see RNG for how to select the algorithm and for
     references to the supplied methods.

References:

     Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) _The New S
     Language_.  Wadsworth & Brooks/Cole.

     Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) _Continuous
     Univariate Distributions_, volume 1, chapter 13.  Wiley, New York.
See Also:

     Distributions for other standard distributions, including 'dlnorm'
     for the _Log_normal distribution.

Examples:

     require(graphics)

     dnorm(0) == 1/sqrt(2*pi)
     dnorm(1) == exp(-1/2)/sqrt(2*pi)
```

Help pages for some built-in distributions ...

```
dnorm(1) == 1/sqrt(2*pi*exp(1))

## Using "log = TRUE" for an extended range :
par(mfrow = c(2,1))
plot(function(x) dnorm(x, log = TRUE), -60, 50,
     main = "log { Normal density }")
curve(log(dnorm(x)), add = TRUE, col = "red", lwd = 2)
mtext("dnorm(x, log=TRUE)", adj = 0)
mtext("log(dnorm(x))", col = "red", adj = 1)

plot(function(x) pnorm(x, log.p = TRUE), -50, 10,
     main = "log { Normal Cumulative }")
curve(log(pnorm(x)), add = TRUE, col = "red", lwd = 2)
mtext("pnorm(x, log=TRUE)", adj = 0)
mtext("log(pnorm(x))", col = "red", adj = 1)

## if you want the so-called 'error function'
erf <- function(x) 2 * pnorm(x * sqrt(2)) - 1
## (see Abramowitz and Stegun 29.2.29)
## and the so-called 'complementary error function'
erfc <- function(x) 2 * pnorm(x * sqrt(2), lower = FALSE)
## and the inverses
erfinv <- function (x) qnorm((1 + x)/2)/sqrt(2)
erfcinv <- function (x) qnorm(x/2, lower = FALSE)/sqrt(2)
```

Each Subgroup Has an Assignment

- Choose one of these distributions (T, χ^2, Poisson, Uniform, Gamma, Beta, Cauchy, Logistic, Weibull, Binomial, or Negative Binomial)
- Review the help page for that (see below)
- Run example() for your distribution (may be helpful, maybe not)
- Run 1 simple set of commands to set the arguments and use the **r** variant. Create a histogram. Here's example demonstrating my use of the normal distribution

```
m <- 7
s <- 3
N <- 2000
y <- rnorm(N, m = m, s = s)
hist(y, breaks = 50, prob = TRUE)
range(y)
```

```
[1] -3.004026 16.992200
```

Each Subgroup Has an Assignment ...

```
str(y)
```

```
num [1:2000] 8.76 9.13 6.67 5.64 8.82 ...
```

- Report to rest of us on following
  - is output variable discrete or floating-point numeric?
  - what parameters control the data generator?
  - can you guess what the range of the variable might be (does it have values from $-\infty$ to ∞, or is it bounded, say, in $(0, \infty]$.

1. T distribution

```
?rt
```

2. χ^2 distribution

```
?rchisq
```

Each Subgroup Has an Assignment ...

3. Poisson distribution

```
?rpois
```

4. Uniform distribution

```
?runif
```

5. Gamma distribution

```
?rgamma
```

6. Beta distribution

```
?rbeta
```

7. Cauchy distribution

```
?rcauchy
```

Each Subgroup Has an Assignment ...

8. Logistic distribution

```
?rlogis
```

9. Weibull distribution

```
?rweibull
```

10. binomial distribution

```
?rbinom
```

11. Negative binomial distribution

```
?rnbinom
```

Binomial distribution

- The number of "Yes" answers in a sequence of "Yes" or "No" trials with fixed probability of "Yes"
- Represents coin flips "Heads" or "Tails"
- Conduct 10 flips with a fair coin, count number of Heads.
  Do that over and over, a total of $N = 100$ times

```
size <- 10
prob <- 0.5
N <- 100
y <- rbinom(n = 100, size = size, prob = prob)
y
```

```
 [1] 4 6 7 4 4 6 7 5 7 7 5 7 8 3 3 6 3 4 3 5 6 4 4 3 5 5 4 5 4 4
     7 3 5 6 1 7 6 3 6 4 5 2 7 4 5 4 8
[48] 2 4 5 6 6 4 5 4 4 5 2 3 3 5 7 3 7 5 5 6 5 6 6 3 5 6 5 4 3 5
    6 6 6 6 3 2 8 5 4 4 4 6 6 3 6 4 7
[95] 6 5 3 6 4 5
```

Binomial distribution ...

- What is the distribution of outcomes?

```
table(y)
```

```
y
 1  2  3  4  5  6  7  8
 1  4 15 22 22 22 11  3
```

I forgot to set the random generator's initial state

```
set.seed(234234)
y1 <- rbinom(n = 100, size = size, prob = prob)
head(y1)
```

```
[1] 5 9 4 6 5 5
```

```
set.seed(234234)
y2 <- rbinom(n = 100, size = size, prob = prob)
head(y2)
```

```
[1] 5 9 4 6 5 5
```

Bernoulli Trials

- We have N random samples and each one uses a collection of $size$ random draws.
- A Bernoulli trial is a sample of N observations in which the size is restricted to 1.
- Bernoulli is the base distribution of logit/probit models, each observation is a draw from a TRUE/FALSE outcome.

Two ways to think about Bernoulli Trials

- I'll do 1000 samples, each of size 1, with prob = 0.4.

```
N <- 1000
size <- 1
prob <- 0.4
y1 <- rbinom(N, size, prob)
head(y1)
```

```
[1] 0 0 0 0 1 0
```

```
## The total number of 1's is
sum(y1)
```

```
[1] 383
```

- I'll draw 1 sample, with size 1000, with prob = 0.4

Two ways to think about Bernoulli Trials ...

```
N <- 1
size <- 1000
y2 <- rbinom(N, size, prob)
y2
```

```
[1] 404
```

- What's the difference?

One Application: Modeling Rare Events

Celiac disease affects 1% of the population. We will draw one sample with size = 10

```
rbinom(1, size = 10, prob = .01)
```

```
[1] 0
```

How many would we expect in a random sample of 100 people?

```
rbinom(1, size = 100, prob = .01)
```

```
[1] 2
```

How many are found in a random sample of 1000 people?

```
rbinom(1, size = 1000, prob = .01)
```

```
[1] 10
```

- As size gets larger, the sample size drawn should get closer and closer to $prob \times size$.

Check that with a little simulation study

- I'll create 4 sets of draws with 4 values of the size parameter

```
N <- 2000
size1 <- 10
size2 <- 100
size3 <- 1000
size4 <- 10000
ohone <- 0.01 ## a joke!
y1 <- rbinom(N, size1, prob = ohone)
y2 <- rbinom(N, size2, prob = ohone)
y3 <- rbinom(N, size3, prob = ohone)
y4 <- rbinom(N, size4, prob = ohone)
```

- Convert output to proportions

Check that with a little simulation study ...

```
## Convert to proportions
y1p <- y1/size1
head(y1p)
```

```
[1] 0.0 0.0 0.1 0.0 0.0 0.0
```

```
y2p <- y2/size2
head(y2p)
```

```
[1] 0.03 0.00 0.00 0.00 0.00 0.00
```

```
y3p <- y3/size3
head(y3p)
```

```
[1] 0.010 0.012 0.006 0.009 0.011 0.011
```

Check that with a little simulation study ...

