This section will briefly outline two general concepts, GLS and `ggplot`. As an addendum, we will go over the recurring `R` questions that cropped up during the midterm. We will examine the characteristics of generalized least squares (GLS), and specifically the efficiency gains from a special case of GLS, weighted least squares (WLS). This is the econometric concepts. We will then recreate the graphs from Figures 2.6 and 2.7, roughly, in the notes using `ggplot2` a very popular graphing package in `R`. This is part is optional, especially since it is only a very brief treatment of the package — there is a lot more to learn.

Let $x \sim U(0, 2000)$ and $\epsilon \sim N(0, (x/1000)^2)$. The underlying data generating process in (2.102) is $y_i = \alpha + x_i\beta + \epsilon$, where $\alpha = 0.5$ and $\beta = 1.5$. The objective is to plot the simulated sampling distribution of the OLS estimator applied to $B = 10,000$ draws, each of size $n = 1000$. First, let's generate the sample data for one draw.

```
n <- 1000
x <- runif(n, min=0, max=2000)
eps <- rnorm(n, 0, sqrt((x/1000)^2))
y <- 0.5 + x*1.5 + eps
```

Now we can calculate the standard OLS parameter vector $[\hat{\alpha} \;\; \hat{\beta}]'$ by noting that $\mathbf{X}$ is just the x vector bound to a column of ones. We will only examine $\hat{\beta}$ for this section, rather than both parameters.

```
X <- cbind(1, x)
params <- solve(t(X) %*% X) %*% t(X) %*% y
beta <- params[2]
print(beta)
```

```
[1] 1.500033
```

Let's package this into a function, called `rnd.beta`, so that we can collect the OLS parameter for an arbitrary number of random samples, noting that n is a constant so we may as well keep it out of the function so that 1000 is not reassigned thousands of times to n.

```
rnd.beta <- function(i) {
  x <- runif(n)
  eps <- rnorm(n, 0, sqrt(x/10))
  y <- 0.5 + x*1.5 + eps
  X <- cbind(1, x)
  params <- solve(t(X) %*% X) %*% t(X) %*% y
  beta <- params[2]
  return(beta)
}
```

Since there aren't any supplied arguments, the function will return an estimated $\hat{\beta}$ from a a different random sample for each call:

```
rnd.beta()
rnd.beta()
```

```
[1] 1.503556
[1] 1.496052
```

This is convenient for bootstrapping without loops, but rather applying the function to a list of effective indices.[1] Now replicating the process for B draws is straightforward:

[1] This is much more comfortable for me, with a background in functional programming. There is some inherent value, however, in keeping code compact by mapping across indices rather than incrementing an index within a `for` loop; the code is more readable and less prone to typos.

```
B <- 1000
beta.vec <- sapply(1:B, rnd.beta)
mean(beta.vec)
```

```
[1] 1.500078
```

Alright. Looking good. The average of the simulated sample is much closer to β than any individual call of `rnd.beta`, suggesting that the distribution of the simulated parameters will be appropriately centered. Now, let's create another, similar function that returns the WLS estimates.

```
rnd.wls.beta <- function(i) {
  x <- runif(n)
  y <- 0.5 + x*1.5 + rnorm(n, 0, sqrt(x/10))
  C <- diag(1/sqrt(x/10))
  y.wt <- C %*% y
  X.wt <- C %*% cbind(1, x)
  param.wls <- solve(t(X.wt) %*% X.wt) %*% t(X.wt) %*% y.wt
  beta <- param.wls[2]
  return(beta)
}
wls.beta.vec <- sapply(1:B, rnd.wls.beta)
```

We now have two collections of parameter estimates, one based on OLS and another based on WLS. It is straightforward to plot two separate histograms using `R`'s core histogram plotting function `hist()`. However, we can use this to introduce a more flexible, powerful graphing package called `ggplot2`.

```
library(ggplot2)
labels <- c(rep("ols", B), rep("wls", B))
data <- data.frame(beta=c(beta.vec, wls.beta.vec), method=labels)
ggplot(data, aes(x=beta, fill=method)) + geom_density(alpha=0.2)
```

