R functions for the known probability distributions

```
#random generation from known Chi-square distribution
x <- rchisq(100, df=20) #arguments are n and degrees of freedom

#dchisq takes random observations from a distribution (or a sequence of #'s) #and returns the prob
abilities that would be associated with those numbers
# according to a given pdf, with parameters you set.
pdf = dchisq(x, df = 20)

#pchisq also takes a vector of values, but this time assigns the
#associated value from the specified CDF
cdf = pchisq(x, df =20)

#qchisq takes probabilities (values form 0 to 1) and assigns the value of
#the random variable that would be associated with the quantile equivalent
#to the probability provided to the function
#So for qchisq, df = 20, entering: .01, .1, .25, .5, .75, .9, and .99
# should return numbers like 5, 12, 15, 20, 23, 27, 35 ... [20 is the mean]
q <- qchisq(seq(.01,.99,length.out=100),df=20)
```

Visualization

```
par(mfrow=c(2,2))
hist(x, main = "distribution of x")
plot(x,pdf, main = "pdf")
lines(density(x),col = "blue",lty=2,lwd=2) #density() can be
used for a curve
plot(x,cdf, main = "CDF")
plot(q,sort(x), main = "Q-Q plot") #equivalent to qqplot() fu
nction
abline(0,1, col = "2") #add the 45-degree line (for reference
)
```

Topic 1 - Methods of Simulation

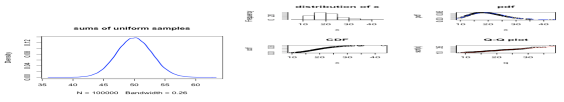
Inverse Transform Method

```
# 1. Take the antiderivative of the pdf (the CDF)
# 2. Take the inverse of that CDF
# 3. For this CDF, substitute runif(n) for x
# 4. The n random-uniform X's will be transformed to the orig
inal pdf
# Note: the inverse function of a function can be found using
optim()

pdf <- function(x) {3 * x^2}
CDF <- function(x) {x^3}
#In this case the inverse is a known result (no need to solve
for it)
inverse <- function(x) {x^(1/3)}
# To produce a random sample from the pdf f(x)=(3 * x^2) of s
ize 100000
sample <- inverse(runif(100000))
hist(sample, prob = T, main = expression(f(x)==3 * x^2))
x <- seq(0,1,.01)
lines(x, 3*x^2, col = "blue", lwd = 2) #overlay the pdf
```

Generate a normal distribution: sums of uniform samples

```
#The sums of uniform random random samples will be normally d
ist.
Y = matrix(runif(100000 * 100), nrow=100000, ncol=100)
X <- rowSums(Y)
plot(density(X), col = "blue", lwd = 2,main="sums of uniform
samples")
```



Topic 2: Monte Carlo simulation

1. Sample randomly from a known (or theorized) distribution (many times!)
2. Compute measures of bias etc for the resulting parameters estimated

a) Bias = E[theta.hat - theta]
b) SD = sqrt(E[(theta.hat - theta)^2])
c) MSE = E[(theta.hat - theta)^2] Note, MSE is approx = (SD)^2 + Bias^2

Example

-Evaluate s^2 as an estimate of sigma^2 for a standard normal pop.

```
#create function to produce a vector of s^2 estimates, from a specified # of
#ind. rnorm samples; all with the same n, mu, and SD parameters:
s.2.estimates <- function(n, replications){
  R <- replications
  out <- NULL #empty object for generating output list
  #compute R s^2 estimates
  for (i in 1:R) {
    x <- rnorm(n, 0 , 1) #generate random normal, size = n, mu = 0, sd = 1
    mean <- mean(x) #mean for the first of R samples
    deviations <- x - mean #compute the distance of each x from the mean
    sq.devs <- deviations^2 #square the deviations from the mean
    out[i] <- sum(sq.devs)/(n-1) #sum the squares, and divide by n - 1
  }
  bias.estimate <- mean(out) - 1 #compute bias over all estimates of s^2
  sd.estimate <- sqrt((1/(R-1))*(sum((out-mean(out))^2)))
  mse.estimate <- sd.estimate^2 + bias.estimate^2 # variance + bias^2
  #create output object (named list in this case)
   return(list(s.2.estimates = out,
               Bias = bias.estimate,
               SD = sd.estimate,
               MSE = mse.estimate,
               n = n,
               R = replications))
}
run1 <- s.2.estimates(n = 100, replications  = 10000)
str(run1)


## List of 6
##  $ s.2.estimates: num [1:10000] 0.899 0.852 1.163 1.435 0.804 ...
##  $ Bias         : num 0.000851
##  $ SD           : num 0.14
##  $ MSE          : num 0.0196
##  $ n            : num 100
##  $ R            : num 10000

mean(run1$s.2.estimates)

## [1] 1.000851

# [1] 0.9980755
```

By running this computation 10000 times, we confirm the sample variance is a relatively unbiased estimator, as theorized.

