Random effect basics

Scott Jackson

November 12, 2019

What's this?

This document is part of a course on mixed-effects models (MEM) for the Language Science Center (LSC) at the University of Maryland (UMD). One of the ultimate goals of this course is to equip students with enough knowledge to be able to confidently write code that will generate the kind of hierarchical data for which MEMs are well-suited. The point of this is twofold. On a practical level, being able to simulate complex data that approximates the kind of data one is analyzing in one's actual work is extremely valuable for a lot of applications. On a pedagogical level, being able to write code *from scratch* to generate MEM-appropriate data requires a solid degree of understanding of the structure of MEMs, and it enables further exploration and understanding of the implications and consequences of different analytic scenarios ad assumptions.

Following an initial tutorial on simulating and estimating simple linear models, in this document we will extend these models with simple random effect structures, simulating and estimating both *random intercepts* and *random slopes* (terminology described more below). The overarching goal of this tutorial is to get you started thinking about how

The suggested way to interact with this document is as follows:

1. Read through the PDF, and *type along* with the code provided. Going through the effort of typing forces one to pay closer attention to the code than simply running or copy-pasting the given code.
2. Try the Practice Exercises. Only peek at the solutions after you find your own solution or get really really stuck. If your solution differs from mine, that's okay! But try to understand and/or ask why.
3. Attempt the Extention Exercises. It's crucial to try things out with data that you care more about.
4. Submit feedback/questions/gripes/victories as "issues" via GitHub. I may not be able to reply, but I will read them all and try to respond either in class or outside of class.

Good luck!

Simulating and fitting random intercept models

Conceptualizing random intercepts

Recall the basic form of the linear model:

$y_i = \beta_0 + \beta_1 x_{i1} + ... + \epsilon_i$

As we discussed in the previous materials, this means we can simulate responses with only a few parameters. Minimally, we need an *intercept* (β_0 in the above equation) that represents the expected value of the response when all predictors have a value of zero, and a parameter (often called σ) that represents the standard deviation of the distribution of errors (ϵ). In most research contexts, we crucially also have one or more *slope* parameters (β_1, etc.) that represent how response values depend on predictor values.

For example, if we have just one predictor x, we can simulate from a simple model as follows:

```
n_obs <- 1000 # number of observations
beta_0 <- 1.5 # intercept parameter
beta_1 <- 3   # slope parameter
sigma <- 2    # standard deviation of normally-distributed errors

x <- rnorm(n_obs) # a random continuous predictor variable
```

```r
errors <- rnorm(n_obs, mean = 0, sd = sigma) # errors, with SD `sigma`

y <- beta_0 + beta_1 * x + errors

summary(lm(y ~ x))
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -7.7391 -1.3944  0.0185  1.3669  7.4879
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  1.52566    0.06601   23.11   <2e-16 ***
## x            3.02568    0.06577   46.00   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.087 on 998 degrees of freedom
## Multiple R-squared:  0.6795, Adjusted R-squared:  0.6792
## F-statistic:  2116 on 1 and 998 DF,  p-value: < 2.2e-16
```

When we have a large number of observations, the parameters returned by the fitted model closely match the "ground truth" values that we used to generate the data. So far so good, but this is just a very simple linear model. Now we want to accomplish the same thing with mixed-effects models (MEMs). In order to do this, we need to understand how a MEM differs from the simpler form of a linear model above.

Fortunately, the difference is conceptually pretty straightforward. The trickiness is that there are a few different ways to think about it, which is where confusion sometimes arises. I will try to present a few different ways of thinking about random effects, in the hopes that at least one will make some sense. But I do think there is some benefit in being able to think about them in a few different ways.

First, let's reconsider the β parameters in our standard linear model. Whatever value these parameters have for a given data set or fitted model, they are fixed for that model. This is where the term *fixed effects* comes from, because the value of the parameter is set (or estimated) to be a single value. However, we might not want to make that assumption, and in fact we might want a model that allows for variability in a β parameter. But why, and what does this mean, that a β parameter could vary?

It can be helpful to think about concrete examples. One of the standard examples, especially in language fields, is the assumption that people vary, along many different dimensions. Let's imagine an experiment in which we are measuring participants' reaction times to a linguistic stimulus. In our model, we will have an intercept term, which will represent something like the average "baseline" reaction time across all participants. In a lexical decision task, we might expect an average reaction time around 600 ms, for example. But we also have a reasonable expectation that people will vary in base "reaction speed," for lack of a more theoretically precise term. That is, independent of any manipulations or conditions, some people are just faster in general than other people. So instead of assuming a single fixed intercept parameter (which would represent a model in which all people are assumed to have the same "baseline" reaction time), we might want to fit a model that allows people to vary, such that different people could have different baseline reaction times. In short, some people are just faster than others, and this can be represented by a *distribution* of β_0 parameters, instead of a single "fixed" parameter for all people.

Random effects as hierarchical parameters

How do we construct this distribution of parameters? In short, we need a distribution family, and we need parameters for that distribution. Because these are parameters determining parameters – namely, parameters of a distribution of β parameters – these are sometimes called *hyperparameters* or *hierarchical* parameters.[1]

The question of what distribution family to use can be a complex one, and we will revisit it, but a typical assumption is that they come from a normal distribution. This is why these are called *random effects* in typical MEM terminology, because the distribution of parameters is assumed to come from a random (normal) distribution. But why random? Why normal?

Let's return to our concrete example. If we are talking about a reaction time experiment, then each person in the experiment is going to differ in a lot of ways from each other, from stable traits like overall processing speed, working memory, or other cognitive factors, to personality traits or external factors that might vary how seriously the person is taking the experiment, to more transient things like the time of day, how much sleep the participant had the night before, the last time they ate (and what they ate), whether they had something emotionally taxing that morning like a stressful conversation, etc. The point here is that it's not hard to imagine many things that could all be influencing a participant's reaction time in the experiment that day, even if any particular thing could have a small effect. But this is exactly the kind of situation where we would expect the Central Limit Theorum to apply, such that adding up a bunch of different little effects will create an overall distribution of "noise" or variance that follows a normal distribution.

