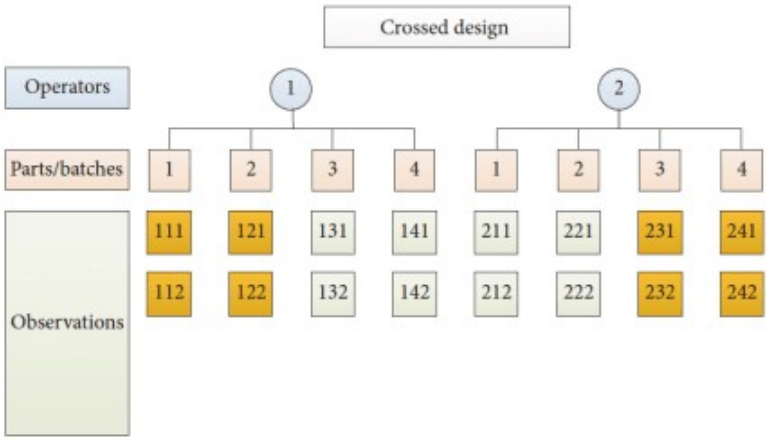
I've heard it said that common statistical tests are just linear models.[1] It turns out that Gage R&R, a commonly used measurement system analysis (MSA), is no different. In this post I'll attempt to provide some background on Gage R&R, describe the underlying model, and then walk through a method for simulation that can be useful for things like power analysis or visualization of uncertainty.

What is Gage R&R?

When evaluating implantable medical devices it is generally necessary to perform the following types of inspection to ensure product quality:

- Dimensional inspection of implant features
- Visual inspection of implant surface and component interfaces
- Benchtop performance evaluation of implant

The primary purpose of these costly and rigorous inspection processes is to screen out bad product. The ramifications of non-conforming parts reaching the field can be fatal. As such, it is important to understand any limitations of key measurement systems and, if possible, quantity their uncertainty. The primary statistical tool for this job is called Gage R&R. The Gage R&R attempts to quantify the total variation within a series of measurements and then describe the relative contributions of parts, operators, and repeated measurements (unexplained error). Operator error is called "reproducibility"; unexplained error when a measurement is repeated under presumably identical conditions is called "repeatability". The total variation among these components must be controlled and limited. A typical crossed structure is shown below: [2]



Gage R&R training for engineers usually involves:

- Definitions of repeatability and reproducibility (the "R&R")
- Guidance for directing Minitab to set up the experiment
- Guidance for directing Minitab to analyze the results and provide an output
- An acceptance criteria (with no or limited context)

When I was first exposed to the material, I recall grappling with terminology and definitions, struggling with rote memorization, and having no understanding of the assumptions or limitations of the technique. Here's the piece that was never explained to me (and many other engineers):

> A Gage R&R study is a random effects regression model with two random variables: operator and part. By modeling the factors as random effects and applying a few assumptions, we can access and analyze the variance associated with each component using standard ANOVA techniques.

Specifically, the model commonly used for a crossed Gage R&R is:

$$y_{ijk} = \mu + O_i + P_j + (PO)_{ij} + E_{(ij)k}$$ where:

y_{ijk} = a specific, individual measurement
μ = overall mean of all the measurements

O = random variable for effect of operator. Assumed normal: $O_i \sim N(0,\sigma^2_O)$
P = random variable for effect of part. Assumed normal: $P_i \sim N(0,\sigma^2_P)$
(PO) = random variable for sample x operator interaction. Assumed normal: $(PO)_{ij} \sim N(0,\sigma^2_{PO})$
E = random variable for unexplained, residual error (referred to as "repeatability" since differences in measurements taken under identical conditions are mapped here). $E_{(ij)k} \sim N(0,\sigma^2_{E})$

This is really cool! – Understanding the model unlocks the insight behind the method and casts a bright light on the assumptions. It puts a seemingly obscure, memorized technique into a familiar regression framework. It also facilitates simulation.

Why is it a random effects model? What is a random effects model? The answer to that question is actually tricky (and beyond the scope of this post) but there is some good information here for those who want to dive deeper. [3] While not a formal definition, it may be sufficient to know that random effects are estimated with partial pooling while others are not.

