Simulation Explanation

In this document we explain the main components of the simulation code. You are currently in the **Simulation Explanation** folder, in the **Simulation** folder there is an **R** script called `run_simulations.R`. This file runs simulations for cases one, two, and three, which are described in the paper. To do so, the script references "case functions," which are individual programs that generate data according to the rules for each case. For example, in case one data are generated such that x causes y, whereas in case two the direction of causation is reversed.

The document you are currently reading explains a single case function – case one – to help understand and replicate our simulations. Please keep in mind that this document is descriptive, actual code can be found in the **Simulation** folder. You will see section labels that partition each code snippet below. These labels are also included as comments in the case functions so that you can see how they are mapped onto the cases themselves.

1 - Set Parameters

The cases begin by setting the sample size and causal lag length.

```
N <- N
lag_amount <- lag_amount
```

The sample size varies between 72, 150, 250, 250, 450, and 550, and the lag amount varies between 0, 1, 2, 3, 4, and 5. These are function parameters that are held constant within one simulation run of 500 replicates, but then change to their next respective value for the next run (all combinations are crossed). Also note that the lag amount parameter will apply to the cross lag coefficients (explained below), whereas the autoregressive effects are always lag one.

We then set the autoregressive coefficients for x, y, and z,

```
autor_x <- 0.22
autor_y <- 0.22
autor_z <- 0.22
```

cross lag coefficients, and the number of time points.

```
x_cause_y <- 0.082
y_cause_x <- 0.082
z_cause <- 0.082


time <- 200
```

The coefficients for the autoregressive and cross lag effects are held, respectively, at 0.22 and 0.082. We chose these values because the average cross-effect coefficient reported by the ESM studies that we cite is 0.082 ($SD = 0.042$, number of coefficients = 23) and the average autoregressive coefficient is 0.22 ($SD = 0.15$, number of coefficients = 12).

Finally, we generate data across 200 time points and then select the last 20 for statistical modeling (shown below). The initial time points represent a burn-in period to ensure the system reaches equilibrium and that initial conditions are not influencing parameter estimates. We then use the final 20 time points because the ESM studies that we cite typically collect data for two weeks.

2 - Initialize Storage Vectors

Here, we set the number of replicates (500) and then create vectors that will store the results from each.

```
sims <- 500
p_wo_partial <- numeric(sims)
se_wo_partial <- numeric(sims)
b1_wo_partial <- numeric(sims)
p_partial <- numeric(sims)
se_partial <- numeric(sims)
b1_partial <- numeric(sims)
```

We will fully explain what these vectors store once we unpack the statistical models. For now, know that these are "blank" vectors for our results.

3 - Begin Replicates

We iterate through 500 replicates

```
for(q in 1:sims){
```

and each replicate begins by creating an empty data set that will store our generated data.

```
df <-  matrix(, ncol = 4, nrow = N*time)
```

Keep in mind that this matrix is different from our storage vectors in step 2. We will generate data, place it within `df`, and then evaluate the data with statistical models – the results of those models will eventually be placed into the storage vectors from step 2.

So, at this point we have an empty data matrix and have specified our coefficients and parameters, now we can generate data.

4 - Generate Data

We start with a count variable that helps store the data and begin our for-loop across individuals.

```
count <- 0


for(i in 1:N){
```

We then generate an unobserved heterogeneity term – one value that is sampled from a distribution with a mean of zero and a standard deviation of one.

```
y_het <- rnorm(1,0,1)
```

This is also known as the unit effect. Each person receives a different value, so `y_het` represents unobserved heterogeneity on y, or stable unit differences. HLM uses an intercept to model this effect, or b_{0i}

Then we take this individual and generate data across time according to the process specified by case one (x causes y).

```
for(j in 1:time){
count <- count + 1


 if(j == 1 | j == 2 | j == 3 | j == 4 | j == 5){
    df[count,1] <- j
    df[count,2] <- i
```

```
    df[count,3] <- rnorm(1,0,1)
    df[count,4] <- rnorm(1,0,1)

}else{

    df[count,1] <- j
    df[count,2] <- i
    df[count,3] <- autor_x * df[(count - 1), 3] + rnorm(1,0,1)
    df[count,4] <- autor_y * df[(count - 1), 4] + x_cause_y * df[count - lag_amount,3] +
                   y_het + rnorm(1,0,1)

      }
    }
```