The other point here is that despite all the millions of different factors (or maybe because of them), some of which may be fluctuating from moment to moment, individual people's data will tend to look more similar to the rest of their own data rather than the data of other individuals. In other words, if we assume that people vary, the flip side of the coin is that the observations for one person will cluster together in some way, since that person is different from others. This means some observations will have a common source of variance – the person – and this is why we would like to have models that can treat this common variance, because the models will better fit the data, which in turn will lead to more accurate parameter estimates and inferences about those estimates.

Therefore, we have a few different ways to conceptualize the need for a model that represents the intercept as a distribution over people, rather than a single fixed effect. On the one hand, we have theoretical reasons to expect that people vary in ways that are consistent with a random intercept. On the other, we have statistical reasons to try to model the common variance of data points coming from the same person, because those models may fit our data better and lead to better inferences. Fortunately, using a MEM addresses both of these motivations.

What do we need in order to simulate this hierarchical extension to our linear model? The only real difference is that instead of a single intercept parameter β_0, we need (hyper)parameters for the *distribution* of β_0. Since we are assuming a normal distribution, we therefore need two parameters, a mean and a standard deviation. The interpretation of the standard deviation should be fairly clear, as representing how much people vary. If the standard deviation of our random intercept is low, then people do not vary much. But what does the *mean* of this distribution represent?

(pause to think about the answer)

Relationship between "fixed effect" values and random effect distribution

Again, there are a few ways to think about the mean of the distribution of random intercepts. If we are talking about a distribution over people, then the mean essentially represents the expected intercept value for the "average person." The odds of having someone in your experiment that is precisely at the middle of this distribution (i.e., exactly at the mean) is low, so it's not the *person* in the middle of your data (which would be the *median* person), it's the value for a hypothetical "average person."

To further wrap your head around this distribution of β_0 values, think about how you could use the same parameter model to mimic the simple linear model. That is, what values for the mean and standard deviation

[1] This is also why MEMs are often called *hierarchical models*.

of β_0 would get you the assumptions of a simple linear model? Recall that we assume a "fixed effect" for a simple linear model, which by definition does not vary. This means that if we simply set the standard deviation of our β_0 distribution to 0, this is another way of saying that we assume that the intercept does not vary by person, and the result is that we only have a single β_0 estimate. So if the standard deviation of β_0 is zero in our simple linear model, what value does the mean take? Yep, the estimated "fixed effect" intercept value. In other words, a MEM where the standard deviation of a random effect is zero is equivalent to a simple linear model with only fixed effects. This also means that there is an equivalence such that we can interpret the mean of our random effect distribution as the "fixed effect" value of that parameter.

Why go on this long excursion about the equivalence of the marginal intercept (aka "fixed effect" intercept) and the mean of the random effect distribution? First, hopefully to help you build a bit more of a mathematical intuition about the structure of MEMs, but also to make it easier to see how we can use some handy mathematical equivalents when computing or simulating random effects.

That is, there are basically two equivalent ways to simulate or represent a random intercept. One way is to represent it as a (normal) distribution of values where the mean corresponds to the marginal, "fixed" effect value. Another way to represent a random effect is a (normal) distribution around a mean of zero, plus an overall (marginal) value. That is, we could either simulate a bunch of values that represent each person's (random) intercept and interpret the mean as the "fixed effect," or we could simulate a bunch of values that represent how each person's intercept *differs* from the fixed effect, and then just add the fixed effect. The point here is that we can use either, depending on what is more convenient or easier to conceptualize, and they are equivalent. These different ways of formulating the MEM may show up in different textbooks or software approaches, so it can be helpful to be able to think about it both ways.

Simulating a random intercept

Let's finally look at this in action. Returning to our concrete example, imagine that we are looking at a reaction-time experiment with two (categorical) conditions, and that each participant has a single observation in each condition. This is maybe not the most realistic experiment, but we will get to more complex examples shortly. We could think about concrete conditions like "unprimed" and "primed," for the sake of illustration, but for now let's just use 0 and 1 values representing the dummy-coding of our factor. Recall that for simulation, we need to convert categorical predictors into numeric values in order to calculate numeric response variables, so if we just start with 0s and 1s we can skip a step of converting a factor to numbers. Let's start with a small number of participants, say 10. The `expand.grid()` function in R gives us a simple way to create a data frame of fully-crossed combinations of factors, which is a convenient way to set up the general shape of our data.

```
subjects <- paste0("s", 1:10)
conditions <- 0:1
df <- expand.grid(subjects, conditions, stringsAsFactors = FALSE)
colnames(df) <- c("subject", "condition")
print(df)
```

```
##    subject condition
## 1       s1         0
## 2       s2         0
## 3       s3         0
## 4       s4         0
## 5       s5         0
## 6       s6         0
## 7       s7         0
## 8       s8         0
## 9       s9         0
## 10     s10         0
## 11      s1         1
## 12      s2         1
```

```
## 13      s3         1
## 14      s4         1
## 15      s5         1
## 16      s6         1
## 17      s7         1
## 18      s8         1
## 19      s9         1
## 20     s10         1
```

Let's imagine that people in the unprimed condition tend to respond around 800 milliseconds, and most people average between 700 and 900. So we can set the parameters of our random intercept thus:

```
intercept_bysubject_mean <- 800
intercept_bysubject_sd <- 50
```

Now we can make a separate data frame that represents our random intercepts pulled from this distribution (re-using some of the variables we constructed above):[2]

```
set.seed(84)
ranefs_bysubject <- data.frame(subject = subjects,
                               intercept_bysubject = rnorm(length(subjects),
                                                           intercept_bysubject_mean,
                                                           intercept_bysubject_sd),
                               stringsAsFactors = FALSE)
print(ranefs_bysubject)
```

```
##    subject intercept_bysubject
## 1       s1            835.9801
## 2       s2            857.0283
## 3       s3            762.7132
## 4       s4            742.6791
## 5       s5            812.7834
## 6       s6            831.3965
## 7       s7            856.5068
## 8       s8            705.4025
## 9       s9            733.6077
## 10     s10            730.8938
```

Once we have this table of random intercepts, we can use the `join` functions from the `dplyr` package to easily get the by-subject intercepts into our main data frame. This ensures that we have the same "random intercept" value appropriately matched to a given subject.

```
library(tidyverse)
```

```
## -- Attaching packages ----------------------- tidyverse 1.2.1 --
```

```
## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
```

```
## -- Conflicts -------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
df <- df %>% left_join(ranefs_bysubject, by = "subject")
print(df)
```