In this post I will attempt to show how to use the lme4 to simulate outcomes using a random effects model like the one listed above and then repeat many such simulations to gain understanding of uncertainty and sensitivity in the underlying experiment.[4] Fun!

Here are the libraries we'll use.

```
library(lme4)
library(tidyverse)
library(knitr)
library(here)
library(broom.mixed)
library(tidybayes)
library(DiagrammeR)
```

Simulating One Outcome of a Gage R&R Experiment

Before we worry about running a bunch of simulations, let's just figure out how to run one instance of a Gage R&R. There are some really good tutorials out there for simulating outcomes from random and mixed effects models.[5] [6] It ends up being a little bit tricky because the parameters that you select need to combine within the design matrix in a certain way such that an analysis of the simulated outcomes can recover the specified parameters. If you are good at matrix math you can do this manually. I am not very good at matrix mat – but fortunately Robert Long on Cross Validated[7] showed me a really cool little hack for figuring out the form of the Design Matrix Z and using it for simulation. Basically, we will first set up a dummy experiment with the desired number of parts, operators, and measurements and then use lme4 to extract Z and store it. Then we can set up a vector of all our simulated random effects and combine them with matrix multiplication to build the simulated observation. It sounds tricky but it's surprisingly simple! in summary, here is the plan:

The good news is that none of the above steps are very hard, even if they look unfamiliar. Let's go through it.

Step 1: Set up a dummy experiment

I call this a dummy experiment because while it will have the proper number of operators/parts/replicates, we'll just drop in some dummy data as the observations. The goal here is to allow lme4 to create the structure of the experiment (the design matrix) which we can use later to get the real simulated observations.

First, we specify the number of parts, operators, and measurements we want the experiment be comprised of. 10, 3, and 2, respectively, is a common experimental setup in industry and we use it here.

```
n_part <- 10 # number of parts
n_oper <- 3 # number of opers
n_measurements <- 2 # number of replications
```

Now assign names to each part, operator, trial and determine how many observations will be in the study: n_matrix.

```
# assign names to each part, operator, trial
part <- str_glue("part_{1:n_part}") %>% as_factor()
operator <- str_glue("oper_{1:n_oper}") %>% as_factor()
measurement <- str_glue("measurment_{1:n_measurements}") %>% as_factor()

n_matrix <- n_part * n_oper * n_measurements # number of observations in the study

n_matrix
```

```
## [1] 60
```

Now we use crossing() to build the full set of experimental conditions. Dummy observations are created for each setting and assigned to a new col: "measurement". Overall mean of 10 is chosen arbitrarily and isn't important. Note: we are creating observations(measurements) here but have not concerned ourselves with specifying the parameters of any random variables yet. We just need a placeholder in the measurement column.

```
# generate experimental design and outcomes for dummy study
grr_dummy_tbl <- crossing(part, operator, measurement) %>%
  mutate(measurement = 10 + rnorm(n_matrix))

grr_dummy_tbl %>%
  head(7) %>%
  kable(align = "c")
```

part	operator	measurement
part_1	oper_1	11.521931
part_1	oper_1	11.434057
part_1	oper_2	10.773945
part_1	oper_2	11.629946
part_1	oper_3	10.672087

part	operator	measurement
part_1	oper_3	9.976531
part_2	oper_1	11.543520

```
grr_dummy_tbl %>%
  tail(7) %>%
  kable(align = "c")
```

part	operator	measurement
part_9	oper_3	10.001443
part_10	oper_1	9.087449
part_10	oper_1	10.722729
part_10	oper_2	9.975832
part_10	oper_2	8.495529
part_10	oper_3	9.980824
part_10	oper_3	11.526295

Step 2: Fit a model to the dummy data

Now we fit a model to the dummy data. The summary isn't important – we just want access to the structure which we will get in the next step.

```
# fit model for dummy study
m1 <- lmer(measurement ~ (1 | part) + (1 | operator) + (1 | part:operator), data =
grr_dummy_tbl)
```

