Nobody can figure this out mathematically, but it's pretty easy with simulation. Here's how to do it.

1. Using the random number generator in some software package, generate 20 independent chi-square values with one degree of freedom, and 20 independent chi-square values with two degrees of freedom.
2. Log transform all the values.
3. Compute the t-test.
4. Check to see if $p < 0.05$.

Do this a large number of times. The proportion of times $p < 0.05$ is the power — or more precisely, a Monte Carlo estimate of the power.

The number of times a statistical experiment is repeated is called the **Monte Carlo sample size**. How big should the Monte Carlo sample size be? It depends on how much precision you need. We will produce confidence intervals for all our Monte Carlo estimates, to get a handle on the probable margin of error of the statements we make. Sometimes, Monte Carlo sample size can be chosen by a power analysis. More details will be given later.

The example below shows several simulations of taking a random sample of size 20 from a standard normal population ($\mu = 0$, $\sigma^2 = 1$). Now actually, computer-generated random numbers are not *really* random; they are completely determined by the execution of the computer program that generates them. The most common (and best) random number generators actually produce a *stream* of pseudo-random numbers that will eventually repeat. In the good ones (and R uses a good one), "eventually" means after the end of the universe. So the pseudo-random numbers that R produces really *act* random, even though they are not. It's safe to say that they come closer to satisfying the assumptions of significance tests than any real data.

If you don't instruct it otherwise, R will use the system clock to decide on *where* in the random number stream it should begin. But sometimes you want to be able to reproduce the results of a simulation exactly, say if you're debugging your program, or you have already spent a lot of time making a graph based on it. In this case you can control the starting place in the random number stream, by setting the "seed" of the random number generator. The seed is a big integer; I used 12345 just as an example.

```
> rnorm(20) # 20 standard normals
 [1]  0.24570675 -0.38857202  0.47642336  0.75657595  0.71355871 -0.74630629
 [7] -0.02485569  1.93346357  0.15663167  1.16734485  0.57486449  1.32309413
[13]  0.63712982  2.00473940  0.04221730  0.70896768  0.42128470 -0.12115292
[19]  1.42043470 -1.04957255

> set.seed(12345) # Be able to reproduce the stream of pseudo-random numbers.
> rnorm(20)
 [1]  0.77795979 -0.89072813  0.05552657  0.67813726  0.80453336 -0.35613672
 [7] -1.24182991 -1.05995791 -2.67914037 -0.01247257 -1.22422266  0.88672878
[13] -1.32824804 -2.73543539  0.40487757  0.41793236 -1.47520817  1.15351981
[19] -1.24888614  1.11605686
> rnorm(20)
 [1]  0.866507371  2.369884323  0.393094088 -0.970983967 -0.292948278
 [6]  0.867358962  0.495983546  0.331635970  0.702292771  2.514734599
[11]  0.522917841 -0.194668990 -0.089222053 -0.491125596 -0.452112445
[16] -0.515548826 -0.244409517 -0.008373764 -1.459415684 -1.433710170

> set.seed(12345)
> rnorm(20)
 [1]  0.77795979 -0.89072813  0.05552657  0.67813726  0.80453336 -0.35613672
 [7] -1.24182991 -1.05995791 -2.67914037 -0.01247257 -1.22422266  0.88672878
[13] -1.32824804 -2.73543539  0.40487757  0.41793236 -1.47520817  1.15351981
[19] -1.24888614  1.11605686
```

The `rnorm` function is probably the most important random number generator, because it is used so often to investigate the properties of statistical tests that assume a normal distribution. Here is some more detail about `rnorm`.

```
> help(rnorm)
Normal                package:base                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.

       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.

     ‘qnorm’ is based on Wichura’s algorithm AS 241 which provides
     precise results up to about 16 digits.

Value:

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

References:

     Wichura, M. J. (1988) Algorithm AS 241: The Percentage Points of
     the Normal Distribution. Applied Statistics, 37, 477-484.

See Also:

     ‘runif’ and ‘.Random.seed’ about random number generation, and
     ‘dlnorm’ for the Lognormal distribution.

Examples:

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

After generating normal random numbers, the next most likely thing you might want to do is randomly scramble some existing data values. The `sample` function will select the elements of some array, either with replacement or without replacement. If you select all the numbers in a set without replacement, you've rearranged them in a random order. This is the basis of randomization tests. Sampling *with* replacement is the basis of the bootstrap.