[2] I will use `set.seed()` just to make the results more stable and easier to refer back to for the sake of this tutorial.

```
##    subject condition intercept_bysubject
## 1       s1         0            835.9801
## 2       s2         0            857.0283
## 3       s3         0            762.7132
## 4       s4         0            742.6791
## 5       s5         0            812.7834
## 6       s6         0            831.3965
## 7       s7         0            856.5068
## 8       s8         0            705.4025
## 9       s9         0            733.6077
## 10     s10         0            730.8938
## 11      s1         1            835.9801
## 12      s2         1            857.0283
## 13      s3         1            762.7132
## 14      s4         1            742.6791
## 15      s5         1            812.7834
## 16      s6         1            831.3965
## 17      s7         1            856.5068
## 18      s8         1            705.4025
## 19      s9         1            733.6077
## 20     s10         1            730.8938
```

Now we can construct the rest of our model and simulate response values.[3]

```
beta_1 <- 100
sigma <- 50

df$y <- df$intercept_bysubject +   # random intercepts
        beta_1 * df$condition +    # effect of condition
        rnorm(nrow(df), 0, sigma)  # residual errors
print(df)
```

```
##    subject condition intercept_bysubject         y
## 1       s1         0            835.9801  763.2584
## 2       s2         0            857.0283  838.2776
## 3       s3         0            762.7132  788.1740
## 4       s4         0            742.6791  773.5117
## 5       s5         0            812.7834  844.7725
## 6       s6         0            831.3965  769.2045
## 7       s7         0            856.5068  891.2235
## 8       s8         0            705.4025  732.1401
## 9       s9         0            733.6077  720.2477
## 10     s10         0            730.8938  723.4624
## 11      s1         1            835.9801  897.6106
## 12      s2         1            857.0283  947.3839
## 13      s3         1            762.7132  911.7548
## 14      s4         1            742.6791  802.1148
## 15      s5         1            812.7834  954.7708
## 16      s6         1            831.3965  901.9230
## 17      s7         1            856.5068 1022.7792
```

[3]Side note: in some contexts, the term "beta"/β is reserved for *standardized* coefficients, that is, coefficients for a model in which all variables have been standardized. This means a beta coefficient has an interpretation which is read completely in terms of standard deviations. For example a beta of 2 would mean that for each standard deviation change in the predictor, there is an expected change of 2 standard deviations in the response. So if we are specifying our effects in terms of a scale like milliseconds, technically these are not beta coefficients. But we do not yet have a reason to belabor the difference because the model-fitting process is the same either way, and so I will stick with using β for coefficients throughout, whether they are standardized or not.

```
## 18      s8         1            705.4025  810.9624
## 19      s9         1            733.6077  869.3716
## 20     s10         1            730.8938  850.2638
```

Now it's time to finally fit our MEM! We will use the `lmer()` function from the `lme4` package, which is the *de facto* standard for fitting MEMs in R. For illustration, I will first fit a simple linear model, using the syntax that makes the model intercept explicit, so you can see how it relates to both the MEM syntax and results.

```
library(lme4)
```

```
## Loading required package: Matrix
```

```
##
## Attaching package: 'Matrix'
```

```
## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
```

```
lm_fit   <-   lm(y ~ 1 + condition              , data = df)
lmer_fit <- lmer(y ~ 1 + condition + (1|subject), data = df)

summary(lm_fit)
```

```
##
## Call:
## lm(formula = y ~ 1 + condition, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -94.779 -48.044  -5.099  51.330 125.886
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   784.43      19.81  39.602  < 2e-16 ***
## condition     112.47      28.01   4.015 0.000812 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 62.64 on 18 degrees of freedom
## Multiple R-squared:  0.4724, Adjusted R-squared:  0.4431
## F-statistic: 16.12 on 1 and 18 DF,  p-value: 0.0008124
```

```
summary(lmer_fit, corr = FALSE) # slightly simplified output
```

```
## Linear mixed model fit by REML ['lmerMod']
## Formula: y ~ 1 + condition + (1 | subject)
##    Data: df
##
## REML criterion at convergence: 193.5
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -1.8695 -0.4372  0.1325  0.3939  1.5063
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  subject  (Intercept) 3306.3   57.50
```

```
##  Residual              617.1   24.84
## Number of obs: 20, groups:  subject, 10
##
## Fixed effects:
##             Estimate Std. Error t value
## (Intercept)   784.43      19.81   39.60
## condition     112.47      11.11   10.12
```

The syntax for `lmer` separates the "fixed" and "random" parts of the model, where the fixed effect syntax is identical to that used in `lm()` and `glm()` functions. The random effect syntax is marked by parenthetical statements which themselves are composed of a left side and a right side, split by the "pipe" character (`|`). On the left side of the pipe is some kind of partial model formula, which is typically a subset of the fixed effects of the model. On the right side of the pipe is the "grouping" variable for that random effect. In this example, our random effect is specified as (only) a random intercept (signified by `1`), which is grouped by `subject`. This essentially tells `lmer()` that there is a random component of the intercept that varies by subject, and it means that `lmer()` will estimate the variance (standard deviation) of that component.

Now that we have successfully simulated and fit our first MEM (in this course, at least), let's take some time to practice!

Practice Exercise #1

1. Adapting the above code, make a function that will generate MEM-structured data. You should be able to feed your function i) "fixed" slope effect, ii) mean and iii) standard deviation for a random intercept, iv) residual variance (more precisely, standard deviation of errors), and v) number of subjects, and the function should return the full resulting data frame, including the generated random intercepts. For now, just assume that each subject has one observation per condition (for two conditions).
2. Test out your function with the following:
    a. Data with 10 subjects, so you can easily print out the entire data frame for inspection.
    b. Data with 1,000 subjects and relatively low values for the variance parameters, and then fit an `lmer()` model to the data to check that you are "recovering" the correct parameters you entered as function arguments.
3. The code above most transparently implements the conception of a random intercept as a distribution of intercepts, one for each subject. Revise your function in #1 to more transparently match the conception of a random intercept as a "fixed" intercept plus a random distribution of by-subject deviations from that fixed intercept. This should be a relatively minor change.
4. Using relatively large subject numbers, check that your two slightly differently formulated functions return equivalent results (with expectations of simulation variation, of course).
5. Return to the `sleep` data (again, I suggest making a local "copy" of the `sleep` data frame in your workspace). Perform the following analyses:
    a. Use the `t.test()` function with default arguments to perform a two-sample *t*-test comparing the means of the two groups in the data. (Hint: try using the formula `extra ~ group` in the function call.)
    b. Fit a simple linear model with `lm()` with the same structure (`group` predicting `extra`)
    c. Perform an omnibus ANOVA with the same structure as the `lm()`. You can use the `aov()` function from the `car` package, or you can also use the `anova()` function on the results of the `lm()` call from b.
    d. Now perform a paired *t*-test (look at the help with the `?t.test` command if you're not sure how to do this).
    e. Perform a within-subjects ANOVA using the `aov()` function from the `car` package. Hint: you can use the same formula as the `lm()`, adding `+ Error(ID)` at the end.

