Stat 5201, Spring 2011, Handout# 1

Getting a random sample of units

Suppose I have a population of values, let's suppose there are five of them:

```
> y <- c(19, 22, 20, 21, 24)
```

These might be a population of $N = 5$ ages, for example.

The R function `sample` can be used to get a SRS of size n. Here are random samples of size 2 and of size 3:

```
> set.seed(123)
> sample(y, 2)

[1] 22 21

> sample(y, 3)

[1] 20 21 24
```

I used the `set.seed` function so that I get the same sample every time—good for handout writing, but not always desirable. If I wanted a SRSWR, I could use

```
> sample(y, 2, replace = TRUE)

[1] 19 20

> sample(y, 3, replace = TRUE)

[1] 24 20 20
```

Here is a few more complex ways to do the same thing. First, rather than sampling `y`, I will sample the **index** vector `1:length(y)`, and then get the indices:

```
> set.seed(123)
> sel <- sample(1:5, 2)
> y[sel]

[1] 22 21

> sel <- sample(1:5, 3)
> y[sel]

[1] 20 21 24
```

If the `sample` function didn't exist, I could get a sample by: (1) generating `length(y)` uniform random numbers:

```
> (rnums <- runif(length(y)))
```

```
[1] 0.04556 0.52811 0.89242 0.55144 0.45661
```

(2) using `order`, which returns a permutation which rearranges its first argument into ascending order

```
> (or <- order(rnums))

[1] 1 5 2 4 3
```

The smallest of the `rnums` is in location 1, and second smallest in location 5, and so on. Using `or` to select the members of `y` that have the smallest corresponding random numbers will give a SRS:

```
> y[or[1:2]]

[1] 19 24
```

All possible samples from an SRS

R is a functional language: you write functions, and then execute them with arguments of your choice. Here is a very simple function:

```
> myadd <- function(a, b = 3) {
+     out <- a + b
+     out
+ }
```

The function in named `myadd`. It has two arguments, `a` and `b`. If you don't set `b`, then it is set to its default value of `3`, but you must give a value for `a`. The left curly bracket `{` opens the function, and the final right-curly bracket `}` ends it. The code in the function uses the arguments to compute something. Variables like `out` that are computed inside the function are generally not visible outside the function. The last line of the function is the returned value, in this case the sum of the arguments:

```
> myadd(2, 6)

[1] 8

> myadd(2)

[1] 5

> myadd(b = 6, a = 2)

[1] 8

> out

Error: object 'out' not found
```

Functions can be useful for organizing your work. For example, the function below finds all possible SRSs of size $n = 2$ and computes summary statistics for each. It prints them, and then computes and prints more summary statistics. After the printing is done, I don't what the function to return anything, so the last line of the function is `invisible()`.

```
> allsamp2 <- function(y) {
+     out <- NULL
+     N <- length(y)
+     Ybar <- mean(y)
+     S2 <- var(y)
+     for (i in 1:(N - 1)) {
+         for (j in (i + 1):N) {
+             samp <- y[c(i, j)]
+             ybar <- mean(samp)
+             sqerror <- (ybar - Ybar)^2
+             out <- rbind(out, c(i = i, j = j, ybar = ybar,
+                 sqerror = sqerror))
+         }
+     }
+     print(out)
+     cat("Average of the ybar =", mean(out[, 3]), "\n")
+     cat("Population Ybar     =", Ybar, "\n")
+     cat("Mean Squared Error  =", mean(out[, 4]), "\n")
+     cat("Population variance =", S2, "\n")
+     cat("S2/n * (1-f)        =", (S2/2) * (1 - 2/N),
+         "\n")
+     invisible()
+ }
```

This functions uses lots of built-in R functions, like `length` to get the length of a vector, `mean` to compute the mean, `var` for the variance (dividing by $n - 1$ by default). Inside the for loops I computed each sample is mean and squared error from the true value of VY. I used `print` to print these values. I used `cat` to do more printing. You can get help for any of these function in R by typing, for example `?mean` or `help(mean)`.

Calling this function with the vector `y` defined previously,

```
> allsamp2(y)

      i j ybar sqerror
 [1,] 1 2 20.5    0.49
 [2,] 1 3 19.5    2.89
 [3,] 1 4 20.0    1.44
 [4,] 1 5 21.5    0.09
 [5,] 2 3 21.0    0.04
 [6,] 2 4 21.5    0.09
 [7,] 2 5 23.0    3.24
```

```
 [8,] 3 4 20.5    0.49
 [9,] 3 5 22.0    0.64
[10,] 4 5 22.5    1.69
Average of the ybar = 21.2
Population Ybar     = 21.2
Mean Squared Error  = 1.11
Population variance = 3.7
S2/n * (1-f)        = 1.11
```

To contrast this output, I have written a second function called `allsamp3` that only differs by taking samples of size 3. The only difference is in changing the `for` loops to have three indices rather than 2. I've also computed a few additional quantities: the value of $s_m = \sqrt{s^2/n)(1-f)}$, and an indicator of whether the $\bar{Y} \in (\bar{y} - 2s_m, \bar{y} + 2s_m)$, to be discussed in class.