Step 3: Extract and store the Design Matrix *Z* from the dummy model

Here's the little hack: Use getME() to pull the design matrix *Z* from the dummy model. Alternately, you can use lFormula() but you will have to fish the matrix out and transpose it which is not as intuitive to me.

```
# extract design matrix Z from dummy model
design_matrix_Z <- getME(m1, "Z") %>% as.matrix()

design_matrix_Z %>% head(1)
```

```
##   part_1:oper_1 part_1:oper_2 part_1:oper_3 part_2:oper_1 part_2:oper_2
## 1             1             0             0             0             0
##   part_2:oper_3 part_3:oper_1 part_3:oper_2 part_3:oper_3 part_4:oper_1
## 1             0             0             0             0             0
##   part_4:oper_2 part_4:oper_3 part_5:oper_1 part_5:oper_2 part_5:oper_3
## 1             0             0             0             0             0
##   part_6:oper_1 part_6:oper_2 part_6:oper_3 part_7:oper_1 part_7:oper_2
## 1             0             0             0             0             0
##   part_7:oper_3 part_8:oper_1 part_8:oper_2 part_8:oper_3 part_9:oper_1
## 1             0             0             0             0             0
##   part_9:oper_2 part_9:oper_3 part_10:oper_1 part_10:oper_2 part_10:oper_3
## 1             0             0              0              0             0
##   part_1 part_2 part_3 part_4 part_5 part_6 part_7 part_8 part_9 part_10
## 1      1      0      0      0      0      0      0      0      0       0
##   oper_1 oper_2 oper_3
## 1      1      0      0
```

```
# alternate method:
# mylF <- lFormula(m1, data = grr_dummy_tbl) # Process the formula against the data
# design_matrix_Z <- mylF$reTrms$Zt %>% as.matrix() %>% t()  # Extract the Z matrix
```

So much easier than constructing this matrix yourself! (at least for me – it's been a while now since I took linear algebra course and tensor notation was always challenging).

Step 4: Simulate random effects using rnorm or similar

With the matrix Z in hand we can get get rid of the dummy model and get down to business with specifying and simulating our random effects. Specify standard deviations for each effect and simulate using rnorm(). 1, 2, 9, and 4 are the parameters that we will compare our estimates against later on.

```
set.seed(0118)
int_intercepts_sd <- 1 # standard dev of interaction random effects
oper_intercepts_sd <- 2 # standard dev of operator random effects
part_intercepts_sd <- 9 # standard dev of operator random effects
random_error_repeatability <- 4 # standard dev of random error (repeatability)

# simulate random effects using input params for sd
int_intercepts <- rnorm(n = n_part * n_oper, mean = 0, sd = int_intercepts_sd)
oper_intercepts <- rnorm(n = n_oper, mean = 0, sd = oper_intercepts_sd)
part_intercepts <- rnorm(n = n_part, mean = 0, sd = part_intercepts_sd)
```

Combine all the random effects into a vector. Order does matter here – see comment below.

```
# vector of all random effect intercepts (order matters here: ineraction, part,
oper if n_oper < n_part, else swith part and oper)

random_effects_intercepts <- c(int_intercepts, part_intercepts, oper_intercepts)
random_effects_intercepts
```

```
##  [1]  -1.676079782   0.167651720  -0.008545182   0.296888139  -1.706489201
##  [6]  -1.049094451  -0.102206749   0.682492401  -0.511117703   0.487346673
## [11]  -1.345812057   0.527128496   0.686104071  -0.221484626   0.532399538
## [16]   0.597393622   0.831437918   0.735023009   0.830043214   0.769163682
## [21]   1.830416344   0.182049999   1.018859437   0.844012288   0.575312043
## [26]   0.006855854  -0.231251230   0.205834471   0.250942908  -1.575663200
## [31]  19.280822421  10.499576059  -3.997253469  -7.111499870   6.986996777
## [36]  -4.861684848 -13.031232590  14.723410605  16.967969396  22.293922487
## [41]  -1.345816596  -3.905496830  -4.061788248
```

Step 5: Multiply Z by random effects vector

Step 6: Combine result with simulated residual error to generate simulated observations

