General comments on PS1

Casey Crisman-Cox

Spring 2025

1 Theory points

1.1 Exogeneity

A surprising number of people thought that x^1 was correlated with ε. If that was the case, then none of our estimators would be any good without an instrument. Let's check this out.

$$\begin{aligned}
\mathrm{E}[\varepsilon_{it}|\mathbf{X}_i] &= \mathrm{E}\left[\sum_{s=0}^{t}(\varepsilon_{i0} + u_{it-s})(0.8)^s \mid \mathbf{X}_i\right] \\
&= \sum_{s=0}^{t} \mathrm{E}\left[\varepsilon_{i0} + u_{it-s}(0.8)^s|\mathbf{X}_i\right] \\
&= \sum_{s=0}^{t} \mathrm{E}\left[u_{it-s}|\mathbf{X}_i\right](0.8)^s \\
&= \sum_{s=0}^{t} 0(0.8)^s \\
&= 0
\end{aligned}$$

Intuitively, just think about how they're generated.

$$\begin{aligned}
x^1_{it} &= 15 - 3a_i - 2z_i^2 + 0.25x^1_{it-1} + x^{1*}_{it} \\
\varepsilon_{it} &= 0.8\varepsilon_{it-1} + u_{it}
\end{aligned}$$

Are there any x^1 terms in ε or vice versa?

I think some of you got confused about this idea that when we have lagged **dependent** variables that lead to a strict exogeneity violation

$$y_t = \rho y_{t-1} + u_t$$

This is not strictly exogenous because

$$\begin{aligned}
\mathrm{E}[u_t|y] &= \mathrm{E}[y_t - \rho y_{t-1}|y] \\
\mathrm{E}[u_t|y] &= y_t - \rho y_{t-1} = u_t \neq 0
\end{aligned}$$

It is weakly exogenous however because

$$\begin{aligned}
\mathrm{E}[u_t|y_t] &= \mathrm{E}[y_t - \rho y_{t-1}|y_{t-1}] \\
&= \mathrm{E}[y_t|y_{t-1}] - \rho y_{t-1} \\
&= \rho y_{t-1} - \rho y_{t-1} \\
&= 0
\end{aligned}$$

In our example however, many of you tried to reason that there was an issue because both ε and x^1 were dynamic, but that doesn't hold up here because y_t contains the complete history u_t, while x^1 doesn't have any ε term entering it.

1.2 Variance of a sum of random variables

Many of you tried to find the variance of x_{it}^1 by exploiting stationarity as in

$$\begin{aligned}
\mathrm{Var}(x_{it}^1) &= \mathrm{Var}(15 - 3a_i - 2z_i^2 + 0.25x_{it-1}^1 + x_{it}^{1*}) \\
&= 9\,\mathrm{Var}(a_i) + 4\,\mathrm{Var}(z_i^2) + 0.5\,\mathrm{Var}(x_{it}^1) + \mathrm{Var}(x_{it}^{1*}) \quad \text{INCOMPLETE}
\end{aligned}$$

and stopping here as if you didn't just find x_{it-1} covaries with α_i and z_i^2

$$\begin{aligned}
\mathrm{Var}(x_{it}^1) &= \mathrm{Var}(15 - 3a_i - 2z_i^2 + 0.25x_{it-1}^1 + x_{it}^{1*}) \\
&= 9\,\mathrm{Var}(a_i) + 4\,\mathrm{Var}(z_i^2) + \frac{1}{16}\mathrm{Var}(x_{it}^1) + \mathrm{Var}(x_{it}^{1*}) \\
&\quad - 3(0.25)2\,\mathrm{Cov}(x_{it}, \alpha_i) - 2(0.25)2\,\mathrm{Cov}(x_{it}, z_i^2) \\
&= 3 + 1 + \frac{\mathrm{Var}(x_{it}^1)}{16} + \frac{1}{4} + 2 + \frac{2}{3} \\
\frac{15}{16}\mathrm{Var}(x_{it}^1) &= \frac{83}{12} \\
\mathrm{Var}(x_{it}^1) &\approx 7.38
\end{aligned}$$

1.3 Size and power of hypothesis tests

Many of you remembered this but to be clear for a test with H_0 versus H_A

1. The size of a test is $\Pr(\text{Reject}|H_0)$ (False positive). We choose this a priori by selecting

a cut point α where we reject H_0 if $\alpha < 0.05$. This should be constant across various testing conditions and alternatives

2. The power of a test is $\Pr(\text{Reject}|H_A)$ (True positive). This will vary based on the true population model and parameters.