Additionally, the theoretical variance and MSE of the s^2 estimate are both: (2*(sigma^2))/(n - 1)

In this case this comes out to 2/99 = .02; and sqrt(.02) = .14, the SD obtained.

Topic 3: Optimization (univariate/two-dimensional)

Method 1: Golden Section [USES ONLY f(x)]

1. Iteratively finds a MINIMUM
2. Evaluates the function at startings points a0 and b0
3. Moves to a1 and b1
4. If f(a1) < f(b1), then it assumes the minimum is between [a0 and b1]
5. Now b1 becomes b0, search in contracted interval between a0 and new b0
6. Continue until the distance between a0 and b0 is less than some previously set criteria. When the distance is approximately zero, f(a) and f(b) will be converging to the minimum value of f.

Mathematically, this procedure is based on the fact that the CDF is a function of x that returns the probability of x. So its inverse is a function that takes a probability (between 0 and 1) and returns x! Thus, entering a random uniform variable between 0 and 1 into the inverse CDF of x will generate a random x from its distribution.

Choose the distance by which each iteration will reduce the length of the interval

Ending distances can thus be (1 - distance)^N by the the Nth iteration. Set up the function of f, starting interval, and ending length (distance)

Summary:

-Compares the value of $f(x)$ at ends of a random interval [a, b] -If $f(a) > f(b)$, constrict the next search interval to [a, $\frac{b-a}{2}$] -End when $f(a) = f(b)$ (or when f(a) - f(b) is very near zero. -Formula for number of iterations:

```
rho <- 2-sqrt(5)/2 ###THIS IS THE GOLDEN RATIO
precision <- 1e-6 #some VERY small number
int <- c(-5,5)#starting interval in which you think the minim
a exists
ceiling(log(precision/(diff(int)))/log(1-rho))

## [1] 8
```

```
#CEILING FUNCTION WILL Round up to the nearest Integer
```

$$CEILING of \frac{log("precision")}{log("golden.ratio")}$$

Method 2: Bisection [USES f(x) and f'(x)]

1. f(x) must be unimodal
2. f(x) is two-dimensional
3. Finds the MINIMUM by:

a) evaluating f'(x)

b) stopping when f'(x) = 0. ####Summary: -Compares the slope of $f'(x)$ at ends of a random interval [a, b] -If the slope at a (or b) is negative, proceed towards the minimum -End when $f'(x) = 0$.

Essentially the same process as the golden section, except that it selects one point evaluate, assesses the direction of the slope (+/-), and if negative chooses to evaluate f'(x) at a point further right, and as point further left if the slope of f'(x) was posiitive. ###Formula for iteration count

$$CEILING of \frac{log("precision")}{log(.5)}$$

Method 3: Newton's method

1. Given $f(x)$, define $q(x)$ such that at x_i:

$$q(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{1}{2}f''(x_i)(x - x_i)^2$$

2. This function approximates the trajectory of (is tangnt to) $f(x)$ at each given point.

Thus:

I. $q'(x) = f'(x)$
II. $q''(x) = f''(x)$

2. Minimize $q(x)$ by setting its derivative equal to zero:

$$q'(x) = 0 = f'(x_i) + f''(x_i)(x - x_i)$$

3. From this formula we obtain:

$$x = x_i - \frac{f'(x_i)}{f''(x_i)}$$

4. EXAMPLE 1. $f(x) = x^4 - 14x^3 + 60x^2 - 70x$ $f'(x) = 4x^3 - 42x^2 + 120x - 70$ $f''(x) = 12x^2 - 84x + 120$

When $f(x_i) = .5$, $q(x)$ has a minimum at .75.