We'll do steps 5 and 6 together here: multiply the design matrix Z by the vector of random effects intercepts and then add in a residual error. %*% is the matrix multiplication operator. Again – the overall mean of 10 is arbitrary and does not change the analysis.

```
# create observations (add in repeatability random error to each term). %*% is
matrix multiplication
grr_sim_tbl <- grr_dummy_tbl %>%
  mutate(measurement = 10 + design_matrix_Z %*% random_effects_intercepts + rnorm(
    n = nrow(grr_dummy_tbl),
    mean = 0,
    sd = random_error_repeatability
  ))

grr_sim_tbl %>%
  head(10) %>%
  kable(align = "c")
```

part	operator	measurement
part_1	oper_1	25.335155
part_1	oper_1	28.246367
part_1	oper_2	15.799074
part_1	oper_2	23.051405
part_1	oper_3	22.791039
part_1	oper_3	17.986515
part_2	oper_1	19.378715
part_2	oper_1	14.934401
part_2	oper_2	9.109815
part_2	oper_2	14.250238

Step 7: Fit a model to the simulated obervations

Once again we fit a model, but this time the observations are meaningful because we constructed them properly using the design matrix Z. broom.mixed::tidy() is able to bring the results into tibble format where we can clean a bit and view the variance contribution of each variable. This gives a point estimate of standard deviations for the random effects that can be compared against the reference inputs.

```
# fit a model to the simulated dataset
sim_m_fit <- lmer(measurement ~ (1 | part) + (1 | operator) + (1 | part:operator),
data = grr_sim_tbl)

# tibble of results for a single simulation
one_grr_result_tbl <-
  broom.mixed::tidy(sim_m_fit, effects = "ran_pars") %>%
  rename(
    st_dev_estimate = estimate,
    variable = group
  ) %>%
  mutate(
    variance_estimate = st_dev_estimate^2,
    sim_number = 1
  ) %>%
  select(sim_number, variable, st_dev_estimate, variance_estimate)

one_grr_result_tbl %>% kable(align = "c")
```

sim_number	variable	st_dev_estimate	variance_estimate
1	part:operator	0.4484307	0.2010901
1	part	11.4718335	131.6029628
1	operator	1.6062900	2.5801677
1	Residual	3.9540576	15.6345714

Recall that the true parameters for sd were 1, 9, 2, and 4. These estimates aren't dead on but we don't really understand how much uncertainty is involved with our estimate. We'll return to that in a moment.

One common performance metric for a Gage R&R is %tolerance: 6 (or some other constant) times the sum of the measurement system variance (everything except for part variance) divided by the tolerance span for this particular measurement.

\[\ Percent\space Tolerance = 6 \times (\hat\sigma_E^2 + \hat\sigma_O^2 + \hat\sigma_{PO}^2)\space/\space Tolerance \]

It is common for the tolerance span to be approximately 6 x the standard deviation of the parts population. Let's make that assumption here so we can estimate the percent tolerance from the data and compare to the true

value. Once the simulation is established, the standard deviation of the parts can be adjusted to see how the percent tolerance changes for a given sd of operators, part:operator interaction, and residual.

```
one_grr_tol_outcome_tbl <- one_grr_result_tbl %>%
  filter(variable != "part") %>%
  group_by(sim_number) %>%
  summarize(grr_variance_est = sum(variance_estimate)) %>%
  mutate(true_tol_pct = scales::percent((int_intercepts_sd^2 + oper_intercepts_sd^2
+ random_error_repeatability^2) / part_intercepts_sd)) %>%
  rowwise() %>%
  mutate(est_tol_pct = scales::percent(grr_variance_est / part_intercepts_sd))

one_grr_tol_outcome_tbl %>% kable(align = "c")
```

sim_number	grr_variance_est	true_tol_pct	est_tol_pct
1	18.41583	233%	205%

Since 205% is > 30%, this experiment would "fail" and the measurement system would not be validated. Not really interesting since we just chose arbitrary numbers but the estimate is reasonably close to the true value which is more important. It would be good to know that if we repeated the simulation a lot of times, the average estimate will converge near the true value.