Let's consider the simplest case where we have a unknown mean from some population of interest:

$$\begin{aligned} H_0 :& \mu = 0 \\ H_A :& \mu \neq 0. \end{aligned}$$

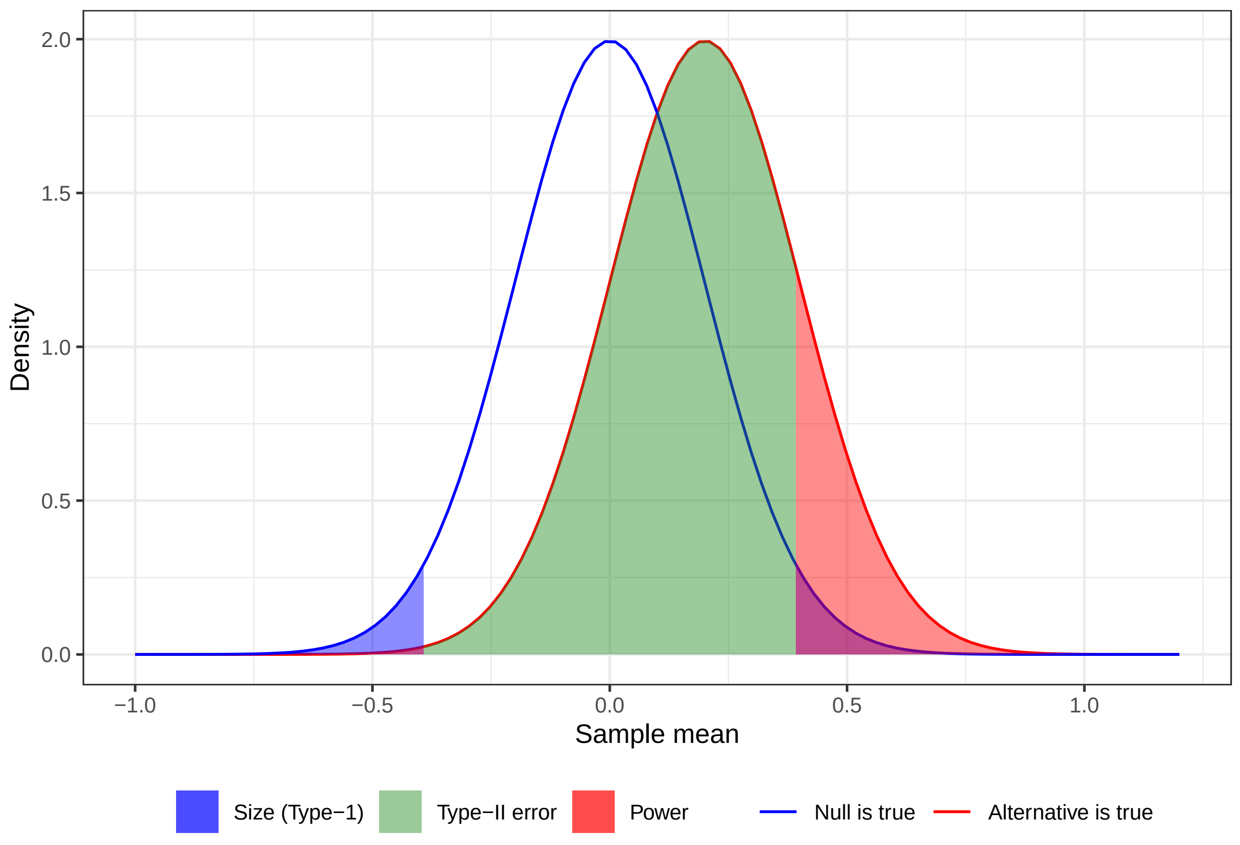
Let's suppose we have a large, iid sample of size N so that the CLT kicks in and $\hat{\mu} = \bar{x} \overset{asy}{\sim} N(\mu, \sigma^2/N)$ and if H_0 is true $Z = (\hat{\mu} - \mu)/(\sigma/\sqrt{N}) \sim N(0,1)$. We will reject if observe $|z| > 1.96$. The size of the test here is

$$\begin{aligned} \Pr(\text{Reject}|H_0) &= 1 - \Pr(Z < 1.96|\mu = 0) + \Pr(Z < -1.96|\mu = 0) \\ &= 0.05 \end{aligned}$$

The power, however, depends on the true μ and the null μ_0. Note that under the alternative the test statistic is no longer standard normal because $\text{E}[\hat{\mu}] = \mu \neq \mu_0$.