f. Fit a MEM with a random intercept by `ID`. Before fitting this model, load the `lmerTest` package, which will display p-values when you use `summary()` on the `lmer()` results object.

6. Look at all the results of the models from #5. Which ones return the same results?

Extension Exercise #1

1. Return to a linear model from the last assignment where you fit a simple model to your data.
   a. Add a random intercept term and fit with `lmer4()` (still as a linear model).
   b. How much do the "fixed effect" parameters differ?
   c. Do the standard errors of those parameters differ?

   d. What is the estimated standard deviation for the random intercept?
   e. How do you interpret the size of the random effect variance in your data?
2. Using the parameter values you extracted from #1:
   a. Use one of your functions from Practice Exercise #1 to simulate data with these parameters.

   b. Try graphing and numerical summaries to compare your original data with the simulated data.

   c. How do the data sets look similar or different?

"Crossed" vs. "nested" random effects

So far we have looked at only single random intercepts. But in experimental designs, this is the exception rather than the rule. For example, a reaction time study with only one observation per condition would be a very unusual design, and it is more likely that we would have multiple items per person per condition. And not only might we have more than one data point per person, but we may also have a common set of experimental items that all participants see. And it is very common to expect that different items will have different properties which could potentially impact responses. So again, we have the same kind of two-sided argument as before, in that items may have properties that could lead to differences in overall response time to different items, and also that there is possibly some shared variance across responses to the same item. All of this boils down to wanting to fit random intercepts both by subject and by item.

Fortunately, it is relatively straightforward to extend MEMs to include additional random effects, even when those effects are related to the same "fixed" effect (for example, both random intercepts). The extension is most straightforward when we think in terms of a fixed effect plus multiple random "deviation" effects. That is, we can think of our linear model including an overall fixed intercept effect, and then both a distribution of random effects for how each subject differs from this fixed effect as well as a distribution of random effects for how each item differs from the fixed effect.

These are sometimes called "crossed" random effects, to distinguish them from nested random effects. In terms of `lme4` and our simulation approach, there is not much of a reason to worry about this distinction. It is more important to simply think through the structure of data and what random effects are appropriate. The distinction can become more important when using different statistical packages, because some do not deal with crossed random effects as flexibly as `lme4`. The concept of nested random effects is simply that one of the random effect grouping factors is strictly nested within another. The classic example is students within schools, because the normal situation is that students only attend one school, so whatever the random effect of students is, it is nested under the random effect of schools.

The only time we need to care about nested effects in `lme4` is if the data is coded in a way that requires us to be explicit about the nesting. For example, if students are identified with numbers from 1 to n within each school, then it would mean that different students in different schools might all be labeled as, say, student #10. In this case, we could use the nested random effect syntax in `lmer()`. For example, if students are nested under schools:

```
fit <- lmer(response ~ condition + (1|school/student))
```

However, it may be more advisable to simply transform your data. I personally don't see the point of re-using something like subject IDs if they actually represent different people, especially because this can lead to some irritating mistakes, such as accidentally aggregating data across different individuals. My suggestion would just be to transform your ID variable to actually be different for different people (or whatever the unit is). When your data is coded in this way, then you don't need to explicitly tell `lmer()` that the random effect is nested, because it is handled the same either way.

Now let's get some practice with crossed random effects.

Practice Exercise #2

1. Start with the function you created for simulating random intercept data, specifically the version where you supply a fixed intercept effect (the one you should have made for exercise 1.3 above). Modify this function to be able to simulate both by-subject and by-item random effects. Note that you can use `expand.grid()` with multiple factors.
2. Simulate a small set of data (a couple of subjects and a couple of items) and print out the whole data set to inspect that you did what you intended.
3. Simulate a larger set of data (enough subjects and items), fit a crossed random intercept model with `lmer()` and confirm that you are getting the paramter values that you expect.

Extension Exercise #2

1. Return to your own data and the model you have been building up. Are there additional random intercept effects you can add? If so, fit a model and examine the results, comparing to previous models you have fit. If not, describe why there are not any additional random intercepts you can fit, and describe a hypothetical additional random intercept if you could augment your data, and what the data augmentation would look like.

Random slopes

Once we understand the concept of random intercepts, we can easily extend this understanding to random slopes. That is, in the same way that we can model a random intercept as a distribution of β_0 values, we can model a random slope as a distribution of β_1 values. The only difference is that the decision about whether to include a random slope can be a little more involved than the decision to include a random intercept.

The simplest rule of thumb, for any kind of random effect, not just slopes, is to ask yourself, "does it make sense that this parameter could vary by [grouping]?" In our hypothetical reaction time experiment, we already discussed how it makes sense for the intercept (interpreted as a baseline reaction time) to vary by person. Recall also that we had both theoretical and statistical perspectives on why this makes sense. Theoretically, we know that people are different, and we expect that some people are just generally faster/slower than others. Statistically, we expect that since more than one observation is gathered from each person, there should be some common variance between those observations (i.e., they are not as independent as they would be if they all came from different people), and our model should take that into account, instead of assuming independence. Again, these two ways of looking at the rationale for a random effect are really two sides of the same coin, but we will explore various circumstances in which there may be some good reason to make a decision on a statistical basis, more as a practical matter. But it's useful to think about it both ways, when you are trying to decide whether to include a random effect.