```
> help(sample)
sample                package:base                R Documentation

Random Samples and Permutations

Description:

     ‘sample’ takes a sample of the specified size from the elements of
     ‘x’ using either with or without replacement.

Usage:

     sample(x, size, replace = FALSE, prob = NULL)

Arguments:

       x: Either a (numeric, complex, character or logical) vector of
          more than one element from which to choose, or a positive
          integer.

    size: A positive integer giving the number of items to choose.

 replace: Should sampling be with replacement?

    prob: A vector of probability weights for obtaining the elements of
          the vector being sampled.

Details:

     If ‘x’ has length 1, sampling takes place from ‘1:x’.

     By default ‘size’ is equal to ‘length(x)’ so that ‘sample(x)’
     generates a random permutation of the elements of ‘x’ (or ‘1:x’).

     The optional ‘prob’ argument can be used to give a vector of
     weights for obtaining the elements of the vector being sampled.
     They need not sum to one, but they should be nonnegative and not
     all zero.  If ‘replace’ is false, these probabilities are applied
     sequentially, that is the probability of choosing the next item is
     proportional to the probabilities amongst the remaining items. The
     number of nonzero weights must be at least 'size' in this case.

Examples:

     x <- 1:12
     # a random permutation
     sample(x)
     # bootstrap sampling
     sample(x,replace=TRUE)

     # 100 Bernoulli trials
     sample(c(0,1), 100, replace = TRUE)
```

6.4.1 Illustrating the Regression Artifact by Simulation

In the ordinary use of the English language, to "regress" means to go backward. In Psychiatry and Abnormal Psychology, the term "regression" is used when a person's behaviour changes to become more typical of an earlier stage of development — like when an older child starts wetting the bed, or an adult under extreme stress sucks his thumb. Isn't this a strange word to use for the fitting of hyperplanes by least-squares?

The term "regression" (as it is used in Statistics) was coined by Sir Francis Galton (1822-1911). For reasons that now seem to have a lot to do with class privilege and White racism, he was very interested in heredity. Galton was investigating the relationship between the heights of fathers and the heights of sons. What about the mothers? Apparently they had no height.

Anyway, Galton noticed that very tall fathers tended to have sons that were a bit shorter than they were, though still taller than average. On the other hand, very short fathers tended to have sons that were taller than they were, though still shorter than average. Galton was quite alarmed by this "regression toward mediocrity" or "regression toward the mean," particularly when he found it in a variety of species, for a variety of physical characteristics. See Galton's "Regression towards mediocrity in hereditary stature", *Journal of the Anthropological Institute* **15** (1886), 246-263. It even happens when you give a standardized test twice to the same people. The people who did the very best the first time tend to do a little worse the second time, and the people who did the very worst the first time tend to do a little better the second time.

Galton thought he had discovered a Law of Nature, though in fact the whole thing follows from the algebra of least squares. Here's a verbal alternative. Height is influenced by a variety of chance factors, many of which are *not* entirely shared by fathers and sons. These include the mother's height, environment and diet, and the vagaries of genetic recombination. You could say that the tallest fathers included some who "got lucky," if you think it's good to be tall (Galton did, of course). The sons of the tall fathers had some a genetic predisposition to be tall, but on average, they didn't get as lucky as their fathers in every respect. A similar argument applies to the short fathers and their sons.

This is the basis for the so-called **regression artifact**. Pre-post designs with extreme groups are doomed to be misleading. Programs for the disadvantaged "work" and programs for the gifted "hurt." This is a very serious methodological trap that has doomed quite a few evaluations of social programs, drug treatments – you name it.

Is this convincing? Well, the argument above may be enough for some people. But perhaps if it's illustrated by simulation, you'll be even more convinced. Let's find out.

Suppose an IQ test is administered to the same 10,000 students on two occasions. Call the scores `pre` and `post`. After the first test, the 100 individuals who did worst are selected for a special remedial program, but it does *nothing*. And, the 100 individuals who did best on the pre-test get a special program for the gifted, but it does *nothing*. We do a matched t-test on the students who got the remedial program, and a matched $t - test$ on the students who got the gifted program.

What should happen? If you followed the stuff about regression artifacts, you'd expect significant improvement from the students who got the remedial program, and significant deterioration from the students who got the gifted program – even though in fact, both programs are completely ineffective (and harmless). How will we simulate this?

According to classical psychometric theory, a test score is the sum of two independent pieces, the *True Score* and *measurement error*. If you measure an individual twice, she has the same True Score, but the measurement error component is different.

True Score and measurement error have population variances. Because they are independent, the variance of the observed score is the sum of the true score variance and the error variance. The proportion of the observed score variance that is True Score variance is called the test's *reliability*. Most "intelligence" tests have a mean of 100, a standard deviation of 15, and a reliability around 0.80.

So here's what we do. Making everything normally distributed and selecting parameter values so the means, standard deviations and reliability come out right, we

- Simulate 10,000 true scores.
- Simulate 10,000 measurement errors for the pre-test and an independent 10,000 measurement errors for the post-test.
- Calculate 10,000 pre-test scores by `pre = True + error1`.
- Calculate 10,000 post-test scores by `pre = True + error2`.
- Do matched t-tests on the individuals with the 100 worst and the 100 best pre-test scores.

This procedure is carried out *once* by the program `regart.R`. In addition, `regart.R` carries out a matched t-test on the entire set of 10,000 pairs, just to verify that there is no systematic change in "IQ" scores.