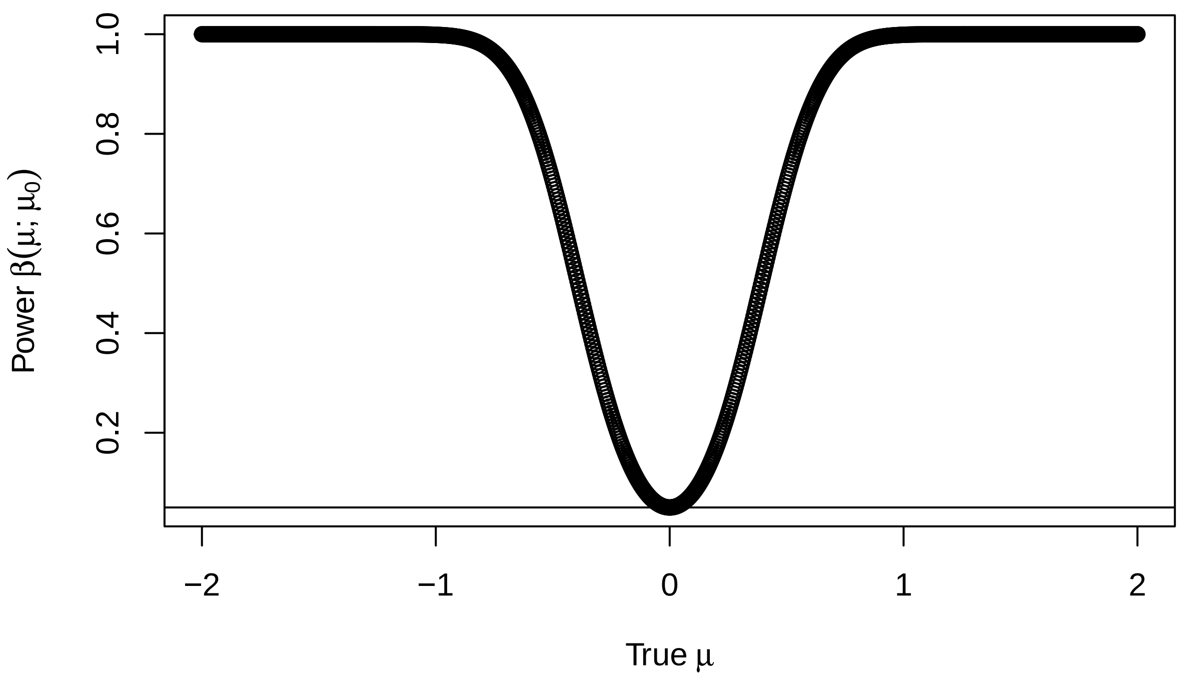
$$\begin{aligned} \beta(\mu; \mu_0) &= \Pr(\text{Reject}|H_A) \\ &= 1 - \Pr(Z < 1.96|\mu) + \Pr(Z < -1.96|\mu) \\ &= 1 - \Pr\left(\frac{\hat{\mu} - \mu_0}{\sigma/\sqrt{N}} < 1.96\right) + \Pr\left(\frac{\hat{\mu} - \mu_0}{\sigma/\sqrt{N}} < -1.96\right) \\ &= 1 - \Pr\left(\frac{\hat{\mu} - \mu_0 - \mu}{\sigma/\sqrt{N}} < 1.96 - \mu/(\sigma\sqrt{N})\right) + \Pr\left(\frac{\hat{\mu} - \mu_0 - \mu}{\sigma/\sqrt{N}} < -1.96 - \mu/(\sigma\sqrt{N})\right) \\ &= 1 - \Pr\left(\frac{\hat{\mu} - \mu}{\sigma/\sqrt{N}} < 1.96 - (\mu - \mu_0)/(\sigma\sqrt{N})\right) + \Pr\left(\frac{\hat{\mu} - \mu}{\sigma/\sqrt{N}} < -1.96 - (\mu - \mu_0)/(\sigma\sqrt{N})\right) \\ &= 1 - \Phi\left(1.96 - (\mu - \mu_0)/(\sigma\sqrt{N})\right) + \Phi\left(-1.96 - (\mu - \mu_0)/(\sigma\sqrt{N})\right) \end{aligned}$$

