Because we didn't get to it last time, we'll begin by demonstrating an asymptotics result using simulation. Then, we'll briefly go through calculating White's estimator of the variance covariance matrix of coefficients that is robust to arbitrary forms of heteroskedasticity.

Asymptopia is the best -topia

In this part we'll build up a simulation in `R` to show why living in asymptopia is great. We'll verify in code what was shown in section 4.2 in lecture. We're going to demonstrate that, given linearity, population orthogonality, and our asymptotic full rank condition, no matter what distribution on the disturbances, the estimated $\boldsymbol{b}$ will be distributed normal for large N.

Let's get started. First, we'll define our true population of interest, the size of sample for each draw, and the number of draws.

```r
set.seed(3172015)
popN <- 100000
n <- 1000
draws <- 5000
popX <- cbind(1, runif(popN, 0, 20))
```

Next we want to define some errors. To make it interesting, we'll do this whole exercise with four different sets of errors. The sets will be distributed normal, uniform, Poisson, and finally a crazy bimodal gamma distribution that's made up. This is going to be awesome.

```r
bimodalDistFunc <- function (n, weight) {
  # This part has mean 4
  d0 <- rgamma(n, shape = 1) + 3
  # This part has mean 1
  d1 <- rgamma(n, shape = 1)
  flag <- rbinom(n, size = 1, prob = weight)
  d <- d1 * (1 - flag) + d0 * flag
  # Subtract the mean so this has mean zero
  d <- d  - weight*4 - (1 - weight)*1
  d
}

popEps0 <- rnorm(popN)
popEps1 <- runif(popN, -5, 5)
popEps2 <- rpois(popN, 1) - 1
popEps3 <- bimodalDistFunc(n = popN, weight = 0.7)
```

It's important to note that I rigged these epsilons so that they all have mean zero:

```r
c(mean(popEps0), mean(popEps1), mean(popEps2), mean(popEps3))

## [1] -5.639668e-05 -1.663842e-03  1.040000e-03 -2.722085e-03
```

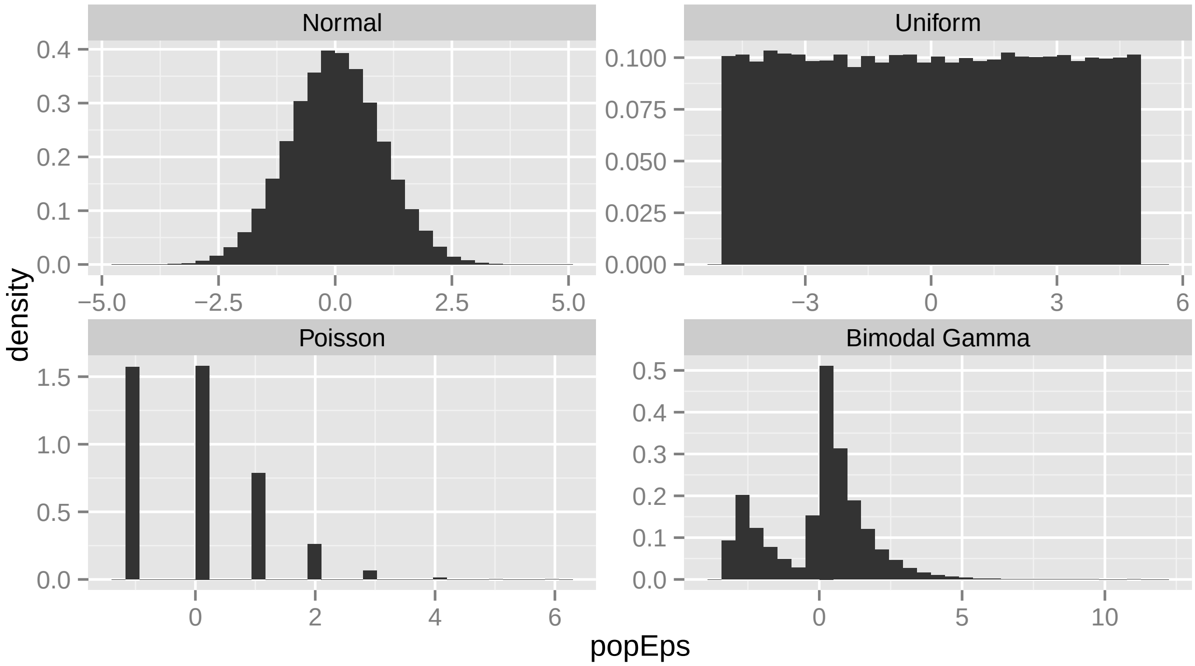
This isn't strictly necessary, but it makes our lives easier down the road. Now we'll create a data frame with our errors that we can easily graph using `ggplot2`. This takes a little doing — we need to stack the errors on top of each other and add a factor variable for the type of error they are.