Scale Simulation with a Function

If want to simulate a lot of Gage R&R's, we can take all the code chunks above and wrap them in a function and then just swap out the values we want to adjust for argument in the function. The function below will take:

- n = number of simulations
- np = number of parts
- no = number of operators
- nm = number of measurements per operator
- iisd = interaction intercepts standard deviation
- oisd = operator intercepts standard deviation
- pisd = part intercepts standard deviation
- rer = random error (repeatability) standard deviation

I'm pretty sure this operation could be done faster and cleaner than the code shown below, but I like how the code maps to the single case simulation above for easy human readability. For this reason, I use a for loop instead of some map() variant.

```
grr_fct <- function(n, np, no, nm, iisd, oisd, pisd, rer) {
  all_grr_results_tbl <- NULL # tibble to hold results
  n_sims <- n # number of simulations
  n_part <- np # number of parts
  n_oper <- no # number of opers
  n_measurements <- nm # number of replications

  int_intercepts_sd <- iisd # standard dev of interaction random effects
  oper_intercepts_sd <- oisd # standard dev of operator random effects
  part_intercepts_sd <- pisd # standard dev of operator random effects
  random_error_repeatability <- rer # standard dev of random error (repeatability)

  # assign names to each per, operator, trial
  part <- str_glue("part_{1:n_part}") %>% as_factor()
  operator <- str_glue("oper_{1:n_oper}") %>% as_factor()
  measurement <- str_glue("measurment_{1:n_measurements}") %>% as_factor()

  n_matrix <- n_part * n_oper * n_measurements # number of observations in the
study
```

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

    # generate experimental designa and outcomes for dummy study
    grr_dummy_tbl <- crossing(part, operator, measurement) %>%
      mutate(measurement = 10 + rnorm(n_matrix))

    # fit model for dummy study
    m1 <- lmer(measurement ~ (1 | part) + (1 | operator) + (1 | part:operator),
data = grr_dummy_tbl)

    # extract design matrix Z from dummy model
    design_matrix_Z <- getME(m1, "Z") %>% as.matrix()

    # simulate random effects using input params for sd
    int_intercepts <- rnorm(n = n_part * n_oper, mean = 0, sd = int_intercepts_sd)
    oper_intercepts <- rnorm(n = n_oper, mean = 0, sd = oper_intercepts_sd)
    part_intercepts <- rnorm(n = n_part, mean = 0, sd = part_intercepts_sd)

    # vector of all random effect intercepts (order matters here: ineraction, oper,
part)
    random_effects_intercepts <- c(int_intercepts, part_intercepts,
oper_intercepts)

    # create observations (add in repeatability random error to each term). %*% is
matrix multiplication
    grr_sim_tbl <- grr_dummy_tbl %>%
      mutate(measurement = 10 + design_matrix_Z %*% random_effects_intercepts +
rnorm(
        n = nrow(grr_dummy_tbl),
        mean = 0,
        sd = random_error_repeatability
      ))

    # fit a model to the simulated dataset
    sim_m_fit <- lmer(measurement ~ (1 | part) + (1 | operator) + (1 |
part:operator), data = grr_sim_tbl)

    # tibble of results for a single simulation
    one_grr_result_tbl <-
      broom.mixed::tidy(sim_m_fit, effects = "ran_pars") %>%
      rename(
        st_dev_estimate = estimate,
        variable = group
      ) %>%
      mutate(
        variance_estimate = st_dev_estimate^2,
        sim_number = i
      ) %>%
      select(sim_number, variable, st_dev_estimate, variance_estimate)

    # append this recent simulation to the others
    all_grr_results_tbl <- bind_rows(all_grr_results_tbl, one_grr_result_tbl)
  }
  return(all_grr_results_tbl)
}
```

Test the function by calling it once, asking for just 3 simulations:

```
fcn_test_tbl <- grr_fct(n = 3, np = 10, no = 3, nm = 2, iisd = 4, oisd = 3, pisd = 2, rer = 1)

fcn_test_tbl %>% kable(align = "c")
```

sim_number	variable	st_dev_estimate	variance_estimate
1	part:operator	3.4214284	11.7061726
1	part	2.1832882	4.7667475
1	operator	1.6916881	2.8618086
1	Residual	1.0355244	1.0723107
2	part:operator	4.7726138	22.7778428
2	part	0.0010317	0.0000011
2	operator	0.0000000	0.0000000
2	Residual	1.0763577	1.1585460
3	part:operator	4.0704728	16.5687489
3	part	1.2724219	1.6190574
3	operator	3.2107023	10.3086089
3	Residual	0.9178515	0.8424514

Everything is looking good! The results from n=3 simulations have completed and are summarized nicely in the results tbl.

Rather than manually call the function and input the arguments every time, we can populate a "setup tbl" that contains all the arguments that we will want to look at. Within the tbl we can look at anything we want. For example, we might want several different values for number of operators, or several different levels of standard deviation for one of the random effects. In this case, I was interested in several different magnitudes of standard deviation for the population of parts because the part sd is used as a surrogate for the tolerance percentage calculation as shown above. This is a useful simulation because we should be able to visualize:

- Does the average of the simulation converge near the true value for percent tolerance?
- How often might individual estimates of pct tol "fail" (> 30%) when the average of the estimates passes (< 30%)?

Fist, the setup_tbl. Note: the variance of the random variables for operator, repeatability(residual), and part:operator interaction are all held at 1 while the variance for part is increased.

```
sim_setup_tbl <- tibble(
  n_sims = 200,
  n_part = 10,
  n_oper = 3,
  n_meas = 2,
  int_sd = 1,
  oper_sd = 1,
  #  part_var = 1,
  part_var = c(2^(0:10)),
  repeatab_sd = 1
) %>%
  mutate(
    row_id = row_number()
  ) %>%
  rowwise() %>%
  mutate(part_sd = part_var^.5) %>%
  mutate(tol_pct_true = 6 * (int_sd^2 + oper_sd^2 + repeatab_sd^2) / (6 * part_sd))

sim_setup_tbl %>% kable(align = "c")
```

n_sims	n_part	n_oper	n_meas	int_sd	oper_sd	part_var	repeatab_sd	row_id	part_sd	tol_pct_true
200	10	3	2	1	1	1	1	1	1.000000	3.0000000
200	10	3	2	1	1	2	1	2	1.414214	2.1213203
200	10	3	2	1	1	4	1	3	2.000000	1.5000000
200	10	3	2	1	1	8	1	4	2.828427	1.0606602
200	10	3	2	1	1	16	1	5	4.000000	0.7500000
200	10	3	2	1	1	32	1	6	5.656854	0.5303301
200	10	3	2	1	1	64	1	7	8.000000	0.3750000
200	10	3	2	1	1	128	1	8	11.313709	0.2651650
200	10	3	2	1	1	256	1	9	16.000000	0.1875000
200	10	3	2	1	1	512	1	10	22.627417	0.1325825
200	10	3	2	1	1	1024	1	11	32.000000	0.0937500

The following code executes many simulations, one for each set of arguments from the rows above. Each simulation from each row results in a tibble of outcomes which is stored in a list column and the unnested later on to make a big tibble. I'm not actually 100% sure that you need rowwise() here – it works fine with it in place and makes sense to me that you would group by rows here but I'm still trying to figure out what row-based workflow works best for me. I believe there are other good options that use the map family.

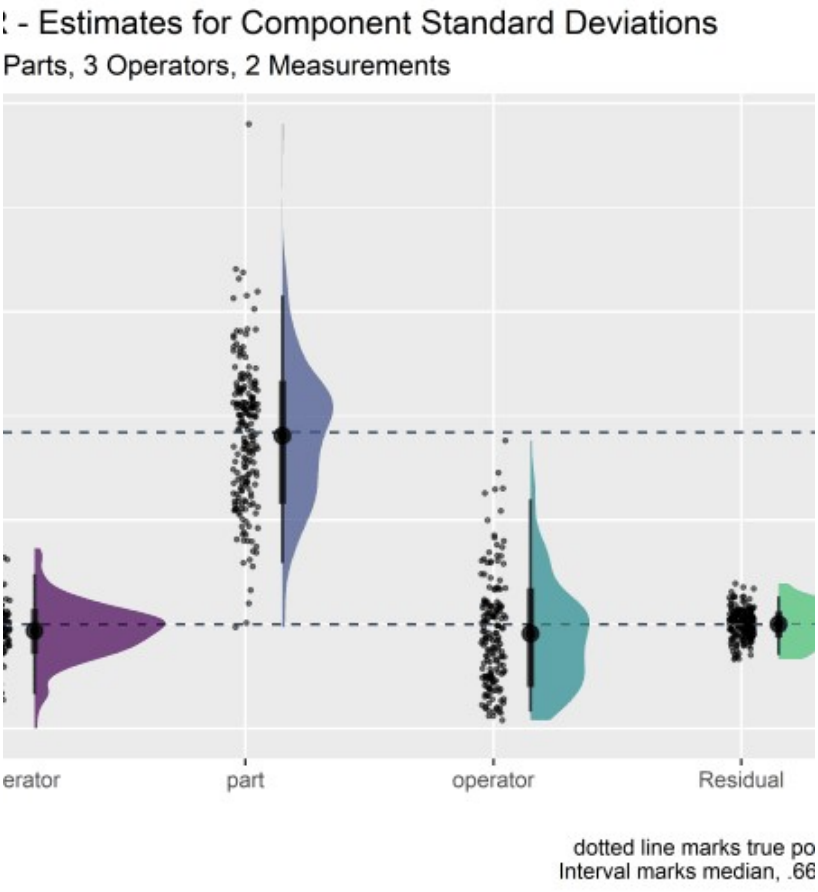
```
set.seed(0118)