So if we have $\mu = .5$, a sample of size 100, and $\sigma = 2$ for example, then we can consider the following



We can also see how power changes as the true values change (given the null)

```
truth <- seq(-2, 2, length=1000)
power <- pnorm(1.96- (truth-0)/(s/sqrt(N)), lower=FALSE)+pnorm(-1.96- (truth-0)/(s/sqrt
plot(power~truth, ylab=expression("Power"~beta(mu*";"~mu[0])), xlab=expression("True"~m
abline(h=0.05)
```

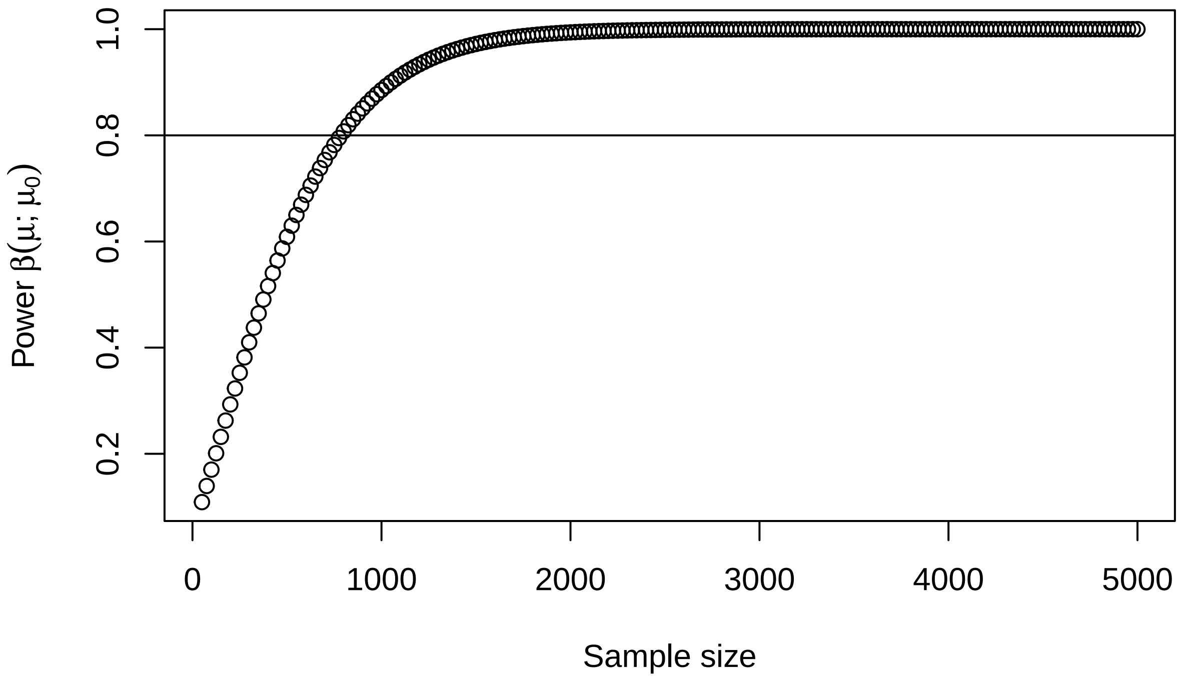


```
min(power)
```

```
## [1] 0.05000727
```

Or as a function of N given a specific μ

```
## How big of a sample do I need if
##I want to have be able to detect a true effect
## of at least .1 more than 80% of the time?
N <- seq(50,5000, by=25)
power <- pnorm(1.96- (mu-0)/(s/sqrt(N)), lower=FALSE)+pnorm(-1.96- (mu-0)/(s/sqrt(N)))
plot(power~N, ylab=expression("Power"~beta(mu*";"~mu[0])), xlab="Sample size")
abline(h=.8)
```



```
N[min(which(power-.8 >0 ))]
```

```
## [1] 800
```