Turning to random slopes in particular, let's look back again at the `sleep` data. We have seen (in practice exercise 1.5 above) how a random intercept makes sense, and that fitting a random intercept model with `lmer()` gives us virtually identical results (in terms of inferences, at least) as a paired t-test of the difference

of means, or a repeated-measures one-way ANOVA. This should be reassuring. But what about a random slope?

We can ask ourselves, "does it make sense to say that the effect of `group` could vary by subject?" Theoretically, this might make sense, with the idea that whatever the `group` treatment is, some people might respond more to it than others. But statistically, it doesn't make sense because we only have one observation per subject per condition. The issue here is that a random effect model needs to be able to estimate the random effect variance as well as the error variance, but when there is only one observation per person per condition, there is no way to estimate those separately.[4]

Here's one way to think about this a bit more intuitively. We can look at people and see that the difference between `group` conditions is in fact different for different people. But if you think about how you would predict the "group 2" value for a person, you would have an estimated effect of condition (i.e., the difference between groups), but then you would have both a residual error effect (i.e., measurement-by-measurement error) and a random subject effect (i.e., the degree to which that person's group effect differs from the average group effect). And even though we can see that the apparent effect of group is different between people, we can't tell whether that's actually the variance of the slope itself, or just variance due to normal error variance. The reason we can't tell is that we don't have multiple observations in order to get something like a "person-level estimate" of the effect of `group` that's distinct from the measurement-level errors.

Another common example of a variable for which a random slope doesn't make sense is something like a person-level covariate. For example, imagine in addition to the data in the `sleep` data frame, we had some other variable like age. Then we could ask ourselves "does it make sense that the effect of age could vary by person?" The answer here is clearly no, because we simply don't have data from the same person at different ages, so there's no way to even observe whether the effect of age varies within a person. Again, maybe theoretically it could make sense, if we think that aging could have different impacts on different people. But if we only have one value of age per person, we simply don't have the data for it to be viable in our model.

The point here is that the hard part is thinking through what should and shouldn't be a random effect. Constructing a model for simulation or fitting is relatively straightforward. But if you force yourself to think through what it would mean to simulate data from a particular model, that can help you reason about whether a given random effect structure makes sense. So let's simulate!

All we really need to do is the same for what we did with random intercepts, and that is add parameters for the standard deviation of the random effect distribution(s). In the crossed random intercept models above, we moved to specifying a fixed effect intercept and modeled the random effect as a random distribution of deviances from this fixed effect (i.e., with a mean of zero), so let's continue in that direction to simplify things.

First, let's specify the parameters we need for a model that has both a random intercept and random slope by subject.

```
beta_0 <- 600
beta_1 <- 75
beta_0_ranef_bysubj_sd <- 100
beta_1_ranef_bysubj_sd <- 25
error_sd <- 50
```

Next, we create the initial data frame, making sure we have repeated measures so that a random slope will make sense.

```
subjects <- paste0("s", 1:5)
items <- paste0("i", 1:5)
conditions <- 0:1

df <- expand.grid(subjects, items, conditions,
```

[4] The technical term for this is "unidentifiability".

```
                  stringsAsFactors = FALSE)
colnames(df) <- c("subject", "item", "condition")
```

As before, we create a separate table where we can store the random effects. The only difference is that each subject has both a random intercept and a random slope, and crucially, the standard deviations of those two different random effects are different (i.e., set by different parameters – there's nothing preventing them from taking the same value, by coincidence).

```
ranefs_bysubject <- data.frame(subject = subjects,
                               beta_0_ranef_bysubj = rnorm(length(subjects),
                                                           0,
                                                           beta_0_ranef_bysubj_sd),
                               beta_1_ranef_bysubj = rnorm(length(subjects),
                                                           0,
                                                           beta_1_ranef_bysubj_sd),
                               stringsAsFactors = FALSE)
```

Finally, we can just merge the random effect table with our main data frame, and then use our parameters and the random effects we merged in to simulate our response vector. **NB**: be careful with your parentheses! Crucially when we are talking about slopes, we need to multiply our predictor values (here, our condition value) by the *sum* of the fixed effect and by-subject random effect.

```
df <- df %>% left_join(ranefs_bysubject, by = "subject") %>%
      mutate(y = (beta_0 + beta_0_ranef_bysubj) +             # intercept
                 (beta_1 + beta_1_ranef_bysubj) * condition + # slope
                 rnorm(nrow(.), 0, error_sd))                 # error
```

Other than the addition of a term, this should all be starting to look pretty familiar. We are still following the same general procedure of i) set up parameters, ii) generate a data frame with predictors, iii) use the linear model equation to simulate responses. Including random effects means that we need to add to our linear equation, and it requires us to simulate the random effects so that they can be included in the equation. Generating those random effects in separate data frames and merging them is a convenient technique.

So now let's get some practice and extend this to cover more models. But first, a few tips regarding model-fitting with `lme4`.

Tip #1*: When you start running random slope models, especially if there are multiple random slopes and a lot of data, the model-fitted process can get pretty slow. I recommend wrapping your code inside a call to `system.time()`, which will print out to the console how long it took. So after the first time you fit a model, you can have a clear expectation for how long it takes. For example, if you start it running and go get some coffee and come back after 30 minutes and it finished, it might be nice to know whether it took 3 minutes or 29 minutes. . .

Tip #2: Depending on the parameter values, number of observations, etc., when you start playing around with more complex models, `lme4` will often return messages about convergence problems. We are going to spend some significant time and energy talking about convergence later. For now, the relevant points are i) `lme4` tends to be pretty cautious about model convergence, so many warnings can turn out to be "false positives," and ii) since we are using model fits to check whether we can "recover" our simulation parameters, we can see for ourselves whether convergence warnings resulted in poor parameter estimates. Bottom line: for our present purposes, don't worry too much about convergence.

Tip #3: We are currently simulating data where the random effects are uncorrelated. That is, we are generating the random effects from independent normal distributions, and when you do that they are (in theory) uncorrelated. The real world isn't always like that. We will discuss correlated random effects in greater detail later, but but for now, I just want to note that `lme4` has a convenient way to assume uncorrelated random effects. This can also sometimes speed up your model-fitting, so if things are just taking too long for your tastes, you can try this syntax to see if it improves. All you do is use a "double bar" in the random

effect terms, like so:

```
fit <- lmer(y ~ condition + (1 + condition || subject), data = yourdata)
```

Again, we'll discuss this more later, but it might be handy in the following exercises if you are running into some issues with the fitting process.