The first column of `df` stores the time point, `j`, and the second column stores the person identifier, `i`. The third column stores x, the fourth y, and the fifth (in case three, not shown here) z. We set random initial conditions during the first five time points. Across the rest of time x is a function of its prior value (to the extent of the autoregressive term) and time specific error. y is a function of its prior value (to the extent of the autoregressive term), the effect of x on y at the lag specified by `lag_amount`. For example, when `lag_amount` is equal to zero x causes y concurrently. Finally, y is also a function of unobserved heterogeneity, `y_het` (the unit effect), and time-specific error.

The snippet just presented is the only place where the code changes across the case functions. We are demonstrating case one here, where x causes y. In the other cases, we change the structure to y causes x (case 2) and z causes x and y (case 3). In this last case, which includes z, we add an additional column to `df` to store the values of z across time.

Once this for-loop completes one run across time, the process iterates for every other individual in the sample.

5 - Tidy Data

We generated data and placed it into `df`, now we are going to tidy it before implementing statistical models. Here, we turn the matrix into a data frame and name the columns. We call x "action" and y "affect."

```
df <- data.frame(df)
names(df) <- c('day','id','action','affect')
```

Next, we select the last 20 time points of the data set and lag the affect variable.

```
df <- df %>%
  mutate(lag_affect = lag(affect)) %>%
  filter(day > 180)
```

Finally, we person mean center the independent variable.

```
person_mean_df <- df %>%
    group_by(id) %>%
    summarise(
      action_person_mean = mean(action)
    )

df <- left_join(df, person_mean_df)

df$action_person_center <- df$action - df$action_person_mean
```

Now we have a data set that is consistent with the reviewed ESM studies: a dependent variable ("affect"), a lagged dependent variable ("lag_affect"), and a person-mean centered independent variable ("action") across 20 time points with a sample size of `N`.

6 - Model

We apply two HLM models to the data. The first regresses affect (y) on action (x)

```
model_wo_partial <- lme(fixed = affect ~ action_person_center,
                        random = ~1|id,
                        data = df,
                        control = lmeControl(opt = 'optim')
  )
```

and the second does the same but also partials the prior observation of affect.

```
model_partial <- lme(fixed = affect ~ action_person_center + lag_affect,
                         random = ~1|id,
                         data = df,
                         control = lmeControl(opt = 'optim'))
```

We then save b_1 and its standard error and p value for both models. b_1 is the coefficient relating x to y (action to affect).

```
  # Without partial model results #
  # p-value for b1 #
  p_wo_partial[q] <- summary(model_wo_partial)$tTable[, "p-value"][2]
  # standard error for b1 #
  se_wo_partial[q] <- summary(model_wo_partial)$tTable[, "Std.Error"][2]
  # b1 itself #
  b1_wo_partial[q] <- summary(model_wo_partial)$tTable[, "Value"][2]

  # With partial model results #
  # p-value for b1 #
  p_partial[q] <- summary(model_partial)$tTable[, "p-value"][2]
  # standard error for b1 #
  se_partial[q] <- summary(model_partial)$tTable[, "Std.Error"][2]
  # b1 itself #
  b1_partial[q] <- summary(model_partial)$tTable[, "Value"][2]
```

Conclusion

The code sequence just presented is the function for case one. The functions for cases two and three are identical except for step four – data are generated such that y causes x (case two) or z causes x and y (case three). These functions take two paramters: sample size and lag amount. Sample size determines the number of people, and lag amount determines the lag length from the exogenous to endogenous variable in each respective case. We apply each of these functions across every combination of the two aforementioned parameters: 72, 150, 250, 350, 450, and 550 for sample size; and 0, 1, 2, 3, 4, and 5 for lag amount. Finally, note that we do not open each case function and run them individually. Instead, we have a `run_simulations.R`

script within the **Simulations** folder that calls each function and maps it over every combination of the aforementioned parameters.