1.4 Specification tests recap

There seemed to be some genuine confusion about implementing these tests. Consider the model:

$$y_{it} = \beta' x_{it} + \gamma' z_i + \alpha_i + \varepsilon_{it}.$$

We decide to fit this with the RE-GLS, within, and Mundlak estimators. These are, respectively

$$\begin{aligned}
\begin{bmatrix}\hat{\alpha}\\ \hat{\beta}\\ \hat{\gamma}\end{bmatrix}_{GLS} &= \left(\left[1-\hat{\omega};\quad X-(\hat{\omega}\cdot\bar{X});\quad Z-(\hat{\omega}\cdot\bar{Z})\right]'\left[1-\hat{\omega};\quad X-(\hat{\omega}\cdot\bar{X});\quad Z-(\hat{\omega}\cdot\bar{Z})\right]\right)^{-1}\\
&\quad\left[1-\hat{\omega};\quad X-(\hat{\omega}\cdot\bar{X});\quad Z-(\hat{\omega}\cdot\bar{Z})\right]'\left[y-(\hat{\omega}\cdot\bar{y})\right]\\
\hat{\beta}_w &= \left(\left[X-\bar{X}\right]'\left[X-\bar{X}\right]\right)^{-1}\left[X-\bar{X}\right]'\left[y-\bar{y})\right]\\
\begin{bmatrix}\hat{\alpha}\\ \hat{\beta}\\ \hat{\gamma}\\ \hat{\psi}\end{bmatrix}_{M} &= \left(\left[1;\quad X;\quad Z;\quad \bar{X}\right]'\left[1;\quad X;\quad Z;\quad \bar{X}\right]\right)^{-1}\left[1;\quad X;\quad Z;\quad \bar{X}\right]'y,
\end{aligned}$$

where $\bar{X}$ refers to a matrix of within-group means. The Hausman test compares $\hat{\beta}_w$ to $\hat{\beta}_{GLS}$. The null hypothesis is that they are equal. The test statistic is

$$H = (\hat{\beta}_w - \hat{\beta}_{GLS})'(\mathrm{Var}(\hat{\beta}_w) - \mathrm{Var}(\hat{\beta}_{GLS}))^{-1}(\hat{\beta}_w - \hat{\beta}_{GLS}) \sim \chi^2_{\dim(\beta)}.$$

The specification test we use for the Mundlak is a Wald test on the hypothesis that $\psi = 0$, here

$$A = [0_{\dim(\psi)\times(\dim(\beta)+\dim(\gamma)+1)};\ I_{\dim(\psi)}]$$

is our hypothesis matrix such that

$$A\begin{bmatrix}\hat{\alpha}\\ \hat{\beta}\\ \hat{\gamma}\\ \hat{\psi}\end{bmatrix}_{M} = \hat{\psi}$$

The Wald test is then

$$\begin{aligned}
W &= \left(A\begin{bmatrix}\hat{\alpha}\\ \hat{\beta}\\ \hat{\gamma}\\ \hat{\psi}\end{bmatrix}_{M}\right)'\left(A\,\mathrm{Var}\left(\begin{bmatrix}\hat{\alpha}\\ \hat{\beta}\\ \hat{\gamma}\\ \hat{\psi}\end{bmatrix}_{M}\right)A'\right)^{-1}\left(A\begin{bmatrix}\hat{\alpha}\\ \hat{\beta}\\ \hat{\gamma}\\ \hat{\psi}\end{bmatrix}_{M}\right)\\
W &= \hat{\psi}'\,\mathrm{Var}(\hat{\psi})^{-1}\hat{\psi} \sim \chi^2_k
\end{aligned}$$

2 Programming thoughts

First of all, thank you for introducing me to the `ave` command for group means. I love it.

Second, I think there are a few programming tips that you may want to have in your toolbox before moving forward. Throughout we'll focus on an example regarding fitting a heteroskedastic linear model with MLE. The model takes the form

$$y_i = \beta' x_i + \varepsilon_i, \quad \varepsilon_i | X \overset{iid}{\sim} N(0, \sigma_i^2), \quad i = 1, \ldots, N.$$

The likelihood function for this is based on the distributional assumption

$$y|X \sim N(X\beta, \Omega), \quad \Omega = [\sigma_1^2, \ldots, \sigma_N^2]' I_N$$

where I_N is an identity matrix of size N. Because we can't estimate N different variance parameters, we'll impose a functional form on the model, such that

$$\sigma_i^2 = \exp(\gamma' z_i),$$

where $z_i \subseteq x_i$.