Tip #4: You will probably run into a message about a "singular" fit in Practice 3.5.b/c below. This is actually expected since we are simulating data with a random effect of zero for one or more effects. In reality, things are rarely exactly zero, so this can be a flag for some issue with your data if you encounter it "in the wild," but for the purposes of our present exercises, you can safely ignore this message.

Practice Exercise #3

1. Using the code above as a foundation, create a function that you can use to simulate a random intercept and slope for a single random effect (i.e., by-subjects).
2. Test out this function by generating a relatively large data set, fitting a model, and confirming that your fitted model returns parameter values close to the ones you supplied as arguments.
3. Make this function even more general by adding the ability to input by-item random effects (intercept and slope) in addition to by-subjects (i.e., "crossed" random effect models).
4. Test out your function from #3 with the following:
    a. Generate a relatively large amount of data, and confirm that you can "recover" the parameter values you entered as arguments by fitting a model to this data.
    b. Do the same, but simulate data from a model in which intercepts and slopes do not actually vary for items. Again, use a fitted model to double-check.
    c. Do the same as b., but instead from a model where effects do not vary by subject but they do by item.

Extension Exercise #3

1. Return to your own data, and fit a model with more than one random effect. This might be subjects and items, or it may be something else, depending on your data. First fit a model with just random intercepts.
2. Do random slopes make sense for your data? If so, fit a random slope model. How do other parameters appear to change from your random intercept model?
3. Now using your fitted model and the functions you developed above, simulate data that matches the paramters in your data (according to your fitted model). Using plots, `summary()`, etc., compare your real data with the simulated data. Are there any striking differences? Why might that be?

BLUPs, estimation, and pooling

Now we turn to a few final concepts, to try to help you nail down your conceptualization of random effects, as well as build an understanding of why MEMs are often better than alternatives. Return to the idea of just a simple random intercept. We discussed this above as a distribution of intercepts, one for each subject. But what about some different ways of handling the clustered nature of this data? Let's consider a few.

First, if we simply ignore the structure of our data and fit a model without a random effect, like a simple regression or omnibus ANOVA, this is called "complete pooling." If the data is balanced, we should end up getting similar parameter estimates, but the standard errors will be off. They could be much smaller or much larger, depending on the data. So in essence, complete pooling is a very bad way to assess the uncertainty around your parameter estimates. If our goal is drawing inferences, this is bad. Additionally, if the data is unbalanced, the parameter estimates themselves can get pretty off. Also not good.

Second, we could fit a separate model to each subject, and then figure out a way to pool our inferences across all subjects. Hopefully this immediately strikes you as a bad idea, or at least really inconvenient. You would in effect be trying to perform a kind of meta-analysis, treating each subject as a separate study. This might actually get some decent results, in terms of leading you towards reasonable inferences, but there are some issues. The main issue is that you will get fairly poor estimates for each individual subject, and the sum of many poor estimates is not necessarily as good as a single estimate. This is called "no-pooling," because you are not making good use of the combination of data from different people. This is shooting yourself in the foot in the opposite way as complete pooling.

A slightly more convenient way to accomplish a no-pooling analysis than fitting separate models is to include a predictor that represent your grouping factor (e.g., subject), and directly estimate each subject's intercept (or other parameter of interest). At some level, this even sounds a little like what we are doing with a MEM, because we are getting a different intercept value for each person.

As you might suspect, MEMs are the kind of model that would make Goldilocks happy, by striking a middle ground between complete pooling and no pooling, referred to as *partial pooling*. The intuition behind partial pooling is that while there is a different intercept (or other parameter) for each subject, the value of each intercept is informed by the other intercepts.

How does this work? In `lme4`, this comes down to a result of how the random effects are estimated. Recall that `lme4` does not actually *estimate* each individual random intercept, in the sense that these are not individual parameters in the model. Instead, it estimates the fixed effect (conceptually similar to the mean of the distribution) and the standard deviation of the distribution of intercepts – just two parameters. But when it makes this estimation, subjects can contribute different degrees of information, depending (in large part) on how much data there is.

For example, imagine a situation in which we have 50 observations for most participants, and they have intercept values that generally range from 500 to 600 (again, maybe imagine reaction times), but with fairly large error variance. But then we have one person who only has 10 observations, with an apparent intercept of 800. A no-pooling estimate would just estimate their intercept as 800, with more uncertainty than the estimates of the other subjects. A complete-pooling estimate would just fold this person in with the rest, which might drag up the global estimate more than it should, but then if your global intercept is estimated at, say, 575, that may be much too low for that person. A partial-pooling estimate would "shrink" this person's estimate towards the mean, as a result of the greater uncertainty. Intuitively, while this person might seem to be an "outlier," it may just be a by-product of having fewer observations, and so the best estimate for that person's individual intercept is somewhere between the 575 "grand mean" and the 800 "individual mean."

In packages like `lme4`, this "best estimate" is called a Best Linear Unbiased Predictor, or "BLUP." Technically, this is not an "estimate," in the sense that BLUPs are not estimated parameters of the model, but rather calculated as the most likely values of the random effects for different subjects, given the fixed effect and random effect variance, which *are* estimated model parameters. You can access the BLUPs from the fitted `lmer` object itself.