Thus, $x = .75$, when $q'(x) = 0$, given that $q(x)$ was constructed with $f(x_{=}.5)$

USING R FUNCTIONS

1. optimize

```
#uses function, and a range to search
#FINDS A MINIMUM(default) OR MAX depending on argument "maxim
um"
#optimize(f, lower = 4, upper = 15, maximum = TRUE)
```

2. Optim

```
#uses the log-likelihood function
#FINDS A MINIMUM
#CAN BE USED WITH A TWO VARIABLE DIST if you set each variable
#To a different column of the one x variable in the function,
#then pass it through the optim function
#optim(c(1,1), func)
```

3. The Newton method then takes .75 as the new "x_i" [or "$x_i + 1$"], and repeats the process until q'(x) is minimized. ###In R

```
newton <- function(f_prime, f_dbl, start, precision = 1e-5) {
  #Compute the first iteration to obtain x and x_i, and their difference
  x_old <- as.numeric(start)
  x_new <- x_old - f_prime(x_old)/f_dbl(x_old)
  diff <- x_new - x_old
  i <- 1
  #if the difference is above the desired precision, repeat
  while (abs(diff) > precision) {
    x_old <- x_new #switch the new value into the position of the old
    x_new <- x_old - f_prime(x_old)/f_dbl(x_old) #compute a new, new value
    i <- i + 1
    diff <- x_new - x_old #compute a new difference, the while-loop will now restart
  }
  return(list(x = x_new, iterations = 1))
}
newton(f_prime = function(x) (4*x^3-42*x^2+120*x-70),
       f_dbl = function(x) (12*x^2-84*x+120),
       start = .56)

## $x
## [1] 0.7808841
##
```

The output is a value of x where the min or max of $f(x)$ exists:

```
f <- function(x) (x^4-14*x^3+60*x^2-70*x)
curve(f,xlim=c(0,2), ylim=c(-30,10),lwd=3,main ="f(x) and f(.
7808841)",ylab="y")
par(new=T)
abline(h = f(.7808841), lwd=3,lty=2,col=2)
```

Issues:

1. The Newton (or Newton-Raphson) method is better for finding local min or max

```
#Test the function using starting points 1, 3.3, and 5.5:
local.root <- NULL #This will capture the x value produced
for (i in 1:3) {
  local.root[i]<- newton(f_prime = function(x) (4*x^3-42*x^2+
120*x-70),
       f_dbl = function(x) (12*x^2-84*x+120),
       start = c(1, 3.3, 5.5)[i])$x
}
output <- matrix(local.root,ncol=1)
rownames(output) <- c("Starting point 1:", "Starting point 3.
3:",
                      "Starting point 5.5:")
colnames(output) <- c("Location of max/min")
output

##                      Location of max/min
## Starting point 1:              0.7808841
## Starting point 3.3:            3.7619213
## Starting point 5.5:            5.9571947
```

Method 4: Secant method

This method is identical to the Newton method, except that when calculating the new value for x, us$x_{i+1} = x_i - \frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})} f(x_i)$e:

Topic 4: Resampling (not bootstrap)

Idea: -Re-sample a portion ("training set") of observed data -Fit a model. -Compute the Mean Squared Prediciton Error -Use the model to predict on the rest of your data ("test") portion -Recompute the Mean Squared Prediction Error.

Two ways to produce these results: 1. LOOCV a. Take one row of data "out" b. Fit the model c. Predict that one left-out line 2. K-fold CV a. Split the data into k portions (say 3) b. use k = 1 as "training" c. Test the model on 2+3 d. Repeat with "train" being k = 2, k=3... k=k Comparison: a. LOOCV has greater bias reduction b. k-fold reduces variability c. When K = exactly 5, or K= 10, it is emprically shown to be preferred

For k-fold CV:

```
#mean(predict("model", newdata = test) - test$y)^2
```

$$MSPE_k = \frac{1}{n_k}\sum_{i=1}^{k}(y_{hat} - y_i)^2$$

This describes when you use the model fit on "training", but y_i are the y values from the "test" dataset. NOTICE: summation is over k-folds.

For the LOOCV method.

```
#ordinary mean squared residuals, but each obtained when we left out
#one of the n observations.
#mean(predict("model") - test$y)^2
```

$$MSPE = \sum_{i=1}^{n}(y_{hat} - y_i)^2$$

NOTICE: here the summation is over n, because we obtain MSE once for each time we left out each of the n observations at a time.

Topic 5: Bootstrap

Idea: 1. Re-sample WITH replacement 2. re-run your test statistic 3. check its SE using the formula: $SE = \sqrt{\frac{1}{B-1}\sum_{r=1}^{B}[\mu^* - mean(\mu^*)]}$Here, the μ^* are the estimates of the test stat from each of the bootstrap samples.