The log-likelihood function becomes

$$\begin{aligned} L(\theta \mid y) &= \log \left[2\pi^{N/2} \det(\Omega)^{-1/2} \exp\left(-\frac{1}{2}(y - X\beta)'\Omega^{-1}(y - X\beta) \right) \right] \\ &= \log \left[\prod_{i=1}^{N} 2\pi^{1/2} (\sigma_i^2)^{-1} \exp\left(-\frac{1}{2}(y_i - x_i'\beta)^2 \sigma_i^{-2} \right) \right] \\ &= \sum_{i=1}^{N} -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log(\sigma_i^2) - \frac{(y_i - \beta' x_i)^2}{2\sigma_i^2} \\ &= \sum_{i=1}^{N} -\frac{1}{2}\log(2\pi) - \frac{1}{2}\gamma' z_i - \frac{(y_i - \beta' x_i)^2}{2\exp(z_i'\gamma)}. \end{aligned}$$

2.1 Vectorizing and loops

Ok now compare the following approaches to coding this

```r
ll1 <- function(theta, X, Z, y){
  beta <- theta[1:ncol(X)]
  gamma <- theta[(ncol(X)+1):length(theta)]
  ll <- 0
  for(i in 1:nrow(X)){
    ll <- ll + (-(Z[i,] %*% gamma )/2 - ((y[i] - X[i,] %*% beta)^2) /(2*exp(Z[i,] %*% ga
  }
  return(-ll)
}




ll2 <- function(theta, X, Z, y){
  beta <- theta[1:ncol(X)]
  gamma <- theta[(ncol(X)+1):length(theta)]
  ZG <- Z%*%gamma
  e <- y - X %*% beta
  ll <- -ZG/2 - (e^2)/(2*exp(ZG))

  return(-sum(ll))
}

N <- 10000
X <- cbind(1, rnorm(N), runif(N))
Z <- X
beta <- c(-1, .5, 1)
gamma <- c(1, .25, .5)
sigma.i <- exp(Z %*% gamma)
summary(sigma.i)
```

```
##        V1
##  Min.   : 1.015
##  1st Qu.: 2.871
##  Median : 3.491
##  Mean   : 3.643
##  3rd Qu.: 4.263
##  Max.   :10.285
```

```r
y <- X %*% beta + rnorm(N, mean=0, sd=sigma.i)
t1 <- system.time(replicate(50, ll1(c(beta, gamma), X, Z, y)))
t2 <- system.time(replicate(50, ll2(c(beta, gamma), X, Z, y)))
t1[3]/t2[3]
```

```
##  elapsed
## 33.71951
```

The looped version takes anywhere from 20-70 times longer to run! The difference between these functions is that the former loops over observations while the latter uses **vectorization**. This means that we avoid loops by just using functions on whole vectors or matricies whenever possible. Languages like R or python are interpreted rather than compiled. This means that they produce results in real time as you run the code as opposed to languages like C or Fortran where you compile your code into a program and then run that. One result of this is that R functions are typically compiled (i.e., link to pre-compiled C or Fortran routines) and are optimized to work on vectors as a whole.

For a single run, you may not notice much difference between looped and vectorized code all, but during optimization you'll run this function many times and small differences become big.

```r
system.time({fit1 <- optim(c(beta, gamma),
                           ll1, method="BFGS",
                           X=X, Z=Z, y=y)})
```

```
##    user  system elapsed
##   9.821   0.000   9.821
```

```r
system.time({fit2 <- optim(c(beta, gamma),
                           ll2, method="BFGS",
                           X=X, Z=Z, y=y)})
```

```
##    user  system elapsed
##   0.365   0.636   0.139
```

```r
fit1$counts
```

```
## function gradient
##       47       12
```

```
fit2$counts
```

```
## function gradient
##       47       12
```

Something else I noted with your problem sets is that many of you were not taking advantage of the fact that things like X and y and $\bar{X}$ and $\bar{y}$ are fixed. Pulling them from a data frame or calculating group means every time you want to run the function adds up too. A good rule of thumb is if you have a line of code in your likelihood function that doesn't use the parameters, then try to move it outside.

With just these two tips: vectorized code and fixing things that are fixed, I was able to make most of your go from hours to minutes. Now, this is not officially a coding class and it's not a problem for you or your career if your code is bad. Ideas and research design publish papers, not code. But, if your code is bad you will lose time to long run times. Which is not the end of the world if you're OK with letting things run overnight or for days. That's also common, but for something like this small changes go a long way.