```
y4p <- y4/size4
head(y4p)
```

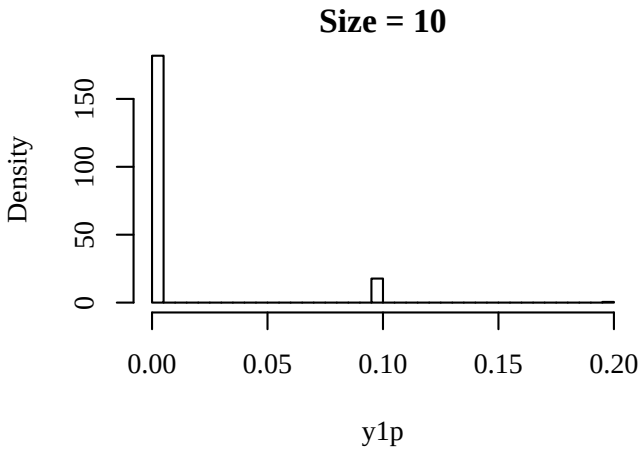
```
[1] 0.0084 0.0103 0.0111 0.0111 0.0099 0.0112
```

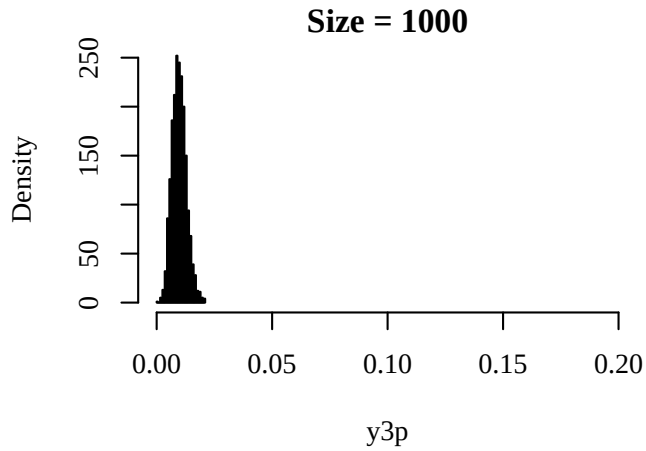
Check that with a little simulation study ...

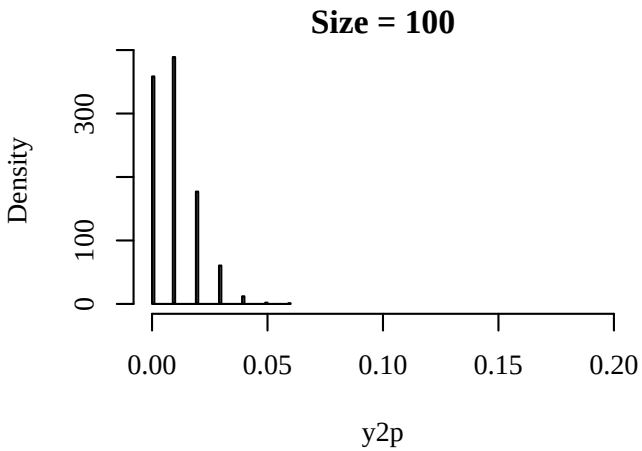
- Make a nice plot

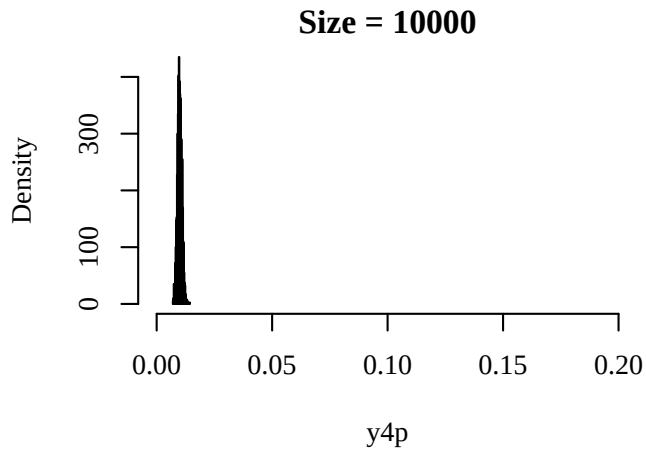
```
y1p.range <- range(y1p)
par(mfcol = c(2,2))
hist(y1p, prob = TRUE, xlim = y1p.range,
    breaks=50, main = paste("Size =", size1))
hist(y2p, prob = TRUE, xlim = y1p.range,
    breaks=50, main = paste("Size =", size2))
hist(y3p, prob = TRUE, xlim = y1p.range,
    breaks=50, main = paste("Size =", size3))
hist(y4p, prob = TRUE, xlim = y1p.range,
    breaks=50, main = paste("Size =", size4))
```

Check that with a little simulation study ...









If You Were Doing that For Real, I'd tighten it up

```
N <- 2000
size <- c(10, 100, 1000, 10000)
ohone <- 0.01 ## a joke!
for(j in seq_along(size)){
   y <- rbinom(N, size[j], prob = ohone)
     yname <- paste0("y", j)
     assign(yname, y)
 }
```

- The `assign()` puts the variables y1, y2, y3, y4 in the global workspace, which is rather careless
- I'd convert the output to a matrix in either
  - wide format: y with N rows and $\#\{\text{size}\}$ columns, or
  - long format: $N \times \#\{size\}$rows and 2 columns (1 column $size$ and 1 column "stacked y")
- Will discuss designing output

Normal distribution

- Most of us are familiar with at least a few special cases of the general linear model: regression, correlation, t tests, ANOVA.
- These assume a normally distributed outcome (at least, a normal residual term).
- These models assert the error term is normally distributed with a standard deviation of some number σ and expected value 0.

```
sigma <- 3
mu <- 0
error <- rnorm(100, m = mu, s = sigma)
```

```
head(error)
```

```
[1]  0.06315714  3.51111143 -7.30378820  0.13006657 -4.77096913
    -0.33498189
```

Normal distribution ...

```
mean(error)
```

```
[1] -0.03904841
```

- The normal distribution is most often written down as $N(\mu, \sigma^2)$, (in words $N(mu, sigma^2)$), we are thinking of the parameters as the expected value and variance
- In Bayesian software like BUGS and JAGS, they say the second parameter is $1/\sigma^2$. They call that precision, writing $N(\mu, \tau)$ for $\tau = 1/\sigma^2$
- In Bayesian software Stan, they differ again, referring to the normal by expected value and standard deviation, $N(\mu, \sigma)$

Normal distribution defaults

- The R rnorm default parameters are m=0 and s=1

```
y <- rnorm(10)
head(y)
```

```
[1] -0.1881444  0.4115536  1.6776777  2.9389290 -0.1080186 -0.3335427
```

These are commonly called Z scores. Referred to as a "standard normal" distribution.

- In many programs, one can only draw from N(0,1) and then manually rescale with the desired mean and standard deviation.
- If e_i is a draw from N(0,1), we can manufacture $N(m, s^2)$

$$y_i = mu + sigma \cdot e_i$$

Normal distribution defaults ...

- In R, it is usually not necessary to do that re-scaling manually because we can specify the expected value and standard deviation parameters.
- Example, IQ scores

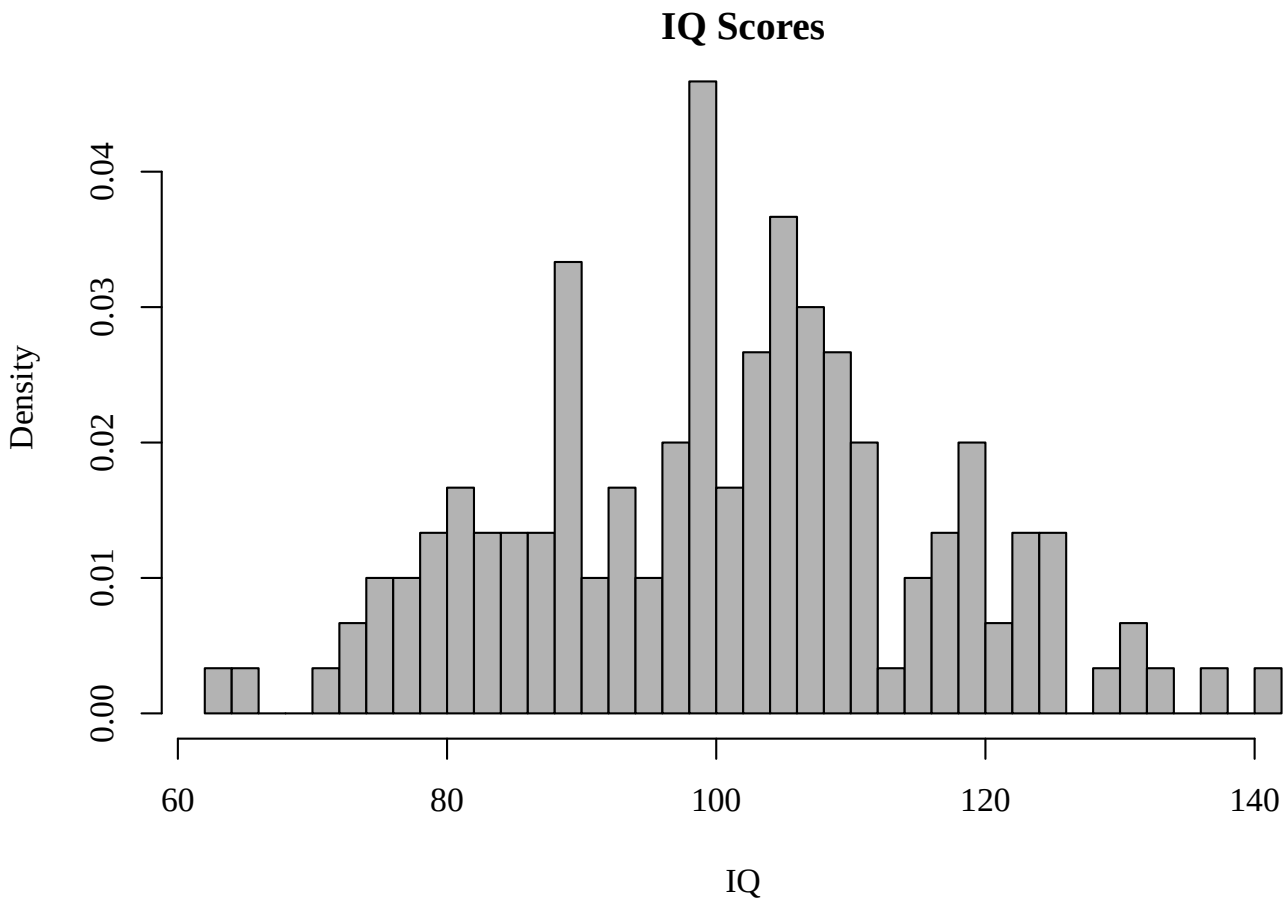
```
y <- rnorm(10, m = 100, s = 15)
head(y)
```

```
[1]  96.57619  87.93997  93.17366 104.17639 113.21681 100.65354
```

- A large enough sample should look "normal", somewhat like the probability density function

```
mu <- 100
sigma <- 15
N <- 150
x <- rnorm(N, mean = mu, sd = sigma)
hist(x, prob = TRUE, main = "IQ Scores", xlab =
    "IQ", col = "grey70", breaks = 30)
```

Normal distribution defaults ...



Normal distribution defaults ...

- Do the sample statistics match the population parameters?