Let's look at this with the `sleep` data. Let's fit three models. First the complete pooling model (i.e., ignoring the repeated-measures or "paired" structure of the data), second a no-pooling model with `ID` as a factor, and third the appropriate partial-pooling random intercept model. Then let's look at the model summaries, but then also look at the by-subject BLUPs in the final model.

```
mysleep <- sleep
fit_complete_pooling <- lm(extra ~ 1 + group, data = mysleep)
fit_no_pooling <- lm(extra ~ 1 + group + ID, data = mysleep)
fit_partial_pooling <- lmer(extra ~ 1 + group + (1|ID), data = mysleep)

summary(fit_complete_pooling)
```

```
##
## Call:
## lm(formula = extra ~ 1 + group, data = mysleep)
```

```
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -2.430 -1.305 -0.580  1.455  3.170
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   0.7500     0.6004   1.249   0.2276
## group2        1.5800     0.8491   1.861   0.0792 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.899 on 18 degrees of freedom
## Multiple R-squared:  0.1613, Adjusted R-squared:  0.1147
## F-statistic: 3.463 on 1 and 18 DF,  p-value: 0.07919
```

```
summary(fit_no_pooling)
```

```
##
## Call:
## lm(formula = extra ~ 1 + group + ID, data = mysleep)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -1.510 -0.215  0.000  0.215  1.510
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   0.5100     0.6450   0.791  0.44946
## group2        1.5800     0.3890   4.062  0.00283 **
## ID2          -1.7000     0.8697  -1.955  0.08235 .
## ID3          -0.8500     0.8697  -0.977  0.35395
## ID4          -1.8500     0.8697  -2.127  0.06232 .
## ID5          -1.4000     0.8697  -1.610  0.14193
## ID6           2.6000     0.8697   2.989  0.01522 *
## ID7           3.3000     0.8697   3.794  0.00425 **
## ID8          -0.1000     0.8697  -0.115  0.91099
## ID9           1.0000     0.8697   1.150  0.27987
## ID10          1.4000     0.8697   1.610  0.14193
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.8697 on 9 degrees of freedom
## Multiple R-squared:  0.912,  Adjusted R-squared:  0.8142
## F-statistic: 9.328 on 10 and 9 DF,  p-value: 0.001254
```

```
summary(fit_partial_pooling)
```

```
## Linear mixed model fit by REML ['lmerMod']
## Formula: extra ~ 1 + group + (1 | ID)
##    Data: mysleep
##
## REML criterion at convergence: 70
##
## Scaled residuals:
```

```
##      Min       1Q   Median       3Q      Max
## -1.63372 -0.34157  0.03346  0.31511  1.83859
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  ID       (Intercept) 2.8483   1.6877
##  Residual             0.7564   0.8697
## Number of obs: 20, groups:  ID, 10
##
## Fixed effects:
##             Estimate Std. Error t value
## (Intercept)   0.7500     0.6004   1.249
## group2        1.5800     0.3890   4.062
##
## Correlation of Fixed Effects:
##        (Intr)
## group2 -0.324
```

```
ranef(fit_partial_pooling)
```

```
## $ID
##    (Intercept)
## 1   -0.2118668
## 2   -1.7125900
## 3   -0.9622284
## 4   -1.8450067
## 5   -1.4477565
## 6    2.0833569
## 7    2.7013017
## 8   -0.3001446
## 9    0.6709115
## 10   1.0240229
##
## with conditional variances for "ID"
```

So let's compare. Recall that this data is balanced: every participant has a single data point in each of two conditions. So when we look at the intercept and `group` coefficient estimates, we can see that they are identical between the complete-pooling (simple `lm`) and partial-pooling (MEM) models (intercept and slope of 0.75 and 1.58, respectively). However, there is a substantial difference in the uncertainty of those estimates, such that the standard error of (for example) the `group` effect is much larger in the complete-pooling case (0.85, compared to 0.39 in the MEM). This is as I said above: complete pooling can sometimes get the same point-estimate values as a MEM, but it can get the uncertainty very wrong (either way, depending on the data).

Now let's look at the no-pooling case. Here, the estimate of `group` is actually the exact same as the estimate from the MEM, including the same standard error. This is as it should be, because these two model are in effect doing the same thing, by estimating the effect of group after partialling out the subject-to-subject variance in the intercept. Again, these two models get the same result because the data is balanced. Where they differ is in the intercept estimate. In the no-pooling model, what we get by default is the intercept estimate for `ID` 1, and then parameters for how much each other subject differs from subject 1. In contrast, the BLUPs represent the distribution of by-subject differences from the fixed effect intercept (which here is the mean intercept, since we have balanced data).

In order to look at the comparable no-pooling effects, we can use "sum" coding. That is, by default, R uses what is called "treatment" coding when converting the categorical factor `ID` into dummy-coded variables for the purpose of the model. While treatment coding is often a convenient way to look at how levels of a

factor differ from a "reference" level, in this case what we want is for the intercept to represent the *marginal* intercept (again, which is equivalent to the mean because the data is balanced), and each subject's value to be the estimated difference from this mean. We want this here because this is analogous to what our BLUPs represent. The sum-coded model summary is given below, along with the BLUPs from the MEM again, for easy comparison.

```
mysleep$ID_sum <- mysleep$ID
contrasts(mysleep$ID)
```

```
##    2 3 4 5 6 7 8 9 10
## 1  0 0 0 0 0 0 0 0  0
## 2  1 0 0 0 0 0 0 0  0
## 3  0 1 0 0 0 0 0 0  0
## 4  0 0 1 0 0 0 0 0  0
## 5  0 0 0 1 0 0 0 0  0
## 6  0 0 0 0 1 0 0 0  0
## 7  0 0 0 0 0 1 0 0  0
## 8  0 0 0 0 0 0 1 0  0
## 9  0 0 0 0 0 0 0 1  0
## 10 0 0 0 0 0 0 0 0  1
```

```
contrasts(mysleep$ID_sum) <- contr.sum(levels(mysleep$ID_sum))
contrasts(mysleep$ID_sum)
```

```
##    [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## 1     1    0    0    0    0    0    0    0    0
## 2     0    1    0    0    0    0    0    0    0
## 3     0    0    1    0    0    0    0    0    0
## 4     0    0    0    1    0    0    0    0    0
## 5     0    0    0    0    1    0    0    0    0
## 6     0    0    0    0    0    1    0    0    0
## 7     0    0    0    0    0    0    1    0    0
## 8     0    0    0    0    0    0    0    1    0
## 9     0    0    0    0    0    0    0    0    1
## 10   -1   -1   -1   -1   -1   -1   -1   -1   -1
```

```
summary(lm(extra ~ group + ID_sum, mysleep))
```

```
##
## Call:
## lm(formula = extra ~ group + ID_sum, data = mysleep)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -1.510 -0.215  0.000  0.215  1.510
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   0.7500     0.2750   2.727 0.023342 *
## group2        1.5800     0.3890   4.062 0.002833 **
## ID_sum1      -0.2400     0.5834  -0.411 0.690435
## ID_sum2      -1.9400     0.5834  -3.325 0.008868 **
## ID_sum3      -1.0900     0.5834  -1.868 0.094564 .
## ID_sum4      -2.0900     0.5834  -3.582 0.005910 **
## ID_sum5      -1.6400     0.5834  -2.811 0.020346 *
## ID_sum6       2.3600     0.5834   4.045 0.002907 **
```

```
## ID_sum7       3.0600     0.5834   5.245 0.000531 ***
## ID_sum8      -0.3400     0.5834  -0.583 0.574368
## ID_sum9       0.7600     0.5834   1.303 0.225047
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.8697 on 9 degrees of freedom
## Multiple R-squared:  0.912,  Adjusted R-squared:  0.8142
## F-statistic: 9.328 on 10 and 9 DF,  p-value: 0.001254
```

```
ranef(fit_partial_pooling)
```

```
## $ID
##    (Intercept)
## 1   -0.2118668
## 2   -1.7125900
## 3   -0.9622284
## 4   -1.8450067
## 5   -1.4477565
## 6    2.0833569
## 7    2.7013017
## 8   -0.3001446
## 9    0.6709115
## 10   1.0240229
##
## with conditional variances for "ID"
```