The difference can be more pronounced when you don't provide gradients as it will run the function many more times to approximate the gradients with finite differences. We can find these first derivatives (this is where symbolic algebra packages pay off, particular in more complicated setting)

```
from sympy import *
init_printing()
x, z, y, b, g = symbols("x_i, z_i, y_i, beta_i gamma_i")
ll = (-(z*g)/2 - ((y- x * b)**2) /(2*exp(z*g)))
ll
```

```
##                                    2  -gamma_i*z_i
##   gamma_i*z_i   (-beta_i*x_i + y_i) *e
## - ----------- - ----------------------------------
##        2                       2
```

```
Dbeta = ll.diff(b)
Dgamma = collect(ll.diff(g), z)

Dbeta
```

```
##                           -gamma_i*z_i
## x_i*(-beta_i*x_i + y_i)*e
```

```
Dgamma
```

```
##     /                     2  -gamma_i*z_i    \
##     |(-beta_i*x_i + y_i) *e                 1|
## z_i*|---------------------------------- - -|
##     \                2                     2/
```

```
latex(Dbeta)
```

```
## x_{i} \left(- \beta_{i} x_{i} + y_{i}\right) e^{- \gamma_{i} z_{i}}
```

```
latex(Dgamma)
```

```
## z_{i} \left(\frac{\left(- \beta_{i} x_{i} + y_{i}\right)^{2} e^{- \gamma_{i} z
## _{i}}}{2} - \frac{1}{2}\right)
```

Which tell us that the first derivatives wrt to β and γ are

$$
\begin{aligned}
D_\beta L(\beta,\gamma \mid y, X) &= \sum_{i=1}^N \exp(-\gamma' z_i)(y_i - \beta' x_i) x_i \\
D_\gamma L(\beta,\gamma \mid y, X) &= \sum_{i=1}^N \frac{1}{2}\left(\exp(-\gamma' z_i)(y_i - \beta' x_i)^2 - 1\right) z_i
\end{aligned}
$$

```
gr <- function(theta, X, Z, y){
  beta <- theta[1:ncol(X)]
  gamma <- theta[(ncol(X)+1):length(theta)]
  ZG <- drop(Z%*%gamma)
  e <- drop(y - X %*% beta)
  Dbeta <- exp(-ZG)*e*X
  Dgamma <- ((e^2)*exp(-ZG) -1) *Z/2
  return(-colSums(cbind(Dbeta, Dgamma)))
}

library(numDeriv) ## a good way to test your gradients
x0 <- runif(6) #random values for comparison
grad(ll2, x0, X=X, Z=Z, y=y) ## first differences
```

```
## [1]   5680.022  -5380.491   2152.816 -36317.243   2110.915 -19254.320
```

```r
gr(x0, X=X, Z=Z, y=y) ## our analytic function
```

```
## [1]   5680.022  -5380.491   2152.816 -36317.243   2110.915 -19254.320
```

```r
system.time({fit1 <- optim(c(beta, gamma),
                           ll1, gr=gr,
                           method="BFGS",
                           X=X, Z=Z, y=y)})
```

```
##    user  system elapsed
##   4.865   4.151   2.694
```

```r
system.time({fit2 <- optim(c(beta, gamma),
                           ll2, gr=gr,
                           method="BFGS",
                           X=X, Z=Z, y=y)})
```

```
##    user  system elapsed
##   0.123   0.188   0.045
```

```r
fit1$counts
```

```
## function gradient
##       47       12
```

```r
fit2$counts
```

```
## function gradient
##       47       12
```

Basically, when coding you want to stop anytime you find yourself writing a loop and determine there is any better way to do it. Sometimes there is (i.e., vectorization) and this is particularly likely when you're dealing with matrices and linear algebra. Sometimes there isn't and the only way to do something is to go group by group. Likewise, other jobs (i.e., iterative procedures) just require doing the same thing over again and loops are a good tool there.