```
## Using x again, sample of N
mu
```

```
[1] 100
```

```
mean(x)
```

```
[1] 100.595
```

```
sigma
```

```
[1] 15
```

```
sd(x)
```

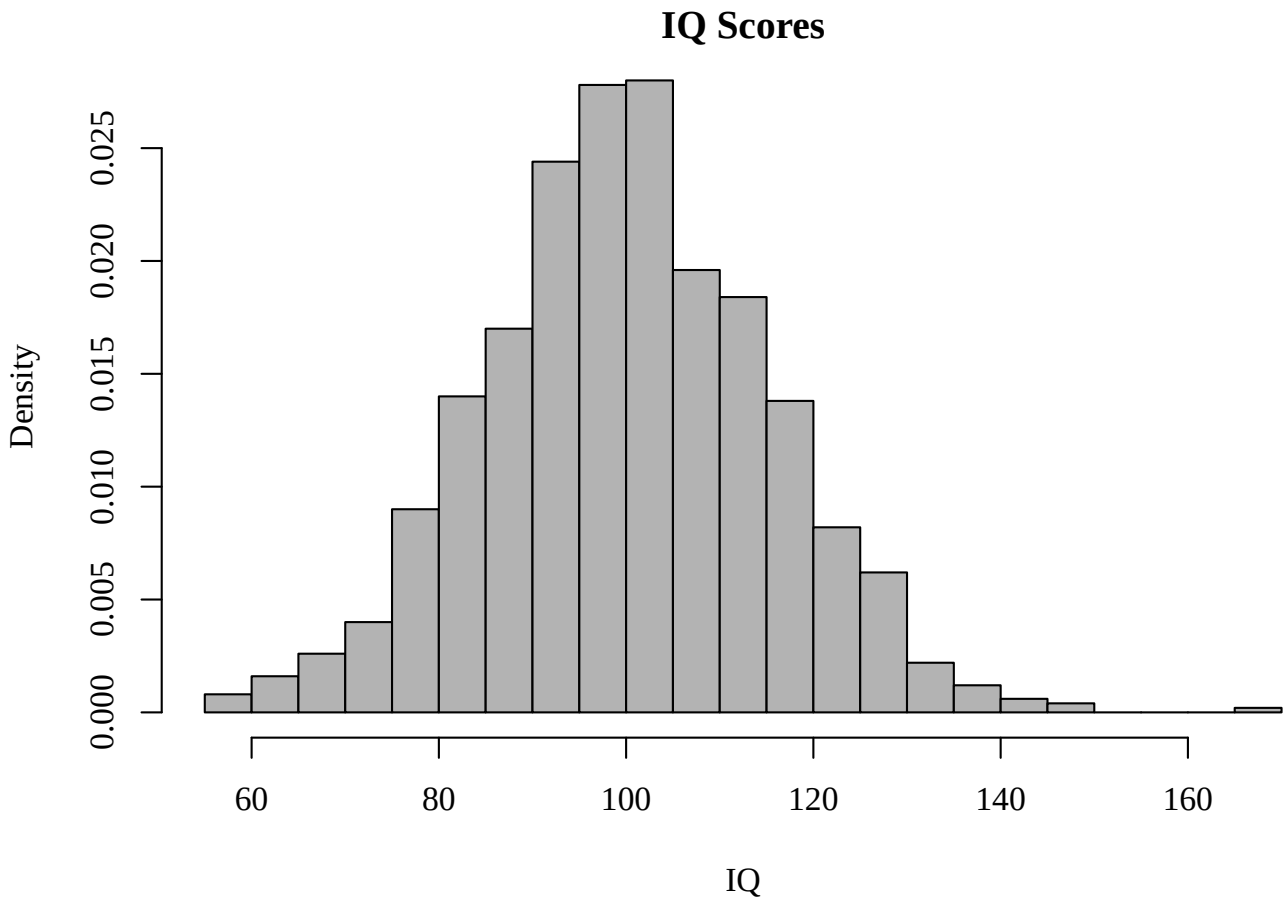
```
[1] 15.39491
```

Normal distribution defaults ...

- Questions
  1. Why don't they match exactly?
  2. What would make them match more closely?
- Bigger sample!

```
x.1000 <- rnorm(1000, mean = mu, sd = sigma)
hist(x.1000, prob = TRUE, main = "IQ Scores",
    xlab = "IQ", col = "grey70", breaks = 30)
```

Normal distribution defaults ...



That's what Statistics is All About!

- Quantify how much we expect the sample mean to differ from the population parameter.
- Remember the standard error?

$$SE = \frac{SD}{sqrt(N)}$$

- If the standard deviation is 15 and the sample size is 150, we expect SE to be the standard deviations of the estimates of the mean

```
sigma <- 15
N <- 150
stderr.theoretical <- sigma / sqrt(N)
stderr.theoretical
```

```
[1] 1.224745
```

What exactly is the SE? Let's use a Monte Carlo investigation

- To illustrate using Monte Carlo methods, we can draw several random samples of N = 150 from the same data generating process (e.g., "population"), saving the mean from each one.
- First, write a very simple function that will do this for one replication:

```
getSampleMean <- function(rep, N, M, SD){
     ## rep is an unused parameter, a place-holder
     x <- rnorm(N, mean = M, sd = SD)
     mean(x)
 }
```

Note the function arguments can have any name we want

- now apply it once to check that it works

What exactly is the SE? Let's use a Monte Carlo investigation ...

```
# Recall
N
```

```
[1] 150
```

```
mu
```

```
[1] 100
```

```
sigma
```

```
[1] 15
```

```
getSampleMean(1, N, mu, sigma)
```

What exactly is the SE? Let's use a Monte Carlo investigation ...

```
[1] 99.19473
```

- OK, now apply it 10,000 times (estimates of mu are called muhat here, $\hat{\mu}$)

```
set.seed(123)
muhat <- vapply(1:10000, getSampleMean, N = 150,
    M = mu, SD = sigma, numeric(1))
```

- print the first few means to see if it looks like the output you expected

```
head(muhat)
```

```
[1]  99.63456 101.39869  99.30975 100.96353  99.56550 100.93269
```

What exactly is the SE? Let's use a Monte Carlo investigation ...

- What is the mean of the sample means?

```
muhatmean <- mean(muhat)
muhatmean
```

```
[1] 99.98058
```

```
mu
```

```
[1] 100
```

```
mu - muhatmean
```

```
[1] 0.01941632
```

Pretty close! Mean of means approaches true mu as N approaches infinity

What exactly is the SE? Let's use a Monte Carlo investigation ...

- What is the SD of the sample means?

```
muhatsd <- sd(muhat)
stderr.theoretical
```

```
[1] 1.224745
```

```
muhatsd - stderr.theoretical
```

```
[1] -0.009331123
```

pretty close! SD of means approaches SE as N approaches infinity

- What's the point of this?
  - Just to test whether the formula for SE works? Maybe
  - Well, we drew random NORMAL numbers, so we would expect the normal-theory formula for SE to work.
  - Can now ask, "What if that assumption were violated?"

Violations of the Normality Assumption

- Suppose we were studying exam scores, with a ceiling effect at 100.
- Re-design previous function to return a vector (rep, mean, std.dev, std.err.)

```
getEstimates <- function(rep, N, M, SD) {
     x <- rnorm(N, mean = M, sd = SD)
     x <- ifelse(x > 100, 100, x)
     c(rep = rep, mean = mean(x), sd = sd(x),
        sterr = sd(x)/sqrt(N))
 }
set.seed(123)
```

- the estimates are returned as a matrix that has 4 rows and one column per replication.

Violations of the Normality Assumption ...

```
## Recall
mu
```

```
[1] 100
```

```
sigma
```

```
[1] 15
```

```
trunc.hat <- vapply(1L:10000L, getEstimates, N =
    150, M = mu, SD = sigma, numeric(4))
trunc.hat[ , 1:3]
```

```
              [,1]        [,2]        [,3]
rep      1.0000000   2.0000000   3.0000000
mean    94.1130062  95.1492024  93.5446341
sd       8.1099243   6.7194443   9.6017264
sterr    0.6621726   0.5486403   0.7839777
```

Violations of the Normality Assumption ...

- With sample of 150, the standard formula for the standard error of the mean is $SD/\sqrt{150}$

```
(trunc.mean <- mean(trunc.hat[2, ]))
```

```
[1] 94.00859
```

```
## Empirical standard error of the mean is:
(trunc.mean.sd <- sd(trunc.hat[2, ]))
```

```
[1] 0.7054973
```

```
## Mean of within sample estimates of stardard
    error:
(trunc.se.mean <- mean(trunc.hat[4, ]))
```

```
[1] 0.7127951
```

Violations of the Normality Assumption ...