```
# regart.R    Demonstrate Regression Artifact
###################### Setup #######################
N <- 10000 ; n <- 100
truevar <- 180 ; errvar <- 45
truesd <- sqrt(truevar) ; errsd <- sqrt(errvar)
# set.seed(44444)
# Now define the function ttest, which does a matched t-test

ttest <- function(d) # Matched t-test. It operates on differences.
    {
    ttest <- numeric(4)
    names(ttest) <- c("Mean Difference"," t "," df "," p-value ")
    ave <- mean(d) ; nn <- length(d) ; sd <- sqrt(var(d)) ; df <- nn-1
    tstat <- ave*sqrt(nn)/sd
    pval <- 2*(1-pt(abs(tstat),df))
    ttest[1] <- ave ; ttest[2] <- tstat; ttest[3] <- df; ttest[4] <- pval
    ttest # Return the value of the function
    }
```

```
#####################################################

error1 <- rnorm(N,0,errsd) ; error2 <- rnorm(N,0,errsd)
truescor <- rnorm(N,100,truesd)
pre <- truescor+error1 ; rankpre <- rank(pre)

# Based on their rank on the pre-test, we take the n worst students and
# place them in a special remedial program, but it does NOTHING.

# Based on their rank on the pre-test, we take the n best students and
# place them in a special program for the gifted, but it does NOTHING.

post <- truescor+error2
diff <- post-pre # Diff represents "improvement."
                 # But of course diff = error2-error1 = noise

cat("\n") # Skip a line
cat("------------------------------------ \n")
dtest <- ttest(diff)
cat("Test on diff (all scores) \n") ; print(dtest) ; cat("\n")

remedial <- diff[rankpre<=n] ; rtest <- ttest(remedial)
cat("Test on Remedial \n") ; print(rtest) ; cat("\n")

gifted <- diff[rankpre>=(N-n+1)] ; gtest <- ttest(gifted)
cat("Test on Gifted \n") ; print(gtest) ; cat("\n")
cat("------------------------------------ \n")
```

The `ttest` function is a little unusual because it takes a whole vector of numbers (length unspecified) as input, and returns an array of 4 values. Often, functions take one or more numbers as input, and return a single value. We will see some more examples shortly. At the R prompt,

```
> source("regart.R")

------------------------------------
Test on diff (all scores)
Mean Difference              t             df         p-value
   1.872566e-02   1.974640e-01   9.999000e+03   8.434685e-01

Test on Remedial
Mean Difference              t             df         p-value
   7.192531e+00   8.102121e+00   9.900000e+01   1.449729e-12