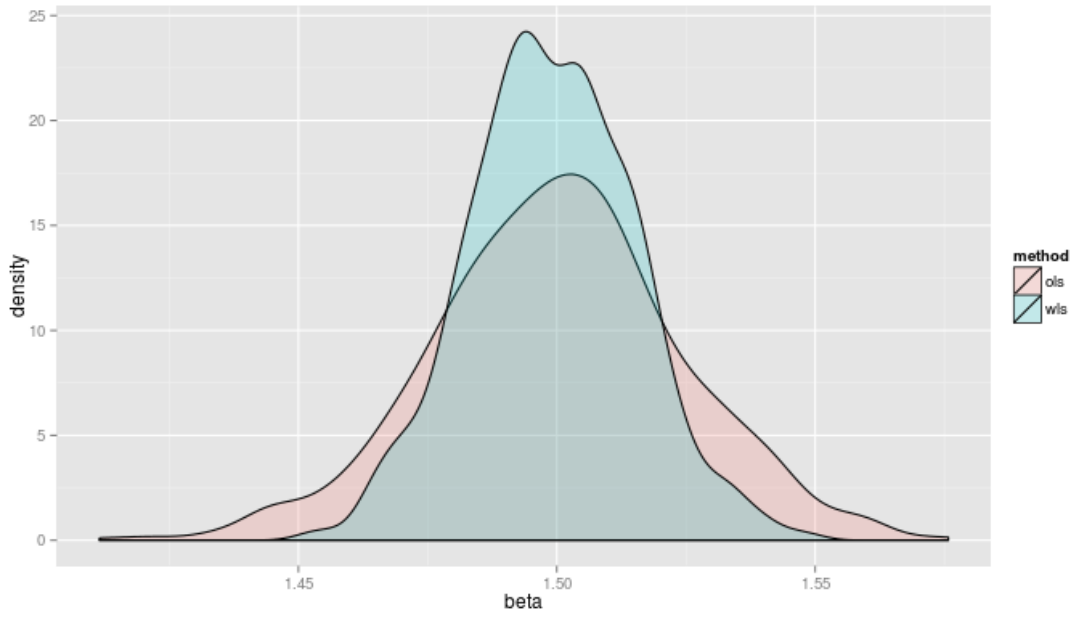

Figure 1: Relative efficiency of WLS

A review of loops, a loopy review

There were many questions about R code on the midterm, and especially code that will traverse a sequence of indices. There are many ways to loop over a sequence and iterate on a function — and we do this a lot in ARE212 to examine the behavior of different estimators. I used `sapply()` to iterate over a sequence. There are other ways to do this. Let's consider an equivalent for-loop.

```
res <- rep(NA, B)
for (i in seq(B)) {
  res[i] <- rnd.wls.beta(i)
}
print(head(res))
```

```
[1] 1.521475 1.485235 1.521788 1.528059 1.473549 1.484154
```

This is fine. It works. You have to pay attention, however, to the expressed indices instead of using them for implicit increments; and you have to preallocate a results vector which also can get confusing over large, sprawling projects. Also, there are more lines of code (LOC) which is often used as a metric for inefficiency.

Another thing that might help avoid loops is something called *vectorized operations*. Instead of looping through indices, is there a way to formulate the problem in terms of element-wise operations? Suppose, for example, that I want to generate a binary vector that indicates whether the value in one variable exceeds the value in another.

```
x <- rnorm(10); y <- rnorm(10)
(D <- ifelse(x > y, 1, 0))
```

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

The function `ifelse()` operates on each individual element of the equal-length `x` and `y` vectors. No looping necessary. This comes up a lot; and you should definitely make use of (and document) this behavior when available. This doesn't always work, however, especially when we are trying to generate a random vector. Suppose we want to create a vector where the elements are pulled from different distributions, depending on the value of `D`.

```
factor(x <- ifelse(D == 1, runif(1), rnorm(1)))
```

```
 [1] -2.02376587549482 0.285012832842767 -2.02376587549482 -2.02376587549482
 [5] 0.285012832842767 0.285012832842767 0.285012832842767 0.285012832842767
 [9] 0.285012832842767 0.285012832842767
Levels: -2.02376587549482 0.285012832842767
```

What happened? There are only two values in this vector. The random value was generated before the vectorized boolean check. This is equivalent, then, to the code by which we generated `D` in the first place. To fix this problem, we may have to apply the function to each value within the reference vector:

```
randme <- function(d) {
  if (d == 1) {
    r <- runif(1)
  } else {
    r <- rnorm(1)
  }
  return(r)
}

factor(x <- sapply(D, randme))
```

```
 [1] -0.65209157185594  0.154534797882661  0.323631716951203  -0.281180265781314
 [5] 0.254660100210458  0.0384278814308345 0.579372271196917  0.513248251518235
 [9] 0.302135252160951  0.867873660288751
10 Levels: -0.65209157185594 -0.281180265781314 ... 0.867873660288751
```