```
## Theory-based std.err based using parameters
   (ignoring trunctation)
stderr.theory <- sigma/sqrt(N)
stderr.theory
```

```
[1] 1.224745
```

The normal theory-based standard error is much higher than the observed standard deviation of the means.

Generating random samples from population parameters

Basic regression model

- we want to ensure that our regression model is estimating a slope properly.
- generate data from a population space where the regression slope for X predicting Y is known.

```
## Generate 100 cases
N <- 100
## X is normally distributed with a mean of 0 and
    a sd of 10
## Y = b0 + b1*X + e
b0 <- 30
b1 <- 2
e.sigma <- 5
x.mu <- 0
x.sigma <- 5
## e = N(0, e.sigma^2)
```

Generating random samples from population parameters ...

```
error <- rnorm(N, 0, e.sigma)
x <- rnorm(N, x.mu, x.sigma)
dtest <- data.frame(x = x,
                    y = b0 + b1 * x + error,
                    ynoe = b0 + b1 * x)
head(dtest)
```

```
          x        y     ynoe
1  1.278349 39.82345 32.55670
2 -1.088651 20.58299 27.82270
3  6.819867 45.73093 43.63973
4  6.390948 38.18484 42.78190
5 -3.679375 26.32133 22.64125
6 -8.880630  8.06160 12.23874
```

```
m1 <- lm(y ~ x, data = dtest)
m1.summary <- summary(m1) ## Save summary in
    object for later
m1.summary
```

Generating random samples from population parameters ...

```
Call:
lm(formula = y ~ x, data = dtest)

Residuals:
     Min       1Q   Median       3Q      Max
-17.4355  -2.9898   0.3711   3.6924  12.0098

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 29.78017    0.50952   58.45   <2e-16 ***
x            1.99847    0.09424   21.20   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 5.094 on 98 degrees of freedom
Multiple R-squared:  0.8211,  Adjusted R-squared:  0.8192
F-statistic: 449.7 on 1 and 98 DF,  p-value: < 2.2e-16
```

```
rockchalk::plotSlopes(m1, plotx = "x")
```

- When estimating regressions, understand the data structures they generate

Generating random samples from population parameters ...

```
names(m1)
```

```
 [1] "coefficients"  "residuals"     "effects"       "rank"
     "fitted.values" "assign"
 [7] "qr"            "df.residual"   "xlevels"       "call"
     "terms"         "model"
```

```
coef(m1) ## just the betas
```

```
(Intercept)           x
  29.780173    1.998473
```

```
names(m1.summary)
```

```
 [1] "call"          "terms"         "residuals"     "coefficients"
     "aliased"       "sigma"
 [7] "df"            "r.squared"     "adj.r.squared" "fstatistic"
     "cov.unscaled"
```

Generating random samples from population parameters ...

```
coef(m1.summary) ## Parameter table
```

```
             Estimate Std. Error  t value     Pr(>|t|)
(Intercept) 29.780173 0.50952160 58.44732 5.451129e-78
x            1.998473 0.09424465 21.20516 2.138829e-38
```

- What if the variance of the error term is larger?

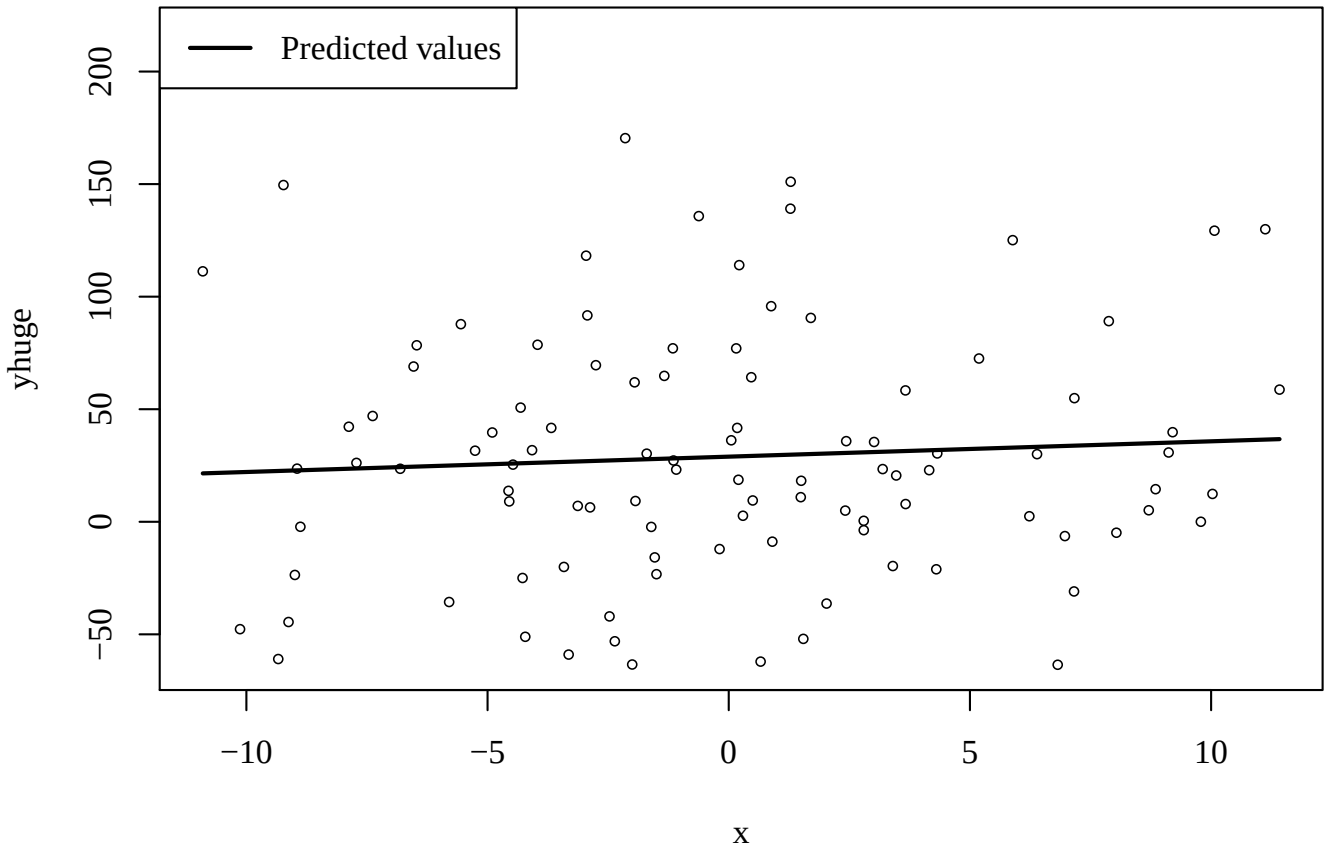
```
ehuge <- rnorm(NROW(dtest), 0, 50)
dtest$yhuge <- b0 + b1 * dtest$x + ehuge
m3 <- lm(yhuge ~ x, dtest)
summary(m3)
```

Generating random samples from population parameters ...

```
Call:
lm(formula = yhuge ~ x, data = dtest)

Residuals:
    Min      1Q  Median      3Q     Max
-97.091 -36.281  -4.686  34.494 142.950

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  28.9358     5.4270   5.332 6.24e-07 ***
x             0.6816     1.0038   0.679    0.499
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 54.26 on 98 degrees of freedom
Multiple R-squared:  0.004682,  Adjusted R-squared:  -0.005474
F-statistic: 0.461 on 1 and 98 DF,  p-value: 0.4988
```

```
rockchalk::plotSlopes(m3, plotx = "x")
```

Generating random samples from population parameters ...



- What if we forget the error term?

Generating random samples from population parameters ...

```
m2 <- lm(ynoe ~ x, data = dtest)
summary(m2)
```

```
Call:
lm(formula = ynoe ~ x, data = dtest)

Residuals:
       Min          1Q      Median          3Q         Max
-7.193e-15 -2.259e-15 -1.152e-15  9.000e-18  1.293e-13

Coefficients:
             Estimate Std. Error   t value Pr(>|t|)
(Intercept) 3.000e+01  1.326e-15 2.262e+16   <2e-16 ***
x           2.000e+00  2.453e-16 8.154e+15   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.326e-14 on 98 degrees of freedom
Multiple R-squared:      1, Adjusted R-squared:      1
F-statistic: 6.648e+31 on 1 and 98 DF,  p-value: < 2.2e-16
```

Generating random samples from population parameters ...

```
## Generates a warning
## Warning message: In summary.lm(m2) :
    essentially perfect fit: summary may be ##
    unreliable
```

Group mean differences

There is evidence that birth order (social more than biological) affects IQ (doi:10.1126/science.1141493).

- We'll create data with the predictor "first" (1 = first-born, 0 = not), and
- the true mean difference between those two populations is 5 IQ points.
- Group 1 is the first-born group, for which the mean is 103, while the mean for Group 0, the the ones who are not first born, is 98. The standard deviations within the 2 groups are equal to 15.
- Mean of group 1 can either be thought of as

$$98 + 5$$

  so 98 is the mean for humans and 5 is a bonus for first borns.
- Suppose that 40% of children are first-borns. (60% of children are 2nd or subsequent).
- The expected value of the IQ score for the entire population is

$$.4 * 103 + .6 * 98$$

KU

Group mean differences ...

- First, I have un-structured code that works

```
## Two groups, first = 0 or 1
set.seed(123)
firstprop <- 0.4
dat <- data.frame(first = rbinom(n = 100, size =
    1, prob = firstprop))
## We have just assigned rows of data into groups
    labeled 1 and 0
head(dat)
```

```
  first
1     0
2     1
3     0
4     1
5     1
6     0
```

Group mean differences ...