Test on Gifted
Mean Difference              t             df         p-value
  -8.311569e+00  -9.259885e+00   9.900000e+01   4.440892e-15

------------------------------------
> source("regart.R")

------------------------------------
Test on diff (all scores)
Mean Difference              t             df         p-value
   2.523976e-02   2.659898e-01   9.999000e+03   7.902525e-01

Test on Remedial
Mean Difference              t             df         p-value
   5.510484e+00   5.891802e+00   9.900000e+01   5.280147e-08

Test on Gifted
Mean Difference              t             df         p-value
      -8.972938     -10.783356      99.000000        0.000000

------------------------------------
> source("regart.R")

------------------------------------
Test on diff (all scores)
Mean Difference              t             df         p-value
```

```
     0.0669827       0.7057641    9999.0000000       0.4803513

Test on Remedial
Mean Difference             t              df         p-value
   8.434609e+00    9.036847e+00    9.900000e+01    1.376677e-14

Test on Gifted
Mean Difference             t              df         p-value
      -8.371483      -10.215295       99.000000        0.000000

------------------------------------
```

The preceding simulation was unusual in that the phenomenon it illustrates happens virtually every time. In the next example, we need to use the Law of Large Numbers.

6.4.2 An Example of Power Analysis by Simulation

Suppose we want to test the effect of some experimental treatment on mean response, comparing an experimental group to a control. We are willing to assume normality, but *not* equal variances. We're ready to use an unequal-variances t-test, and we want to do a power analysis.

Unfortunately it's safe to say that nobody knows the exact non-central distribution of this monster. In fact, even the central distribution isn't exact; it's just a very good approximation. So, we have to resort to first principles. There are four parameters: $\theta = (\mu_1, \mu_2, \sigma_1^2, \sigma_2^2)$. For a given set of parameter values, we will simulate samples of size n_1 and n_2 from normal distributions, do the significance test, and see if it's significant. We'll do it over and over. By the Law of Large Numbers, the proportion of times the test is significant will approach the power as the Monte Carlo sample size (the number of data sets we simulate) increases.

The number we get, of course, will just be an *estimate* of the power. How accurate is the estimate? As promised earlier, we'll accompany every Monte Carlo estimate of a probability with a confidence interval. Here's the formula. For the record, it's based on the normal approximation to the binomial, not bothering with a continuity correction.

$$\widehat{P} \pm z_{1-\frac{\alpha}{2}} \sqrt{\frac{\widehat{P}(1-\widehat{P})}{m}} \tag{6.1}$$

This formula will be implemented in the S function `merror` for "margin of error."

```
merror <- function(phat,m,alpha) # (1-alpha)*100% merror for a proportion
    {
    z <- qnorm(1-alpha/2)
    merror <- z * sqrt(phat*(1-phat)/m)  # m is (Monte Carlo) sample size
    merror
    }
```

The Monte Carlo estimate of the probability is denoted by $\widehat{P}$, the quantity m is the Monte Carlo sample size, and $z_{1-\alpha/2}$ is the value with area $1-\frac{\alpha}{2}$ to the left of it, under the standard normal curve. Typically, we will choose $\alpha = 0.01$ to get a 99% confidence interval, so $z_{1-\alpha/2} = 2.575829$.

How should we choose m? In other words, how many data sets should we simulate? It depends on how much accuracy we want.Since our policy is to accompany Monte Carlo estimates with confidence intervals, we will choose the Monte Carlo sample size to control the width of the confidence interval.

According to Equation (6.1), the confidence interval is an estimated probability, plus or minus a margin of error. The margin of error is $z_{1-\frac{\alpha}{2}}\sqrt{\frac{\widehat{P}(1-\widehat{P})}{m}}$, which may be viewed as an *estimate* of $z_{1-\frac{\alpha}{2}}\sqrt{\frac{P(1-P)}{m}}$. So, for any given probability we are trying to estimate, we can set the desired margin of error to some small value, and solve for m. Denoting the *criterion* margin of error by c, the general solution is

$$m = \frac{z^2_{1-\frac{\alpha}{2}}}{c^2} P(1-P), \tag{6.2}$$

which is implemented in the S function `mmargin`.