Recall that factor variables are just categorical variables in R.

```
popEps <- c(popEps0, popEps1, popEps2, popEps3)
popEpsType <- c(rep("Normal",popN), rep("Uniform", popN),
  rep("Poisson", popN), rep("Bimodal Gamma", popN))
class(popEpsType)

## [1] "character"

popEpsType <- factor(popEpsType, levels = c("Normal", "Uniform",
  "Poisson", "Bimodal Gamma"))
levels(popEpsType)

## [1] "Normal"        "Uniform"       "Poisson"       "Bimodal Gamma"

class(popEpsType)

## [1] "factor"

popEpsDf <- data.frame(popEps, popEpsType)
```

Now that we've set all of this up, why not see what these crazy errors look like?

```
library(ggplot2)
g <- ggplot(data = popEpsDf, aes(x = popEps)) +
  geom_histogram(aes(y=..density..)) +
  facet_wrap( ~ popEpsType, ncol=2, scales = "free")
g
```



Cool. The last three all violate our normality assumption pretty egregiously. Let's see how they do... in **asymptopia**!

Now we'll define our population outcome variables, one for each set of errors:

```
beta <- c(4, 2)
# All four stacked again
popYDf <- data.frame(type= popEpsType, y =rep(popX %*% beta, 4) + popEpsDf$popEps)
```

We also need to build a function that returns OLS estimates for a given set of population variables. `getb` will be that function, and the code it in should be familiar from previous sections. We'll randomly sample indices (without replacement) and then pull the corresponding data from `popX` and `popY`.

```
getb <- function(popY) {
  indices <- sample(1:popN, n)
  y <- popY[indices]
  X <- popX[indices, ]
  b <- solve(t(X) %*% X) %*% t(X) %*% y
  return(b)
}
```

Note that the above function depends on both the variable that is passed to it and variables that are defined in the global environment. Functions can access both, and if there are conflicts, the function will choose the one that is passed to the function.

```
library(foreach)
bListDf <- data.frame(foreach(type = unique(popEpsType), .combine = rbind) %do% {
  popY <- popYDf[popYDf$type == type,]
  foreach(draw = rep(NA, draws), .combine = rbind) %do% {
    b <- getb(popY$y)
    data.frame(alpha = b[1], beta1 = b[2], type = type)
  }
})
```

The above stacks the results in a really tall `data.frame`. This is good practice as it makes using `facets` in `ggplot2` a lot easier.