```
## Now we sample IQ scores, taking group
   membership into account.
## Here we use vectorized inputs to the data
   generator
dat$IQ <- rnorm(NROW(dat), m = 98 + 5 *
   dat$first, sd = 15)
## round to nearest whole number, since that is
   how IQ scores are
## reported
dat$IQ <- round(dat$IQ)
head(dat, 10)
```

Group mean differences ...

```
   first  IQ
1      0 102
2      1 103
3      0  97
4      1 124
5      1 100
6      0 121
7      0  75
8      1 112
9      0 100
10     0 101
```

- Test out various estimators, note results are all equivalent

```
## Do the sample statistics match the data
   generator (population) parameters?
## Several convenient ways to retrieve the
   answers
m1 <- lm(IQ ~ first, data = dat)
summary(m1)
```

Group mean differences ...

```
Call:
lm(formula = IQ ~ first, data = dat)

Residuals:
    Min      1Q  Median      3Q     Max
-33.425  -8.969  -0.817   8.727  33.183

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   97.817      1.884  51.920   <2e-16 ***
first          3.608      2.979   1.211    0.229
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 14.59 on 98 degrees of freedom
Multiple R-squared:  0.01475, Adjusted R-squared:  0.004698
F-statistic: 1.467 on 1 and 98 DF,  p-value: 0.2287
```

```
## Or

t.test(IQ ~ first, data = dat)
```

Group mean differences ...

```
  Welch Two Sample t-test

data:  IQ by first
t = -1.2193, df = 85.601, p-value = 0.2261
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -9.491625  2.274959
sample estimates:
mean in group 0 mean in group 1
       97.81667        101.42500
```

```
## Or
aggregate(IQ ~ first, data = dat,
              FUN = function(x) c(M = mean(x),
                  SD = sd(x)))
```

```
  first       IQ.M     IQ.SD
1     0   97.81667  14.78030
2     1  101.42500  14.30597
```

Group mean differences ...

```
## I'm sure we could find a more tedious way
   to get group differences, but this is near
   the maximum
diff(aggregate(IQ ~ first, data = dat,
   mean)$IQ)
```

```
[1] 3.608333
```

Plan a Monte Carlo Study

- We expect sampling variability, so the observed group difference will not be exactly 5
- And it isn't
- Let's use Monte Carlo methods to find out how much the mean-difference varies
- I prefer to think of any MC exercise as 3 chores
  1. Write a data-generator function
  2. Write a function that analyzes a data set
  3. Write a function that orchestrates the first 2 functions.

Plan a Monte Carlo Study ...

- Here's my data generator

```
##' Create one data set for the first born question
##'
##' This uses a vectorized call to rnorm
##' @param rep Integer to name repetition
##' @param N Sample Size
##' @param M1 Mean of first born group
##' @param M0 Mean of non-first born group
##' @param SD1 Standard Deviation of first born
##' @param SD0 Standard Deviation of non-first born
##' @return A data frame
##' @author Paul Johnson
## define a data-generator function for one replication
getData <- function(rep, N, M1, M0, SD1, SD0) {
     dat <- data.frame(first = rbinom(n = N, size = 1, prob = .4))
     dat$rep <- rep
     dat$IQ <- rnorm(N, m = M0 + dat$first * (M1 - M0),
                        s = SD0 + dat$first * (SD1 - SD0))
     dat$IQ <- round(dat$IQ)
     dat
 }
```

Plan a Monte Carlo Study ...

- Here's the analysis function

```
##' Calculate difference between groups
##'
##' This setup is lazy because it assumes the names
##' of the variables are simply "first" and "IQ".
##' I'd never do this in a real project.
##' @param dframe a data frame with input data
##' @return A floating point number for the difference
getDiff <- function(dframe){
     diff(aggregate(IQ ~ first, data = dframe, mean)$IQ)
 }
```

- Test that

```
## try it on one replication first
dat1 <- getData(1, N = 100, M1 = 103, M0 = 98, SD1 = 15, SD0 = 15)
getDiff(dat1)
```

```
[1] 7.899117
```

Plan a Monte Carlo Study ...

```
##
## Combine into 1 step if we don't want to save the data
getDiff(getData(1, N = 100, M1 = 103, M0 = 98, SD1 = 15, SD0 = 15))
```

```
[1] 5.374122
```

I'd run in debugger to make sure everything looks correct

- Do that lots of times

```
## Make a wrapper function
oneSim <- function(rep, N = 100, M1 = 103, M0 = 98, SD1 = 15, SD0 =
    15){
    getDiff(getData(1, N = N, M1 = M1, M0 = M0, SD1 = SD1, SD0 = SD0))
 }
## now do it 2000 times
## vapply here not different from R's replicate, but we have
## more control
set.seed(123)
myMeanDiffs <- vapply(1:2000, oneSim, N = 100,
                      M1 = 103, M0 = 98, SD1 = 15, SD0 = 15,
                      numeric(1))
```

Plan a Monte Carlo Study ...

```
## check results
mean(myMeanDiffs)
```

```
[1] 4.920373
```

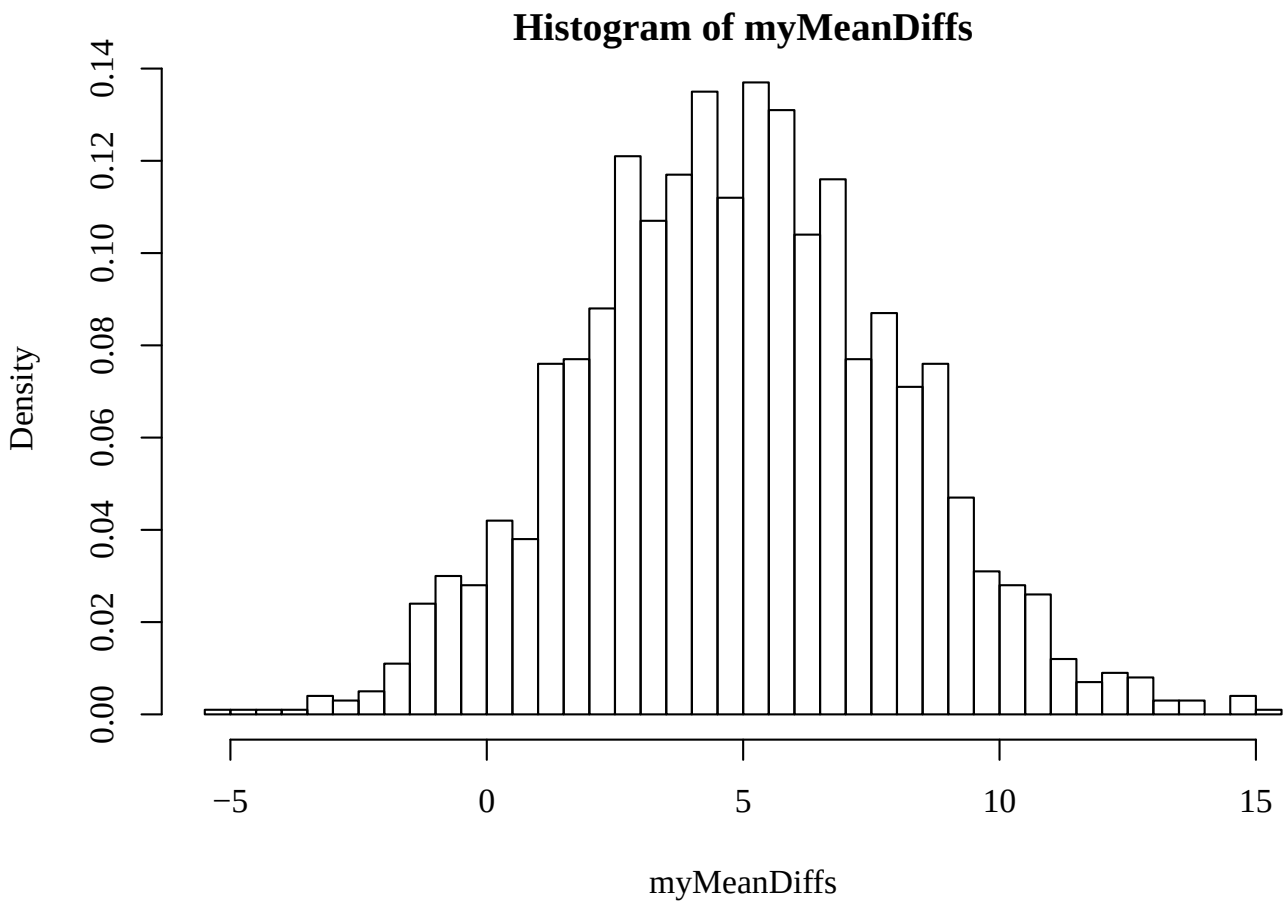
```
sd(myMeanDiffs)
```

```
[1] 3.073736
```

Plan a Monte Carlo Study ...

- What did we find out?