Note that various apply functions typically aren't actually better than loops, and for the most part do the loop internally. The main advantage of *ply function is readability and reduced space.

```r
x <- matrix(rnorm(5000), ncol=5)
system.time(replicate(10000, colSums(x))) ## fast; vectorized
```

```
##    user  system elapsed
##   0.122   0.003   0.125
```

```r
system.time(replicate(10000, apply(x, 2, sum)))
```

```
##    user  system elapsed
##   0.817   0.000   0.817
```

```r
system.time(
  replicate(10000, {
    out <- rep(0, 5)
    for(i in 1:ncol(x)){
      out[i] <- sum(x[,i])
    }
  })
)
```

```
##    user  system elapsed
##   0.344   0.000   0.344
```

3 Other stray thoughts

One wild thing I've noticed is that someone really beat in your head that everything should be its own function. While there's nothing inherently wrong with that, I think it created some blind spots. Many of you ended up fitting, refitting, and fitting certain models like the within estimator multiple times within a single Monte Carlo iteration. While that's a relatively cheap act and not likely to be a real issue, it can be if you were doing this with something more real.

The other part of the functions is that it does, at least for me, make it harder to think about/work through debugging because now I have to scroll between multiple function definitions and set different arguments to find where a wrong result is coming from.

Also y'all really don't like working with matricies do you? I've never seen so many data frames passed around.

3.1 When parallelizing can be helpful and easy tools for it

Sometimes when faced with an "embarassingly" parallel job like simulations or bootstraps, it can make sense to parallelize. Note we want to be careful here, as there are overhead costs to doing it and we need to be careful not over burden our machines (particularly if we want to do things like the code runs). Parallelization can sound scary, but it's gotten a lot easier over the years.

There are two main types of clusters: PSOCK and FORK. FORK works on Mac and Linux (not Windows, Microsoft security prevents this) and creates "child" R sessions that can share objects with the main session. This means that we don't actually have to copy or export your baseline objects that don't change across iterations. This can be critical if you have big data, that you don't want to copy time after time, but it has more overhead in terms of accessing the parent isn't costly.

PSOCK is fully cross platform but creates independent R sessions and exports everything to each one. If you're not dealing with huge memory-crushing things, then this copying can save time. This is the default and 99% of the time it's fine to just stick with that.

There are 3 packages that I like to use for parallelizing.

```r
library(doParallel)
library(doRNG)
library(foreach)

detectCores() ## find the number of cores you have available
```

```
## [1] 8
```

```r
K <- 4 ## You can fix the number of cores you want to use

## Or use a function of that if you want to use it across different machines
## without always changing the number
K <- floor(detectCores()*.75)

## build a cluster (PSOCK)
cl <- makeCluster(K)

registerDoParallel(cl)
set.seed(1)
```

```r
N <- 2500
X <- cbind(1, rnorm(N), runif(N))
Z <- X
beta <- c(-1, .5, 1)
gamma <- c(1, .25, .5)
sigma.i <- exp(Z %*% gamma)

system.time({
## see ?foreach for more options
out <- foreach(b =1:500,  ##number of iterations like a for loop but = instead of in
               .combine = "rbind",  ## how to combine the output (default list)
               .packages = "MASS", ## any packages you want to export (default none)
               .inorder=FALSE ## Does the order of the iterations matter?
)%dorng%{
  ## after the end of the foreach call but before the {
  #### we include one of the following:
  #### %do%  runs sequentially (not parallel)
  #### %dopar%  runs parallel but ignores outside seed (reproducability becomes tricky
  #### %dorng%  runs parallel using seed (fully reproducable out of the bag)
  y <- X %*% beta + rnorm(N, mean=0, sd=sigma.i)
  t1 <- system.time({fit1 <- optim(c(beta, gamma),
                                   ll1, gr=gr,
                                   method="BFGS",
                                   X=X, Z=Z, y=y)})
  t2 <- system.time({fit2 <- optim(c(beta, gamma),
                                   ll2, gr=gr,
                                   method="BFGS",
                                   X=X, Z=Z, y=y)})
  c(t1["elapsed"], t2["elapsed"])
}
})
```

```
##    user  system elapsed
##   0.339   0.044  79.448
```

```r
stopCluster(cl)
summary(out)
```

```
##     elapsed           elapsed
##  Min.   :0.5090   Min.   :0.00500
##  1st Qu.:0.6665   1st Qu.:0.00600
##  Median :0.9260   Median :0.00800
##  Mean   :0.8740   Mean   :0.00783
##  3rd Qu.:1.0475   3rd Qu.:0.00800
##  Max.   :1.2760   Max.   :0.03000
```

Note that stopping the cluster is important otherwise those processes stay open and running as zombies in the background with whatever memory you exported to them.