```
mmargin <- function(p,cc,alpha)
           # Choose m to get (1-alpha)*100% margin of error equal to cc
           {
           mmargin <- p*(1-p)*qnorm(1-alpha/2)^2/cc^2
           mmargin <- trunc(mmargin+1) # Round up to next integer
           mmargin
           } # End definition of function mmargin
```

Suppose we want a 99% confidence interval around a power of 0.80 to be accurate to plus or minus 0.01.

```
> mmargin(.8,.01,.01)
[1] 10616
```

The table below shows Monte Carlo sample sizes for estimating power with a 99% confidence interval.

Table 6.1: Monte Carlo Sample Size Required to Estimate Power with a Specified 99% Margin of Error

	Power Being Estimated					
Margin of Error	0.70	0.75	0.80	0.85	0.90	0.99
0.10	140	125	107	85	60	7
0.05	558	498	425	339	239	27
0.01	13,934	12,441	10,616	8,460	5,972	657
0.005	55,734	49,762	42,464	33,838	23,886	2,628
0.001	1,393,329	1,244,044	1,061,584	845,950	59,7141	65,686

It's somewhat informative to see how the rows of the table were obtained.

```
> wpow <- c(.7,.75,.8,.85,.9,.99)
> mmargin(wpow,.1,.01)
[1] 140 125 107  85  60   7
> mmargin(wpow,.05,.01)
[1] 558 498 425 339 239  27
> mmargin(wpow,.01,.01)
[1] 13934 12441 10616  8460  5972   657
> mmargin(wpow,.005,.01)
[1] 55734 49762 42464 33838 23886  2628
> mmargin(wpow,.001,.01)
[1] 1393329 1244044 1061584  845950  597141   65686
```

Equations (6.1) and (6.2) are general; they apply to the Monte Carlo estimation of *any* probability, and Table 6.1 applies to any Monte Carlo estimation of power. Let's return to the specific example at hand. Suppose we the population standard deviation of the Control Group is 2 and the standard deviation of the Experimental Group is 6. We'll let the population means be $\mu_1 = 1$ and $\mu_2 = 3$, so that the difference between population means is half the *average* within-group population standard deviation.

To select a good starting value of n, let's pretend that the standard deviations are equal to the average value, and we are planning an ordinary two-sample t-test. Referring to formula (4.4) for the non-centrality parameter of the non-central F-distribution, we'll let $q = \frac{1}{2}$; this is optimal when the variances are equal. Since $\delta = \frac{1}{2}$, we have $\phi = n\,q(1-q)\,\delta^2 = \frac{n}{16}$. Here's some S. It's short — and sweet. Well, maybe it's an acquired taste. It's also true that I know this problem pretty well, so I knew a good range of n values to try.

```
> n <- 125:135
> pow <- 1-pf(qf(.95,1,(n-2)),1,(n-2),(n/16))
> cbind(n,pow)
        n       pow
 [1,] 125 0.7919594
 [2,] 126 0.7951683
 [3,] 127 0.7983349
 [4,] 128 0.8014596
 [5,] 129 0.8045426
 [6,] 130 0.8075844
 [7,] 131 0.8105855
 [8,] 132 0.8135460
 [9,] 133 0.8164666
[10,] 134 0.8193475
[11,] 135 0.8221892
```

We will start the unequal variance search at $n = 128$. And, though we are interested in more accuracy, it makes sense to start with a target margin of error of 0.05. The idea is to start out with rough estimation, and get more accurate only once we think we are close to the right n.

```
> n1 <- 64 ; mu1 <- 1 ; sd1 <- 2 # Control Group
> n2 <- 64 ; mu2 <- 3 ; sd2 <- 6 # Experimental Group
>
> con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)

> help(t.test)
```

The output of help is omitted, but we learn that the default is a test assuming unequal variances – just what we want.