#commented out because this takes a while to run

# sim_outcomes_tbl <- sim_setup_tbl %>%
#   rowwise() %>% # may not be needed
#   mutate(sim_outcomes = list(grr_fct(n = n_sims, np = n_part, no = n_oper, nm =
n_meas, iisd = int_sd, oisd = oper_sd, pisd = part_sd, rer = repeatab_sd))) %>%
#   select(sim_outcomes, everything()) %>%
#   unnest(cols = c(sim_outcomes)) %>%
#   mutate_if(is.character, as_factor)

sim_outcomes_tbl %>%
  select(sim_number, variable, st_dev_estimate, n_sims, n_part, n_meas, int_sd,
oper_sd, repeatab_sd, part_sd) %>%
  head(12) %>%
  kable(align = "c")
```

sim_number	variable	st_dev_estimate	n_sims	n_part	n_meas	int_sd	oper_sd	repeatab_sd	part_sd
1	part:operator	0.5511238	200	10	2	1	1	1	1
1	part	0.8165034	200	10	2	1	1	1	1
1	operator	0.5369841	200	10	2	1	1	1	1
1	Residual	1.0638795	200	10	2	1	1	1	1
2	part:operator	0.8046511	200	10	2	1	1	1	1
2	part	1.1717770	200	10	2	1	1	1	1
2	operator	0.9339445	200	10	2	1	1	1	1
2	Residual	0.9631640	200	10	2	1	1	1	1
3	part:operator	1.0850250	200	10	2	1	1	1	1
3	part	0.0031052	200	10	2	1	1	1	1
3	operator	1.7490297	200	10	2	1	1	1	1
3	Residual	0.9284068	200	10	2	1	1	1	1

We might look at the results for a single group of arguments in the setup tbl by filtering to a specific row (I choose row 4 arbitrarily here). Note that "row_id" is a little misleading here because the data has been reshaped. Filtering fora specific row_id returns results from the set of simulations from a single row of parameters in the original

setup_tbl. Here we look at the results form row_4 where the true part sd was 2.82 while the sd for operator, Residual, and interaction were all 1.)

