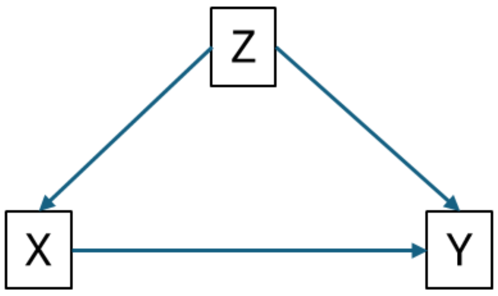
Demonstration 1: Confounding.

We aim, here, to demonstrate the effect that an unobserved confounder might have on regression estimates. With regression we might be interested in the effect of X on Y. But there might be an unobserved variable, Z, which affects both X and Y. This can be visualised with this path diagram:



For instance, if X is education and Y is earnings, Z might be a measure of intelligence – it might be that smarter people are more likely to do well in school, and independent of that be more likely to earn more. In that case, a correlation would be generated between education and earnings, even if additional schooling had no impact on earnings at all.

We start by clearing our data (to make sure nothing is hidden that we don't know about), and then setting the size of the dataset – we'll do 1000 observations for now

```
clear
```

```
set obs 1000
```

In order to simulate data which matches this, we need to start by generating the variable(s) that (in this simplified version of the world at least) isn't caused by anything else – that is, Z

For now we will do this with Z being drawn from a random normal distribution, with a mean of zero and a variance of 2. You might want to adapt this depending on what you imagine the variable being. For instance, if you wanted to generate a variable with a uniform distribution, you would use `runiform()`

```
gen Z = rnormal(0,2)
```

Now we have Z, we are able to generate the other variables. We can generate X as a function of Z

```
gen X = 0.5*Z + rnormal(0,2)
```

And Y as a function of both X and Z

```
gen Y = 1 + 2*X + 2*Z + rnormal(0,4)
```

Again, we might choose to vary the parameters we are using here, in order to mimic a particular real-life scenario.

Now that we have all our variables have been created, we can run our model. Remember, we are envisioning that Z is unmeasured (even though here we have measured it!) so we wouldn't be able to include that in our model

```
regress Y X
```

We know that the true value here should be 2 (as that is what we put in our code when defining Y). In this case, we would expect there to be bias in the estimate of the effect of X because of the confounding that we have simulated in our data.

To fix this, we could control for Z (of course, this might not be possible if Z in unmeasured, but in simulation world, we can still do this!)

```
regress Y X Z
```

We would expect this model to be unbiased in the estimate of Z – that is it should be approximately 2 – although because we are simulating our variables from random distributions, it won't be exactly 2.

Check: do your results match your expectations?

It is likely that we might want to test different levels of confounding – that is, what happens when there is no confounding, a small amount of confounding, lots of confounding, etc. We could do this by repeating the above code, but changing the size of the effect of Z on Y: e.g. if we replaced Y as followed:

```
gen Y = 1 + 2*X + 0*Z + rnormal(0,4)
```

We would expect our estimate of the effect of X to be relatively unbiased – that is fairly close to 2. However if we generated Y with a really large Z effect

```
gen Y = 1 + 2*X + 4*Z + rnormal(0,4)
```

We'd expect significant bias in our estimates of the effect of Z.

It's a bit time consuming to do loads of different repetitions (particularly if we are testing lots of combinations of estimates). So we might utilise loops to automate the running of multiple models for us. In this case, that might look like this:

```
forvalues B2 = 0/3 {
clear
set obs 1000
gen e = rnormal(0,4)
gen Z = rnormal(0,2)
gen X = 0.5*Z + rnormal(0,2)
gen Y = 1 + 2*X + `B2'*Z + e
regress Y X
regress Y X Z
}
```

So, what is happening here? The first line introduces a “forvalues” loop – it says that for values of B2 between 0 and 3 (that is: 0, 1, 2, and 3) do the things that are within the {} parentheses. What is within those parentheses is the exact code that we went through above. The only difference is that in defining Y, the coefficient associated with Z is replaced with `` `B2′ ``. The contrasting quote marks `’ tell Stata that this refers to the B2 specified in the first line. It effectively takes the value of B2 for that round of the loop. As such, this code will repeat the simulation 4 times, for each of the four specified values of B2.