```
> t.test(con,exp)

Welch Two Sample t-test

data:  con and exp
t = -2.4462, df = 78.609, p-value = 0.01667
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.4632952 -0.3556207
sample estimates:
mean of x mean of y
 1.117435  3.026893

> t.test(con,exp)[1]
$statistic
        t
-2.446186

> t.test(con,exp)[3]
$p.value
[1] 0.01667109
>
> m <- 500 # Monte Carlo sample size (Number of simulations)
> numsig <- 0 # Initializing
> for(i in 1:m)
+     {
+     con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)
+     numsig <- numsig+(t.test(con,exp)[3]<.05)
+     }
> pow <- numsig/m
> cat ("Monte Carlo Power = ",pow,"\n") ; cat ("\n")
Monte Carlo Power =  0.708

> m <- 500 # Monte Carlo sample size (Number of simulations)
> numsig <- 0 # Initializing
> for(i in 1:m)
+     {
+     con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)
+     numsig <- numsig+(t.test(con,exp)[3]<.05)
+     }
> pow <- numsig/m
> cat ("Monte Carlo Power = ",pow,"\n") ; cat ("\n")
Monte Carlo Power =  0.698
```

Try it again.

```
>
> m <- 500 # Monte Carlo sample size (Number of simulations)
> numsig <- 0 # Initializing
> for(i in 1:m)
+     {
+     con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)
+     numsig <- numsig+(t.test(con,exp)[3]<.05)
+     }
> pow <- numsig/m
> cat ("Monte Carlo Power = ",pow,"\n") ; cat ("\n")
Monte Carlo Power =  0.702
```

Try a larger sample size.

```
> n1 <- 80 ; mu1 <- 1 ; sd1 <- 2 # Control Group
> n2 <- 80 ; mu2 <- 3 ; sd2 <- 6 # Experimental Group
> m <- 500 # Monte Carlo sample size (Number of simulations)
> numsig <- 0 # Initializing
> for(i in 1:m)
+     {
+     con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)
+     numsig <- numsig+(t.test(con,exp)[3]<.05)
+     }
> pow <- numsig/m
> cat ("Monte Carlo Power = ",pow,"\n") ; cat ("\n")
Monte Carlo Power =  0.812
```

Try it again.

```
> n1 <- 80 ; mu1 <- 1 ; sd1 <- 2 # Control Group
> n2 <- 80 ; mu2 <- 3 ; sd2 <- 6 # Experimental Group
> m <- 500 # Monte Carlo sample size (Number of simulations)
> numsig <- 0 # Initializing
> for(i in 1:m)
+     {
+     con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)
+     numsig <- numsig+(t.test(con,exp)[3]<.05)
+     }
> pow <- numsig/m
> cat ("Monte Carlo Power = ",pow,"\n") ; cat ("\n")
Monte Carlo Power =  0.792
```

It seems that was a remarkably lucky guess. Now seek margin of error around 0.01.

```
>
> m <- 10000 # Monte Carlo sample size (Number of simulations)
> numsig <- 0 # Initializing
> for(i in 1:m)
+     {
+     con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)
+     numsig <- numsig+(t.test(con,exp)[3]<.05)
+     }
> pow <- numsig/m
> cat ("Monte Carlo Power = ",pow,"\n") ; cat ("\n")
Monte Carlo Power =  0.8001

> merror <- function(phat,m,alpha) # (1-alpha)*100% merror for a proportion
+     {
+     z <- qnorm(1-alpha/2)
+     merror <- z * sqrt(phat*(1-phat)/m)  # m is (Monte Carlo) sample size
+     merror
+     }
> margin <- merror(.8001,10000,.01) ; margin
[1] 0.01030138
> cat("99% CI from ",(pow-margin)," to ",(pow+margin),"\n")
99% CI from  0.7897986  to  0.810
```

This is very nice, except that I can't believe equal sample sizes are optimal when the variances are unequal. Let's try sample sizes proportional to the standard deviations, so $n_1 = 40$ and $n_2 = 120$. The idea is that perhaps the two population means should be estimated with roughly the same precision, and we need a bigger sample size in the experimental condition to compensate for the larger variance. Well, actually I chose the relative sample sizes to minimize the standard deviation of the sampling distribution of the difference between means — the quantity that is estimated by the denominator of the t statistic.