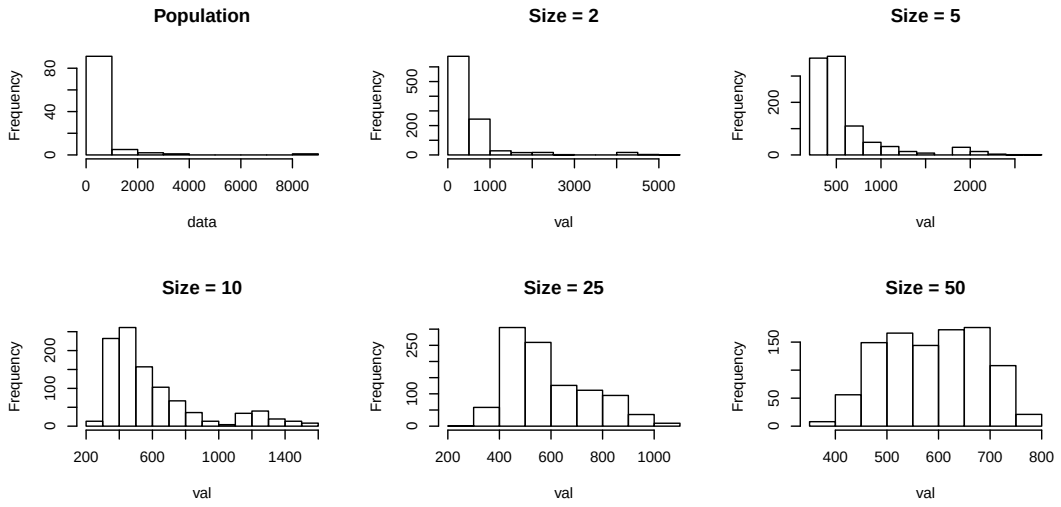
```
> allsamp3(y, z = 2)

      i j k  ybar sqerror  smean good
 [1,] 1 2 3 20.33 0.75111 0.5578    1
 [2,] 1 2 4 20.67 0.28444 0.5578    1
 [3,] 1 2 5 21.67 0.21778 0.9189    1
 [4,] 1 3 4 20.00 1.44000 0.3651    0
 [5,] 1 3 5 21.00 0.04000 0.9661    1
 [6,] 1 4 5 21.33 0.01778 0.9189    1
 [7,] 2 3 4 21.00 0.04000 0.3651    1
 [8,] 2 3 5 22.00 0.64000 0.7303    1
 [9,] 2 4 5 22.33 1.28444 0.5578    0
[10,] 3 4 5 21.67 0.21778 0.7601    1
Average of the ybar = 21.2
Population Ybar     = 21.2
Mean Squared Error  = 0.4933
Population variance = 3.7
S2/n * (1-f)        = 0.4933
```

43 ## Distribution of the Mean in Large Samples

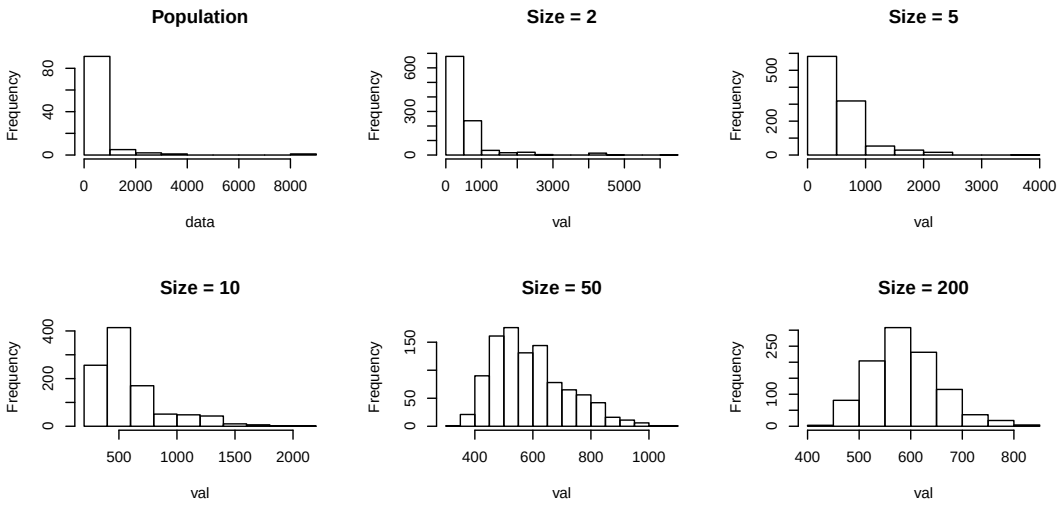
```
> mean.simulation <- function(data, sizes = c(2, 5, 10,
+     25, 50), nsamp = 1000, replace = FALSE, ...) {
+     par(mfrow = c(2, 3))
+     hist(data, main = "Population", ...)
+     for (size in sizes) {
+         val <- rep(0, nsamp)
+         for (j in 1:nsamp) val[j] <- mean(sample(data,
+             size, replace = replace))
+         hist(val, main = paste("Size =", size))
+     }
+     invisible()
+ }
```

44 The data for this example comes from the populations of the 100 largest US cities.

```
> data <- read.csv("http://tinyurl.com/4tea8js", header = TRUE)
> mean.simulation(data$Population/1000)
```




```
> data <- read.csv("http://tinyurl.com/4tea8js", header = TRUE)
> mean.simulation(data$Population/1000, replace = TRUE,
+     sizes = c(2, 5, 10, 50, 200))
```




```
> mean.simulation(log10(data$Population))
```