Have a look at the results from these four regressions.

Do the results match what you would have expected? That is, does there appear to be more bias for bigger values of B2?

There estimate when B2 = 0 is not exactly 2 – that is it isn’t getting the truth perfectly. Can you explain why?

Exercise 1: Colliders

We now want you to repeat this exercise, but with a different DGP. That is instead of an unmeasured confounder, we now want to introduce a collider. A collider is a variable that is caused by two other variables. In the below case, Z is caused by both X and Y.

$$\begin{tikzcd}
 & \boxed{Z} & \\
\boxed{X} \arrow[ur] \arrow[rr] & & \boxed{Y} \arrow[ul]
\end{tikzcd}$$

For instance, if X is education and Y is earnings, Z might be a measure of occupational prestige.

The process for this is the same as before:

- Start with the variable that doesn't have any arrows going into it, and create a simulated version of that
- Then generate those that are only produced by that variable, and so on until all the variables are generated
- You can then attempt to run the regression model of interest, to see how including Z in your model (or not) affect the estimation of X.

Demonstration 2: a full simulation study using confounding

In this session, we're going to extend the code we used before, and extend it to create a full simulation study. That is, rather than just running a single simulation, we're going to do multiple simulations (in this case, 100), collate the results, and then find things out about those simulations (bias, RMSE, optimism, etc).

First, we're going to set the seed. The problem with simulation studies is that it's surprisingly difficult for computers to generate genuinely random data. The data that is generated is made to appear random, and that's mostly fine for our purposes. But in order to do this, it takes an initial value (or seed) and does some complex maths to that as the basis of the subsequent variable generation. This is actually helpful for us – it means if we run the analysis with the same seed twice, we'll get the same answer, meaning our results can be replicated.

We can set the seed to anything that we want – here I arbitrarily choose 12345

```
clear
```

```
set seed 12345
```

Next, we're going to create a matrix where we are going to store our results. By default, Stata can only store one model at once. But because we are going to need to clear our dataset each time we generate our data, we can't store our model results there. So we're going to store them in a matrix that will remain even when we clear the data.

Here, we want to store the model estimates and their associated SEs. In the model we're interested in (`regress Y X`) there will be a total of 4 things that we will be estimating – the intercept, the slope, and the standard errors of each of those. We will also want to keep a record of the different values of B2 that we are predicting for for different simulations. So we're going to create a matrix (called "A") with five columns to store each of those 5 things, and we're going to give them column names that match those.

```
mat A = (1,2,3,4,5)
```

```
mat colnames A = B2true B1 B0 B1se B0se
```

We're then ready to run some code that runs our models as before. This is exactly the same as the code we ran in the previous session, but we've now added an additional loop within the loop (in red) – that is, for each of the four values of B2, we're going to generate the data and run the model 100 times.

```
forvalues B2 = 0/3 {
forvalues iteration = 1/100 {
clear
set obs 1000
gen e = rnormal(0,4)
gen Z = rnormal(0,2)
gen X = 0.5*Z + rnormal(0,2)
gen Y = 1 + 2*X + `B2'*Z + e
regress Y X
}
}
```

All good – but we forgot to store the estimates! So we need to add a bit extra code in (in red below)

```
forvalues B2 = 0/3 {
forvalues iteration = 1/100 {
clear
set obs 1000
gen e = rnormal(0,4)
gen Z = rnormal(0,2)
gen X = 0.5*Z + rnormal(0,2)
gen Y = 1 + 2*X + `B2'*Z + e
regress Y X

mat b = e(b)
mat vars = vecdiag(e(V))

mat info = `B2'
mat A = ( A \ info, b, vars )

}
}
```

These additional 4 lines of code do the following, respectively

- Extract the beta estimates from the model
- Extract the uncertainty measures – stata stores these as variances, rather than standard errors – so we'll need to square root them later to convert them to standard errors
- Extract the true value of beta 2 from the simulation code
- Combine them all together in a new row of the matrix A

So this code will run 400 models, 100 for each of the four values of B2. For each, it will generate new data, run the model, extract the results from that model, and store them as a line in the matrix A. And it does it all (on my computer at least) in under 3 seconds!