```
> n1 <- 40 ; mu1 <- 1 ; sd1 <- 2 # Control Group
> n2 <- 120 ; mu2 <- 3 ; sd2 <- 6 # Experimental Group
> m <- 500 # Monte Carlo sample size (Number of simulations)
> numsig <- 0 # Initializing
> for(i in 1:m)
+     {
+     con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)
+     numsig <- numsig+(t.test(con,exp)[3]<.05)
+     }
> pow <- numsig/m
> cat ("Monte Carlo Power = ",pow,"\n") ; cat ("\n")
Monte Carlo Power =  0.89

> margin <- merror(pow,m,.01)
> cat("99% CI from ",(pow-margin)," to ",(pow+margin),"\n")
99% CI from  0.8539568  to  0.9260432
>
> # This is promising. Get some precision.
>
> n1 <- 40 ; mu1 <- 1 ; sd1 <- 2 # Control Group
> n2 <- 120 ; mu2 <- 3 ; sd2 <- 6 # Experimental Group
> m <- 10000 # Monte Carlo sample size (Number of simulations)
> numsig <- 0 # Initializing
> for(i in 1:m)
+     {
+     con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)
+     numsig <- numsig+(t.test(con,exp)[3]<.05)
+     }
> pow <- numsig/m
> cat ("Monte Carlo Power = ",pow,"\n") ; cat ("\n")
```

```
Monte Carlo Power =  0.8803

> margin <- merror(pow,m,.01)
> cat("99% CI from ",(pow-margin)," to ",(pow+margin),"\n")
99% CI from  0.8719386  to  0.8886614
```

So again we see that power depends on *design* as well as on effect size and sample size. It will be left as an exercise to find out how much sample size we could save (over the $n_1 = n_2 = 80$ solution) by taking this into account in the present case.

Finally, it should be clear that R has a t-test function, and the custom function `ttest` was unnecessary. What other classical tests are available?

```
> library(help=ctest)
ctest           Classical Tests

Description:

Package: ctest
Version: 1.4.0
Priority: base
Title: Classical Tests
Author: Kurt Hornik <Kurt.Hornik@ci.tuwien.ac.at>, with major
  contributions by Peter Dalgaard <p.dalgaard@kubism.ku.dk> and
  Torsten Hothorn <Torsten.Hothorn@rzmail.uni-erlangen.de>.
Maintainer: R Core Team <R-core@r-project.org>
Description: A collection of classical tests, including the
  Ansari-Bradley, Bartlett, chi-squared, Fisher, Kruskal-Wallis,
  Kolmogorov-Smirnov, t, and Wilcoxon tests.
License: GPL

Index:

ansari.test              Ansari-Bradley Test
bartlett.test            Bartlett Test for Homogeneity of Variances
binom.test               Exact Binomial Test
chisq.test               Pearson's Chi-squared Test for Count Data
```

cor.test	Test for Zero Correlation
fisher.test	Fisher's Exact Test for Count Data
fligner.test	Fligner-Killeen Test for Homogeneity of Variances
friedman.test	Friedman Rank Sum Test
kruskal.test	Kruskal-Wallis Rank Sum Test
ks.test	Kolmogorov-Smirnov Tests
mantelhaen.test	Cochran-Mantel-Haenszel Chi-Squared Test for Count Data
mcnemar.test	McNemar's Chi-squared Test for Count Data
mood.test	Mood Two-Sample Test of Scale
oneway.test	Test for Equal Means in a One-Way Layout
pairwise.prop.test	Pairwise comparisons of proportions
pairwise.t.test	Pairwise t tests
pairwise.table	Tabulate p values for pairwise comparisons
pairwise.wilcox.test	Pairwise Wilcoxon rank sum tests
power.prop.test	Power calculations two sample test for of proportions
power.t.test	Power calculations for one and two sample t tests
print.pairwise.htest	Print method for pairwise tests
print.power.htest	Print method for power calculation object
prop.test	Test for Equal or Given Proportions
prop.trend.test	Test for trend in proportions
quade.test	Quade Test
shapiro.test	Shapiro-Wilk Normality Test
t.test	Student's t-Test
var.test	F Test to Compare Two Variances
wilcox.test	Wilcoxon Rank Sum and Signed Rank Tests