```
hist(myMeanDiffs, prob = TRUE, breaks = 30)
```



Remember the assumptions of a t-test?

T-test for difference of means historically assumed

- equal variances (or SD) in each group (called homoscedasticity)
- normally distributed data
- independent (uncorrelated) observations randomly sampled

Perhaps you've also heard that a t-test is "robust" to a moderate violation of normality

- You'll make about as many Type I errors with moderately non-normal data as you would with normal data.
- The t-test is also somewhat robust to heteroscedasticity (different variances), as long as the sample sizes are roughly equal.

Remember the assumptions of a t-test?

- Let's design a simulation to see what the effects of these factors are on the result of a t-test.
- Research question: Does violating these assumptions increase the probability of making a Type I error.

Problem: User interface for simulation

- Many years ago (when you were infants), some CRMDA GRAs decided to write a simulator that would receive parameters as colon-separated strings.
- For example, they would want to provide a parameter in a string like "40:20" and they wanted that to turn into a vector c(40, 20).
- First, we need to explore some string magic

```
x <- "40:20"
strsplit(x, ":")
```

```
[[1]]
[1] "40" "20"
```

```
## It is wrapped in an R list
unlist(strsplit(x, ":"))
```

```
[1] "40" "20"
```

Problem: User interface for simulation ...

```
## It is still characters, need numbers
as.numeric(unlist(strsplit(x, ":")))
```

```
[1] 40 20
```

- Create a function that can receive those strings for N, M and SD.

```
## Define a function for one replication
getTdata <- function(rep, N, M, SD) {
    ## tease apart two sample sizes
    Nvec <- as.numeric(unlist(strsplit(N, ":")))
    ## tease apart two means
    Mvec <- as.numeric(unlist(strsplit(M, ":")))
    ## tease apart two SDs
    SDvec <- as.numeric(unlist(strsplit(SD,
        ":")))
```

Problem: User interface for simulation ...

```
    ## assign dummy variables to each group's
       data set
    dat <- data.frame(first = c(rep(0, times =
       Nvec[1]),
                               rep(1, times =
                                  Nvec[2])))
    ## generate random IQ scores
    dat$IQ <- rnorm(sum(Nvec), m =
       Mvec[(dat$first + 1)],
                    sd = SDvec[(dat$first + 1)])
    dat$IQ <- round(dat$IQ)
    attr(dat, "rep") <- rep
    attr(dat, "parms") <- c(N = N, M = M, SD =
       SD)
    dat
}
```

Problem: User interface for simulation ...

```
## Test
dframe1 <- getTdata(1, N = "30:30", M = "98:102",
    SD = "15:15")
```

- I've used attributes to store copies of the rep number and the parms, in case I wanted to do record keeping

```
## Note the attributes stored with the data frame:
attributes(dframe1)
```

Problem: User interface for simulation ...

```
$names
[1] "first" "IQ"

$row.names
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
     23 24 25 26 27 28 29 30 31 32
[33] 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
     55 56 57 58 59 60

$class
[1] "data.frame"

$rep
[1] 1

$parms
       N         M        SD
 "30:30" "98:102"  "15:15"
```

```
## Individual attributes can be retrieved
attr(dframe1, "rep")
```

Problem: User interface for simulation ...

```
[1] 1
```

```
attr(dframe1, "parms")
```

```
      N         M        SD
"30:30" "98:102"  "15:15"
```

- The analysis function

```
##' A small wrapper to calculate a t-test
##' @param dframe A data frame
##' @param y character string for name of
   dependent variable. Default is "IQ"
##' @param x character string for name of
   independent variable. Default is "first"
##' @return We return only the p-value.
```

Problem: User interface for simulation ...

```
conductTtest <- function (dframe, y = "IQ", x =
    "first"){
    t.test(formula(paste(y, "~", x)), data =
        dframe, var.equal = TRUE)$p.value
 }
## Test it once, wrapping 2 function calls
    together
conductTtest(getTdata(1, N = "30:30", M =
    "98:103", SD = "15:15"))
```

```
[1] 0.02832306
```

Problem: User interface for simulation ...

```
## Create a one-step wrapper to put those together
runOneSim <- function(nreps, N, M, SD){
     df <- getTdata(1, N = "30:30", M = "98:103",
         SD = "15:15")
     reslt <- conductTtest(df)
     reslt
 }
```

```
## Now apply it 10 times to see the format of the
    output
sim10 <- sapply(1:10, runOneSim, N = "30:30", M =
    "98:103", SD = "15:15")
sim10
```

```
[1] 5.655317e-02 3.943166e-04 1.184527e-01 3.095466e-01 8.163624e-02
    4.154827e-05 4.396998e-01
[8] 5.338259e-01 3.253789e-01 2.306124e-01
```

Problem: User interface for simulation ...

```
## Oops, I did not snatch the attributes for
   records.
## Oops, I also forgot to store the rejection
   decision, so insert it

runOneSim <- function(rep, N, M, SD){
    dframe <- getTdata(rep, N = N, M = M, SD =
        SD)
    reslt <- conductTtest(dframe)
    parms <- attr(dframe, "parms")
    dframe2 <- data.frame(rep = attr(dframe,
        "rep"),
                pvalue = reslt, reject = if
                   (reslt <= 0.05) 1 else 0,
                N = parms["N"], M = parms["M"],
                   SD = parms["SD"])
  dframe2
```

Problem: User interface for simulation ...

```
  }
## test that
runOneSim(1, N = "30:30", M = "98:103", SD =
    "15:15")
```

```
  rep     pvalue reject     N       M     SD
N   1 0.1556859      0 30:30 98:103 15:15
```

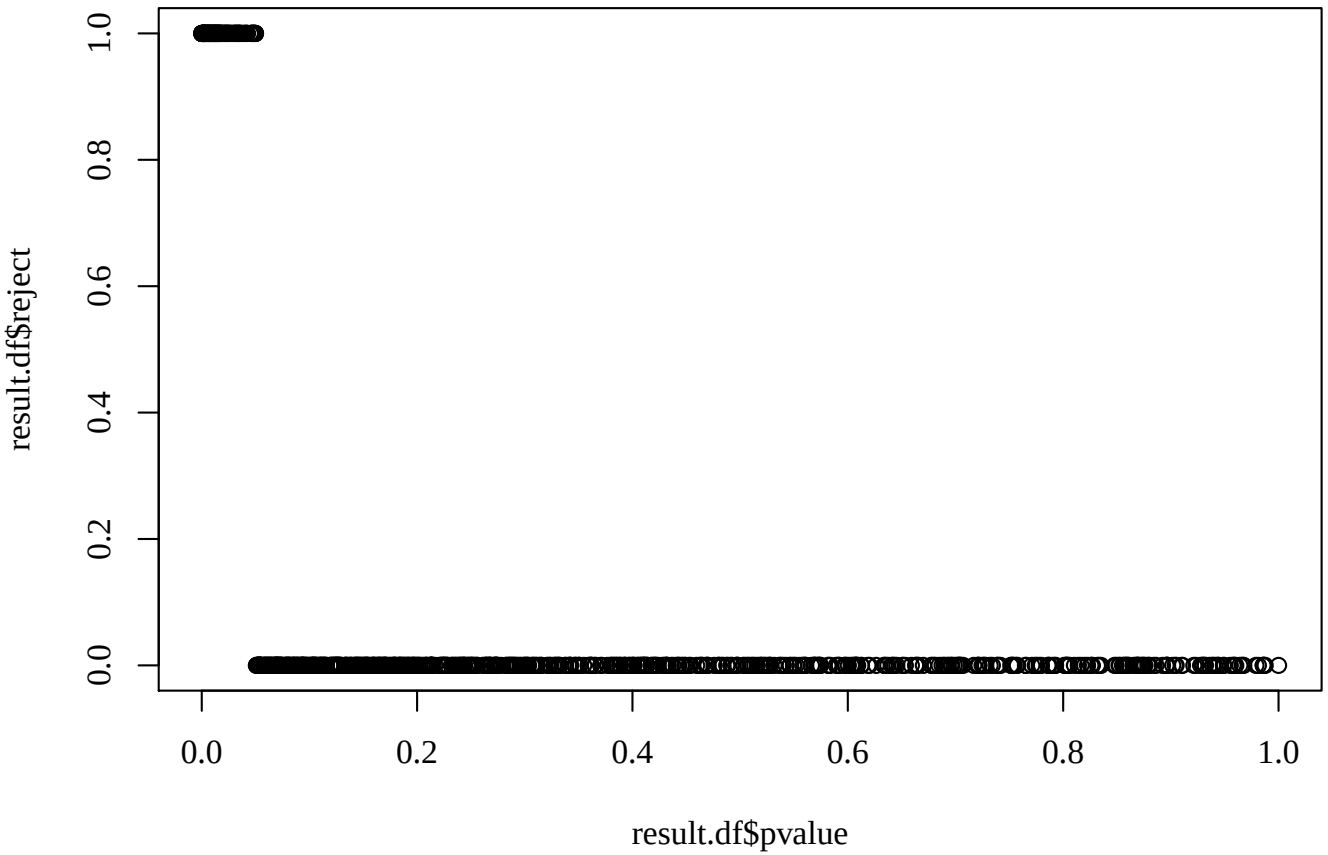
```
## Returns a list of one row data frames
set.seed(123)
nReps <- 1000
result.list <- lapply(1:nReps, runOneSim, N =
    "30:30", M = "98:103", SD = "15:15")
## Smash those down into one data frame with 1 row
## per replication
result.df <- do.call("rbind", result.list)
mean(result.df$reject)
```

Problem: User interface for simulation ...