We could now look at the matrix A

```
mat list A
```

You will see that the first row of this matrix are the numbers that we initially put in to create the matrix. These aren' t meaningful so can be removed

```
mat A = A[2...,.]
```

Next, we can clear our dataset, and turn the matrix into the data so we can use it. We can do this with the svmat function:

```
clear
```

```
svmat A, names(col)
```

Finally – remember that our uncertainty measures are currently the squares of the standard errors. To convert back to standard errors, we can square-root them.

```
replace B0se = sqrt(B0se)
```

```
replace B1se = sqrt(B1se)
```

Take a look at the dataset that has been created. Is it the same length as you expect (it should be 400)? Are there the right number of columns (should be 5)?

We can now calculate our measures of interest about these variables. For this, we want to calculate a value for each of the true values of B2. So we could use this code to calculate bias:

```
egen meanB1 =  mean(B1), by(B2true)
gen bias = meanB1 / 2
```

That is, we are taking the average estimate of B1 (for each value of B2) and then dividing it by the true value of B2 (2). In other words, we are calculating the ratio between the average estimate and the truth.

Next we could do the same with Root Mean Square Error (RMSE)

```
gen sqerror = (B1 - 2)^2
egen mse = mean(sqerror), by(B2true)
gen rmse = sqrt(mse)
```

Here, we are calculating the deviation of each estimate of B1 from the truth of 2, squaring it (so they are all positive), again taking the mean for each value of B2, and finally square rooting that number to get back to the coefficients' original scale.

Lastly for this exercise, we calculate optimism. This is effectively a measure of bias for the SE rather than the beta estimate. However, the code is a little bit more involved, since the "true" value of the SE needs to be derived from the estimates, as well as the estimated values.

To calculate the true values, we want to find the mean deviation of the B1 estimate from the mean B1 estimate. To do this we calculate that deviation, square it to make it positive, take the mean for each value of B2, and then square root back to the original scale:

```
gen sqdiff = (B1-meanB1)^2
egen meansqdiff = mean(sqdiff), by(B2true)
gen numerator = sqrt(meansqdiff)
```

To calcuate the estimated SEs, we do something similar: to get the scale right, we need to square, mean and then square root back again, to match the process for calculating the true SE

```
gen SEsq = B1se^2
egen meanSEsq = mean(SEsq), by(B2true)
gen denominator = sqrt(meanSEsq)
```

We can then calculate the optimism as the ratio between these two values.

```
gen optimism = numerator / denominator
```

The last manipulation to our data involves collapsing our data – we no longer need all 400 simulations, and just storing our estimates for bias, RMSE, and optimism

```
collapse (mean) bias rmse optimism, by(B2true)
```

So that's an awful lot of work to get just a few numbers!

We can now think about how to present our results. It might be that we just want to present our results in a table. But we might also want to plot them. Some examples are below:

```
graph bar bias, over(B2true)
```

```
twoway line bias B2true
```

```
twoway line optimism B2true
```

Have a think about what you want graphs to look like, and create a graph that fits the brief.

Exercise 2: a full simulation study of collider bias

Your task, now, is to develop a full simulation study for the collider bias example. That is, what does *controlling for Z* do, with different levels of collider bias. The process will be the same as that undertaken above for the confounding example.

- Take the code you developed this morning with collider bias, with a model that includes both X and Z are independent variables and Y as a dependent variable
- Add an additional forvalues loop to repeat it 100 times (it might be wise to start with just 10 iterations to ensure it works)
- Create a matrix at the beginning of this code, to store your estimates. Note that, because the model stores more things (estimates associated with the intercept, X *and Z*) the matrix will need to be bigger
- Add some code within the loop to ensure that the results are stored in the matrix
- Check that this code works and creates a matrix that looks right
- Based on the confounder example, write some code that extracts the results, summarises them into measures of bias, RMSE and optimism, and then plot those results.
- *Do the results match your expectations?*

If you have time, or want to play with this at home, you could think about the following

- What happens if you vary the sample size (ie “`set obs 1000`”)?
- Could you simulate some data that looks a bit like data you are using? What things would you like to experiment with / vary?
- What are the plots/tables that you would like to create from your simulation? How would you produce those?