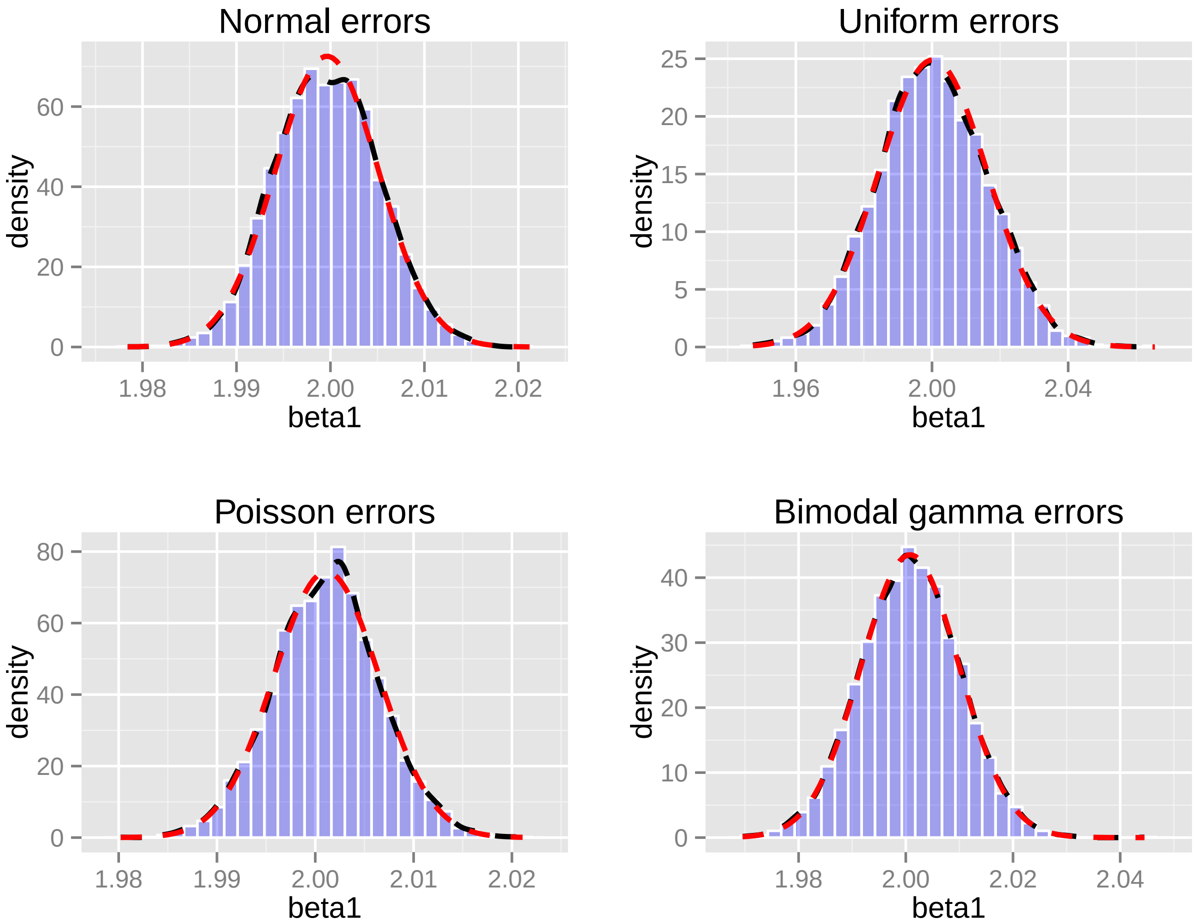
From now on we'll focus on β_1. We've generated a huge amount of these point estimates, but now we'd like to see if they're truly distributed normally. To do this, we'll create a function that graphs each of them.

```
makePlots <- function(blist) {
  estmean <- mean(blist$beta1)
  estsd <- sd(blist$beta1)
  g <- ggplot(data = blist, aes(x = beta1)) +
   geom_histogram(aes(y=..density..), fill="blue", colour="white", alpha=0.3) +
    geom_line(colour = "black", size = 1, linetype = "dashed", stat="density") +
   stat_function(geom="line", fun=dnorm, arg=list(mean = estmean, sd = estsd),
     colour = "red", size = 1, linetype = "dashed")
  return(g)
}

g0 <- makePlots(bListDf[bListDf$type == "Normal",]) + ggtitle("Normal errors")
```

```
g1 <- makePlots(bListDf[bListDf$type == "Uniform",]) + ggtitle("Uniform errors")
g2 <- makePlots(bListDf[bListDf$type == "Poisson",]) + ggtitle("Poisson errors")
g3 <- makePlots(bListDf[bListDf$type == "Bimodal Gamma",]) + ggtitle("Bimodal gamma errors")
```

Our function `makePlots` returns a histogram, smoothed density, and a normal curve with the same mean and standard deviation as the data, so we just run it 4 times to get our 4 graphs (one for each list of our estimated b values). Next, we'll use the `gridExtra` package (which you may need to install) to display the graphs together, just as we did with our errors. Note that we could also use `facets` here to grid up the plots, but using `gridExtra` can be more flexible. We need it here as the `stat_function` part can't be used without a lot of pain.

```
library(gridExtra)
grid.arrange(g0, g1, g2, g3, ncol = 2)
```



Hey, these actually look pretty good! But this isn't a rigorous test. There are actually statistical methods available to test whether a distribution is normal or not. We'll use one called the Shapiro-Wilk's test of normality. Unfortunately, the computation of Shapiro-Wilk's is a bit complicated, so we'll be leaning on a canned command here to perform it. There are other, simpler, tests (like Kolmogorov-Smirnov), but I like Shapiro-Wilk's because it's meant specifically for normal distributions.