```
[1] 0.248
```

```
plot(result.df$pvalue, result.df$reject)
```

Problem: User interface for simulation ...



Explore range of conditions

- R has a magic function named `expand.grid`
- It is easier to display it than to describe it.

```
expand.grid(x = c("a", "b", "c"), y = c("j",
    "k"), z = c(1, 2, 3))
```

```
   x y z
1  a j 1
2  b j 1
3  c j 1
4  a k 1
5  b k 1
6  c k 1
7  a j 2
8  b j 2
9  c j 2
10 a k 2
11 b k 2
12 c k 2
13 a j 3
14 b j 3
15 c j 3
16 a k 3
```

Explore range of conditions ...

```
17 b k 3
18 c k 3
```

- What did we get
  - a "mix and match" of elements, one per row
- Caution: It turned our character strings into factors:

```
eg <- expand.grid(x = c("a", "b", "c"), y =
    c("j", "k"), z = c(1, 2, 3))
## In document production, this causes an
    error. Should be OK interactively
## str(eg)
lapply(eg, class)
```

Explore range of conditions ...

```
$x
[1] "factor"

$y
[1] "factor"

$z
[1] "numeric"
```

- Prevent unwanted creation of R factors

```
eg <- expand.grid(x = c("a", "b", "c"),
                  y = c("j", "k"),
                  z = c(1, 2, 3),
                  stringsAsFactors = FALSE)
## str(eg)
lapply(eg, class)
```

Explore range of conditions ...

```
$x
[1] "character"

$y
[1] "character"

$z
[1] "numeric"
```

- In simulations, we usually have vectors of settings. We will use expand.grid to "mix and match" all of them.

Explore range of conditions

- Create a conds data frame to summarize the work to be done

```
## Set a variety of factors and compare p-values

## We need to choose levels of our predictors
   (sample size and SD)
## - equal v. unequal group sample sizes
## - equal v. unequal group variances
## - mean-difference: 0, 5, or 10
cond.N <- c("30:30", "40:20")
cond.SD <- c("10:20", "15:15", "20:10")
cond.M <- c("100:100") # for now, mean-difference
   = 0
## A fully crossed design runs all combinations
   of these levels.  This
## is a 2 (N) by 3 (SD) factorial design, so it
   has 2 * 3 = 6
```

Explore range of conditions ...

```
## conditions
conds <- expand.grid(maxReps = 3, SD = cond.SD, N
    = cond.N, M = cond.M,
                         stringsAsFactors = FALSE)
head(conds)
```

```
  maxReps     SD      N        M
1       3 10:20 30:30 100:100
2       3 15:15 30:30 100:100
3       3 20:10 30:30 100:100
4       3 10:20 40:20 100:100
5       3 15:15 40:20 100:100
6       3 20:10 40:20 100:100
```

- I did not think of an elegant approach, so here's my strategy.
- Create one more function that can receive the conds matrix and pick one row out of it. Let that function run as many simulations as we need, and return a data.frame.

Explore range of conditions ...

```
##' Tell this function the condition row to use,
##' and it creates a batch of simulations
runOneCondition <- function(i, conds){
    x <- conds[i, ]
    result.list <- lapply(1:x$maxReps, runOneSim,
                          N = x$N, M = x$M, SD = x$SD)
    do.call("rbind", result.list)
 }
allResults <- lapply(1:NROW(conds),
   runOneCondition, conds)
## Each sample drawn from a particular population
   is a "case" (like
## subjects).  We can easily combine our list of
   results as a single
## data set for analysis
output <- do.call(rbind, allResults)
head(output, 30)
```

Explore range of conditions ...

	rep	pvalue	reject	N	M	SD
N	1	0.42719499	0	30:30	100:100	10:20
N1	2	0.31725809	0	30:30	100:100	10:20
N2	3	0.57996600	0	30:30	100:100	10:20
N3	1	0.74422299	0	30:30	100:100	15:15
N11	2	0.41279370	0	30:30	100:100	15:15
N21	3	0.63699226	0	30:30	100:100	15:15
N4	1	0.55661831	0	30:30	100:100	20:10
N12	2	0.61536653	0	30:30	100:100	20:10
N22	3	0.83029046	0	30:30	100:100	20:10
N5	1	0.86315193	0	40:20	100:100	10:20
N13	2	0.18103430	0	40:20	100:100	10:20
N23	3	0.09913700	0	40:20	100:100	10:20
N6	1	0.80104357	0	40:20	100:100	15:15
N14	2	0.08603419	0	40:20	100:100	15:15
N24	3	0.04868749	1	40:20	100:100	15:15
N7	1	0.66700548	0	40:20	100:100	20:10
N15	2	0.25679324	0	40:20	100:100	20:10
N25	3	0.93516193	0	40:20	100:100	20:10

Explore range of conditions ...

```
## Finally, we are ready to run several
   replications in each condition.
conds$maxReps <- 1000
conds
```

```
  maxReps     SD      N         M
1    1000 10:20 30:30 100:100
2    1000 15:15 30:30 100:100
3    1000 20:10 30:30 100:100
4    1000 10:20 40:20 100:100
5    1000 15:15 40:20 100:100
6    1000 20:10 40:20 100:100
```

Explore range of conditions ...

```
## Our Monte Carlo study will take a few moments
   to run
set.seed(123)
bigResults <- lapply(1:NROW(conds),
   runOneCondition, conds)
stackedResults <- do.call(rbind, bigResults)
## Now summarize the rejection rate for each
   condition
output <- aggregate(reject ~ N + SD, data =
   stackedResults, FUN = mean)
names(output) <- c("N", "SD", "Type.I.Rate")
output
```

Explore range of conditions ...

```
      N     SD Type.I.Rate
1 30:30 10:20        0.044
2 40:20 10:20        0.108
3 30:30 15:15        0.053
4 40:20 15:15        0.050
5 30:30 20:10        0.041
6 40:20 20:10        0.012
```

- The findings are sobering for t-testing with unequal sample sizes

```
## How does it perform when sample sizes are
    equal?
output[output$N == "30:30",]
```

```
      N     SD Type.I.Rate
1 30:30 10:20        0.044
3 30:30 15:15        0.053
5 30:30 20:10        0.041
```

Explore range of conditions ...

```
## Unequal?
output[output$N != "30:30",]
```

```
      N     SD Type.I.Rate
2 40:20 10:20        0.108
4 40:20 15:15        0.050
6 40:20 20:10        0.012
```

- The corrected version of the t-test. Does it reduce the problem?

```
## Since 15 years ago, R's default t-test
    uses Welch's
## correction for difference in variance.

## A replacement for the t-test function
conductTtest <- function (dframe, y = "IQ", x
    = "first"){
    t.test(formula(paste(y, "~", x)), data =
        dframe)$p.value
```

Explore range of conditions ...

```
 }
set.seed(123)
bigResults <- lapply(1:NROW(conds),
    runOneCondition, conds)
stackedResults <- do.call(rbind, bigResults)
## Now summarize the rejection rate for each
    condition
output <- aggregate(reject ~ N + SD, data =
    stackedResults, FUN = mean)
names(output) <- c("N", "SD", "Type.I.Rate")
output
```

```
      N     SD Type.I.Rate
1 30:30 10:20       0.040
2 40:20 10:20       0.047
3 30:30 15:15       0.053
4 40:20 15:15       0.048
5 30:30 20:10       0.041
6 40:20 20:10       0.047
```

Explore range of conditions ...

- Yes!

Outline

Advice for Monte Carlo Designers

DO NOT:

- Think of the Monte Carlo experiment as "One Giant Sequential Script" of commands
- Generate a massive block of data that needs to be saved and re-loaded every time you run a procedure on it

Rather, create separate functions that

1. Generate and manipulate data for one "run" of the simulation
   - May receive a random seed for replication purposes
   - Handles all of the data-related changes (impose missingness, etc.)
2. Accept & analyze 1 data set (Runs one complete replication, saves results)
3. Orchestrate repetition of the above steps
4. Harvest estimates, summarize/plot results

KU

Example of what to NOT Do

R code generated by ML-Pow-SIM

```
###     A programme to obtain the power of parameters in 2 level
#       balanced model  with  Normal response
#                    generated on 09/11/16
###~~~~~~~~~~~~~~~~~~~~    Required packages  ~~~~~~~~~~~~~~~~~~~~~~~~###
    library(MASS)
    library(lme4)
###~~~~~~~~~~~~~~~~~~~~     Initial inputs    ~~~~~~~~~~~~~~~~~~~~~~~~###

set.seed(666)
siglevel<-0.025
z1score<-abs(qnorm(siglevel))
simus<-100
n1low<-5
n1high<-6
n1step<-1
n2low<-35
n2high<-40
n2step<-5
npred<-1
randsize<-1
beta<-c(0.00000,+.500000)
betasize<-length(beta)
effectbeta<-abs(beta)
sgnbeta<-sign(beta)
randcolumn<-0
xprob<-c(0,0.500000)
meanpred<-c(0,0.000000)
sigma2u<-matrix(c(1.000000),randsize,randsize)
sigmae<-sqrt(2.000000)
n1range<-seq(n1low,n1high,n1step)
n2range<-seq(n2low,n2high,n2step)
```

Example of what to NOT Do ...