Notice a few things. First, the parameter value for the intercept is now the same for both models, but the no-pooling model now greatly underestimates the uncertainty in that estimate (leading to much-too-small standard error, as compared to the MEM). This is because the interpretation of this is slightly different, as it is not taking into account the uncertainty in the individual subject estimates. Second, as expected our slope parameter `group` is still the same, since it does not vary by subject. Third, the by-subject "slopes" for `ID_sum` are very close to the BLUP values from our MEM, but they are all a bit more "extreme." That is, this is an illustration of how the BLUPs are "shrunk" towards the mean, as compared to no-pooling estimates.

Again, this is a good thing! There is a fair amount of uncertainty in all of the individual by-subject intercept estimates. The failing of the no-pooling model is that all of these individual estimates essentially "ignore" each other. This is equivalent to simply computing by-subject means, without paying attention to the means of other subjects. When you do partial pooling, which takes information from the whole group of subjects, then then "estimated" means are better-informed, and they all shrink towards the overall mean a bit. Intuitively, when we see an "extreme" subject, it is a very good bet that their underlying by-subject parameter is not as extreme as it appears, because we can see what the overall distribution looks like. This shares an intuitive rationale with the general phenomenon of "regression to the mean," if that helps you understand how this happens.

Let's recap with the main take-aways. The first take-away is that this is one of the important advantages of MEMs, because partial pooling results in better inferences, both about the by-subject (or by-whatever) variation and about the fixed effects. When data is balanced, we can see that complete-pooling and no-pooling can get some parameter estimates right (e.g., intercept and slope), in terms of the point-estimate of the most likely value for that parameter, but they can be off a great deal in terms of the uncertainty around those estimates, which is crucial for proper inferences.

The second take-away is that when you fit a MEM with `lmer()`, you can easily access the BLUPs. These are conceptually similar to estimated differences from the fixed effect for each subject (or item or whatever), but they show *shrinkage* towards the mean (i.e., towards the fixed effect) as compared to what you would get if you fit a model with subject as a fixed effect. Assuming we are using an appropriate random effect structure, BLUPs therefore provide better "estimates" (in scare quotes because they are technically not

estimated parameters of the model) of the underlying by-subject values than values you might get if you tried estimating them with a fixed effect. This is especially true when the data are unbalanced.

So let's try out some simulations!

Practice Exercise #4

1. Use one of your previously created functions to simulate some relatively simple data with just a by-subject random intercept (i.e., no by-item random effects). Use a relatively large number of items per subject, but only 10 subjects.
2. Fit a no-pooling model to this data. Create a new version of your `subject` factor and set the contrasts to be "sum" coded. Then fit a simple `lm()` model with this sum-coded subject variable as a fixed effect, along with your condition effect. Compare the fixed-effect by-subject estimates from this model to the "ground truth" random effect values used to simulate your data. Try plotting one against the other.
3. Fit a random-intercept MEM to this same data (use the regular `subject` variable as a random effect). Extract the BLUPs for the by-subject random effect, and compare those to the simulated "ground truth" values. Plot them against each other.
4. Now compare (plot) the BLUPs vs. the no-pooling estimates. What do you notice?

Extension Exercise #4

1. Take a subset of your own data with just a few subjects (or a few items, or other grouping). Fit a simple MEM with a random intercept to both the full data and the smaller set.
2. Compare the BLUPs for the same set of subjects in both models. How do they appear to differ? Why do you think they differ in this way?
3. Now fit a no-pooling model for the same subset of subjects (again using `contr.sum()` to make them easier to compare to BLUPs). How do these compare with the BLUPs from your two other models?
4. How do your fixed-effect estimates (and standard errors) differ between these models?
5. One thing that BLUPs do not provide is a direct estimate of the uncertainty around them. If you wanted to make inferences about whether one subject's BLUP is larger than another, how might one do that? If you can think of how, give it a try!

TL;DR summary bullets

- Random effects are modeled as distributions of parameters.
- The parameters that control/describe these distributions are called *hyperparameters* or *hierarchical* parameters.

- There is a mathematical and conceptual equivalence between the mean of a random effect distribution (e.g., the mean of the random intercepts by person) and a "fixed effect" for that parameter.
- "Crossed" random effects are not special, they just require an additional distribution of effects, based on a different grouping/clustering factor.
- Random slopes are not (much) different than random intercepts, in the abstract. In reality, they may require a bit more thought.
- When considering a random effect, ask yourself (at least) the following:
    1. Does it make theoretical sense that [parameter, e.g. slope] could/would/should vary by [grouping factor, e.g. subject]?
    2. Does my data structure imply grouping/clustering of observations such that they might have some shared variance within a cluster?
    3. Does my data have sufficient structure such that a model can estimate group-by-group variance in addition to observation-by-observation error variance?

- BLUPs have a conceptual similary to by-group estimates. Under certain conditions, these estimates may closely match the estimates that would be produced if you fit a plain regression with the grouping factor as a fixed effect. However, especially when data are not balanced perfectly across groups, BLUPs may benefit from "partial pooling" as a result of the random effect estimation process. This is a *good thing*, and is a major reason for preferring MEMs over alternative ways of handling clustered data.