First, we'll verify that the test works by testing it against a normal distribution and a non-normal distribution. The command `shapiro.test()` returns a number of objects, but we'll focus on the p-value.

```
cbind(shapiro.test(rnorm(draws))[2], shapiro.test(rpois(draws,10))[2])
```

```
##          [,1]      [,2]
## p.value 0.6331787 1.353845e-22
```

Great! The null is that the distribution is distributed normal, so we fail to reject (at any reasonable level) the null for the normal distribution and reject the null with a great deal of confidence for the Poisson distribution. Now let's try it on our estimated coefficients:

```
cbind(
  shapiro.test(bListDf[bListDf$type == "Normal","beta1"])[2],
  shapiro.test(bListDf[bListDf$type == "Uniform","beta1"])[2],
  shapiro.test(bListDf[bListDf$type == "Poisson","beta1"])[2],
  shapiro.test(bListDf[bListDf$type == "Bimodal Gamma","beta1"])[2])
```

```
##          [,1]      [,2]      [,3]      [,4]
## p.value 0.3929794 0.2436704 0.6138136 0.7504847
```

Great! We fail to reject the null for all four distributions, so we can feel some confidence that our coefficients are truly distributed normal! All is well in asymptopia!

White SE's

There's a bunch of great stuff we can do in Asymptopia. One of these things is to easily calculate standard errors that are robust to heteroskedasticity, using the Eicker-Huber-White formula.[1] We get White SE's using the process of calculating meat and bread, then making a sandwich.[2] Let's first make some data and run OLS.

```
X <- cbind(1, rnorm(1000))

# Generate errors that scale with the size of X
e <- rnorm(1000)*X[,2]^2

y <- X %*% matrix(c(1, 2), nrow=2) + e
XXInv <- solve(t(X) %*% X)
b <- XXInv %*% t(X) %*% y
```

Now, let's go ahead and get the meat and bread. The meat corresponds to $N\boldsymbol{\Upsilon}$ in Max's notes, and the bread corresponds to the $1/N * \boldsymbol{Q}^{-1}$. The variance covariance matrix of the coefficients is the `vcov` variable, and we extract the standard errors by taking the square root of the diagonal. Then, if we want, we can apply a degrees of freedom correction like STATA does. Easy!

[1] For a quick discussion of some methods for calculating finite sample robust standard errors, see `http://economics.mit.edu/files/7422`.

[2] While the meat, bread, and sandwich language is pretty commonly used in this context, I've just now realized that you don't using end up eating the whole sandwich. The part of the resultant matrix we usually care about is just the diagonal, which, if we were to continue with the sandwich analogy, would be a very strange way to eat food.

```
library(Matrix)
meat <- t(X) %*% Diagonal(length(y), (y - X %*% b)^2) %*% X
bread <- XXInv
vcov <- t(bread) %*% (meat) %*% bread
se <- sqrt(diag(vcov))
se

## [1] 0.05707992 0.14911716

# This is what STATA returns
seDfCorr <- se * sqrt(NROW(X) / (NROW(X) - NCOL(X)))
seDfCorr

## [1] 0.05713709 0.14926650
```

In R, it's a bit trickier to get robust standard errors than in STATA, but not too much. Basically, there is a package which calculates the `vcov` matrix for you.

```
library(sandwich)
model <- lm(y ~ X - 1)
vcovCanned <- vcovHC(model, type = "HC0")
seCanned <- sqrt(diag(vcovCanned))
all.equal(se, seCanned, check.names = FALSE)

## [1] TRUE

# STATA's ,robust
vcovCannedDfCorr <- vcovHC(model, type = "HC1")
seCannedDfCorr <- sqrt(diag(vcovCannedDfCorr))
all.equal(seDfCorr, seCannedDfCorr, check.names = FALSE)

## [1] TRUE

# This gets a summary table like the summary() function
library(lmtest)
coeftest(model, vcovCanned)

##
## t test of coefficients:
##
##    Estimate Std. Error t value  Pr(>|t|)
## X1  0.94763    0.05708  16.602 < 2.2e-16 ***
## X2  2.07643    0.14912  13.925 < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```