```
n1size<-length(n1range)
n2size<-length(n2range)
totalsize<-n1size*n2size
finaloutput<-matrix(0,totalsize,6*betasize)
rowcount<-1
##-----------------        Inputs for model fitting       -----------------##

fixname<-c("x0","x1")
fixform<-"1+x1"
randform<-"(1|l2id)"
expression<-paste(c(fixform,randform),collapse="+")
modelformula<-formula(paste("y ~",expression))
data<-vector("list",2+length(fixname))
names(data)<-c("l2id","y",fixname)

#####--------- Initial input for power in two approaches ----------------#####

   powaprox<-vector("list",betasize)
    names(powaprox)<-c("b0","b1")
     powsde<-powaprox

cat("               The programme was executed at", date(),"\n")
cat("   ------------------------------------------------------------------\n")

 for(n2 in seq(n2low,n2high,n2step)){
  for(n1 in seq(n1low,n1high,n1step)){

                                          length=n1*n2
                                        x<-matrix(1,length,betasize)
                                       z<-matrix(1,length,randsize)
                                      l2id<-rep(c(1:n2),each=n1)
                                    sdepower<-matrix(0,betasize,simus)
                                  powaprox[1:betasize]<-rep(0,betasize)
                                 powsde<-powaprox
```

Example of what to NOT Do ...

```
cat(" Start of simulation for sample sizes of ",n1," micro and ",n2,"macro units\n")
  for(iter in 1:simus){

                      if(iter/10==floor(iter/10)){
                                                   cat(" Iteration remain=",simus-iter,"\n")
                                                  }
#######-----------       To set up X matrix          --------------########

               x[,2]<-rbinom(length,1,xprob[2])
#####-------------------------------------------------------------------#####
                  e<-rnorm(length,0,sigmae)
                   u<-mvrnorm(n2,rep(0,randsize),sigma2u)
                    fixpart<-x%*%beta
                     randpart<-rowSums(z*u[l2id,])
                      y<-fixpart+randpart+e
##------------------        Inputs for model fitting       ---------------##

  data$l2id<-as.factor(l2id)
  data$y<-y
    data$x0<-x[,1]
    data$x1<-x[,2]
###~~~~~~~~~~      Fitting the model using lmer funtion    ~~~~~~~~~~###

(fitmodel <- lmer(modelformula,data,REML=TRUE))

######~~~~~~~~~~   To obtain the power of parameter(s) ~~~~~~~~~~######

estbeta<-fixef(fitmodel)
 sdebeta<-sqrt(diag(vcov(fitmodel)))
  for(l in 1:betasize)
  {
   cibeta<-estbeta[l]-sgnbeta[l]*z1score*sdebeta[l]
    if(beta[l]*cibeta>0)              powaprox[[l]]<-powaprox[[l]]+1
```

Example of what to NOT Do ...

```
      sdepower[l,iter]<-as.numeric(sdebeta[l])
  }
##-------------------------------------------------------------------##
        } ##  iteration end here

 ###---------                  Powers and their CIs             ---------###

                         for(l in 1:betasize){

meanaprox<-powaprox[[l]]<-unlist(powaprox[[l]]/simus)
Laprox<-meanaprox-z1score*sqrt(meanaprox*(1-meanaprox)/simus)
Uaprox<-meanaprox+z1score*sqrt(meanaprox*(1-meanaprox)/simus)
meansde<-mean(sdepower[l,])
varsde<-var(sdepower[l,])
USDE<-meansde-z1score*sqrt(varsde/simus)
LSDE<-meansde+z1score*sqrt(varsde/simus)
powLSDE<- pnorm(effectbeta[l]/LSDE-z1score)
powUSDE<- pnorm(effectbeta[l]/USDE-z1score)
powsde[[l]]<-pnorm(effectbeta[l]/meansde-z1score)


    ###---------   Restrict the CIs within 0 and 1  ---------##
        if(Laprox<0) Laprox<-0
        if(Uaprox>1) Uaprox<-1
        if(powLSDE<0) powLSDE<-0
        if(powUSDE>1) powUSDE<-1

finaloutput[rowcount,(6*l-5):(6*l-3)]<-c(Laprox,meanaprox,Uaprox)
finaloutput[rowcount,(6*l-2):(6*l)]<-c(powLSDE,powsde[[l]],powUSDE)

                                        }

###~~~~~~~~~~            Set out the results in a data frame    ~~~~~~~~~###
```

Example of what to NOT Do ...

```
rowcount<-rowcount+1
cat("------------------------------------------------------------------\n")
                              } ## end of the loop  over the first level
                            } ## end of the loop  over the second level

 ###---------          Export output in a file                    ---------###

finaloutput<-as.data.frame(round(finaloutput,3))
 output<-data.frame(cbind(rep(n2range,each=n1size),rep(n1range,n2size),finaloutput))
  names(output)<-c("N","n","zLb0","zpb0","zUb0","sLb0","spb0","sUb0","zLb1","zpb1","zUb1","sLb1","spb1","sUb1"
write.table(output,"powerout.txt",sep="\t
      ",quote=F,eol="\n",dec=".",col.names=T,row.names=F,qmethod="double")
```

MC Designs from an ANOVA Point of View

- Think of a random sample as a person/case in a study
  - Multiple samples in each condition
- Between-subjects factors change the data-generating process
  - Parameters, distributions, missing data, scales
- Within-subjects factors analyze the same data using different methods – Estimation method, with/out covariates, N?

KU

Monte Carlo Outcomes

- Sampling distributions of... anything!
- Consistency, efficiency, normality
- Bias in point and SE estimates
- (Root) mean-squared error
- Confidence Interval coverage rates
- Hypothesis tests rejection rates (alpha, power)
- Model fit

Design your study to test hypotheses

Exploratory simulations can get out of hand

- Are all conditions necessary to test your hypotheses?
  - Consider how factors are expected to affect outcomes of interest, including interactions
- If exploring potential effects, try 2 levels of each parameter you want to explore.
  - That gives 2^k separate conditions for k parameters for pilot study
  - Reduce number of conditions by intentionally confounding higher-order interactions
- Alternative strategy
  - think of each parameter as a continuum,
  - draw parameters for a run from the continuous space

Write down a recipe to plan your study

Writing syntax can be daunting, so start with plain language

- Ingredients
  - Characteristics of your population(s)
  - Manipulated factors, outcomes of interest
- Write down steps from beginning to end
  - Can start broad, move to specific
  - Ultimately, easier to translate to R, C, Fortran if you remember what you are trying to do

KU

Variance Reduction Techniques

Save time and computing power, as well as reduce the amount of noise in your results

- When is it necessary to draw new samples?
  - NOT for factors like sample size, different estimators, prior variance, competing models
  - Typically, ONLY when the population differs (e.g., normal/nonnormal data), or the factor reflects an aspect of design that changes characteristics of the data (e.g., number of response categories)

Variance Reduction Techniques

- Consider sample size, etc., to be within- sample (or within-replication) factors
  - Recycle same seeds, or better yet, perform all analyses/conditions on the data the one time is generated
  - Generate largest N, then take first N_j from sample
  - Repeat this for # of replications, within each cell of between-replication design

Analysis Plan

- Carefully consider outcomes of interest
  - Have testable hypotheses/predictions
  - In each replication, save the output you intend to investigate, in a way that makes it easy to analyze
- Picture your analysis of results ahead of time
  - Perhaps make up data in a spreadsheet that mimics the format of your results
  - Could help your design

Useful Tools

- In R, the package portableParallelSeeds allows you to exercise great control of replicability using random seed-states
  - Developed at CRMDA, hosted on our KRAN server
  - To install and find help files:

```
CRAN <- "http://rweb.crmda.ku.edu/cran"
KRAN <- "http://rweb.crmda.ku.edu/kran"
options(repos = c(KRAN, CRAN))
install.packages("portableParallelSeeds", type =
    "source")
```

References

R Core Team (2017). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Session

```
sessionInfo()
```

```
R version 3.4.4 (2018-03-15)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 18.04 LTS

Matrix products: default
BLAS: /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.7.1
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.7.1

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
     LC_TIME=en_US.UTF-8
 [4] LC_COLLATE=en_US.UTF-8     LC_MONETARY=en_US.UTF-8
     LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C                  LC_ADDRESS=C
[10] LC_TELEPHONE=C             LC_MEASUREMENT=en_US.UTF-8
    LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  base

loaded via a namespace (and not attached):
```

Session ...

```
 [1] Rcpp_0.12.15          lattice_0.20-35     MASS_7.3-49
     grid_3.4.4            MatrixModels_0.4-1
 [6] nlme_3.1-137          rockchalk_1.8.111   SparseM_1.77
     minqa_1.2.4           nloptr_1.0.4
[11] car_2.1-6             Matrix_1.2-14       splines_3.4.4
     lme4_1.1-17           tools_3.4.4
[16] pbkrtest_0.4-7        parallel_3.4.4      compiler_3.4.4
     mgcv_1.8-23           nnet_7.3-12
[21] quantreg_5.35         methods_3.4.4
```