```
a <- sim_outcomes_tbl %>%
  filter(row_id == 4) %>%
  filter(st_dev_estimate > .001) %>%
  ggplot(aes(x = variable, y = st_dev_estimate)) +
  geom_jitter(width = .05, alpha = .5, size = .6) +
  geom_hline(yintercept = 1, lty = 2, color = "#2c3e50") +
  geom_hline(yintercept = 2.8428472, lty = 2, color = "#2c3e50") +
  #    stat_summary(fun.y= mean, fun.ymin=mean, fun.ymax=mean, geom="crossbar",
width=0.2, color="red") +
  stat_halfeye(aes(fill = variable), point_interval = mean_qi, alpha = .7, position
= position_nudge(x = .15)) +
  labs(
    title = "Gage R&R - Estimates for Component Standard Deviations",
    subtitle = str_glue("Settings: {sim_outcomes_tbl$n_part[1]} Parts,
{sim_outcomes_tbl$n_oper[1]} Operators, {sim_outcomes_tbl$n_meas[1]}
Measurements"),
    x = "",
    y = "Standard Deviation Estimate",
    caption = "dotted line marks true population standard dev\n Interval marks
median, .66 quantile, .95 quantile"
  ) +
  theme(legend.position = "none") +
  scale_fill_viridis_d(option = "c", end = .7)

a
```



Looks good! The averages of the set of simulations is converging nicely and we get a good feel for how much uncertainty would be expected for a single trial.

Now to plot multiple simulations. A bit of data preparation is required to plot the true values on the same canvas as the individual sims. There is probably a cleaner way to do this directly within ggplot – but the way I do it here is to pull the values from the rows down into tidy format with pivot_longer and then do some grouping and joining.

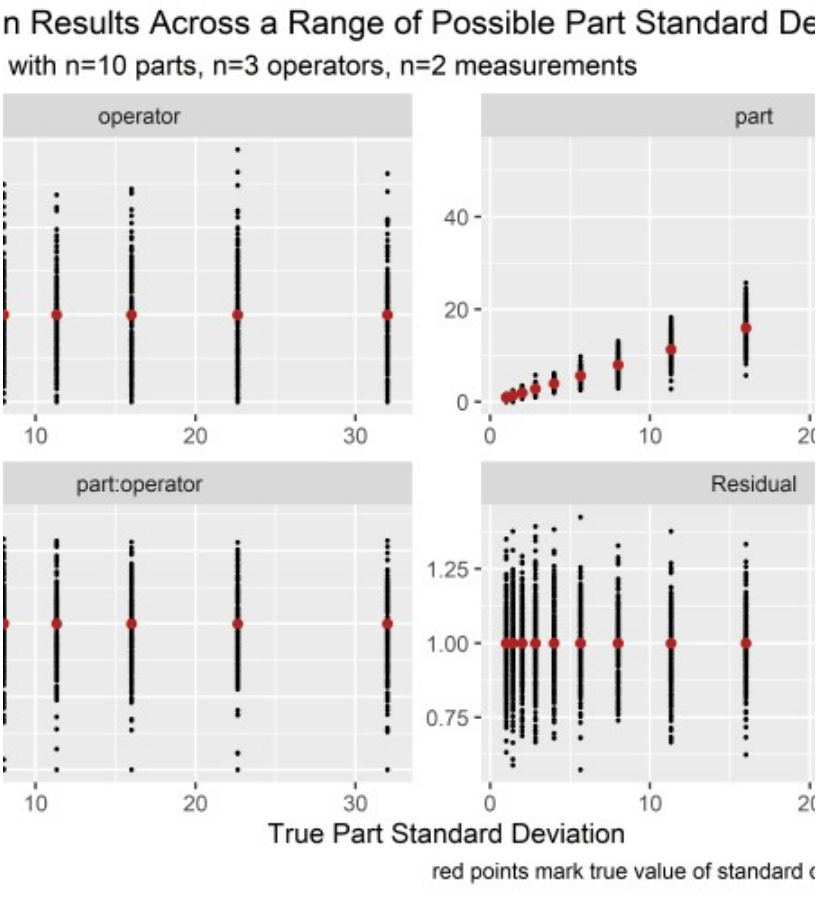
```
sim_tbl <- sim_outcomes_tbl %>% select(sim_number, row_id)

t <- sim_outcomes_tbl %>%
  select(sim_number, variable, row_id, int_sd, oper_sd, part_sd, repeatab_sd,
st_dev_estimate) %>%
  pivot_longer(cols = c(int_sd, oper_sd, part_sd, repeatab_sd)) %>%
  right_join(sim_tbl) %>%
  group_by(variable, name, value, row_id) %>%
  count() %>%
  ungroup() %>%
  mutate(variable = case_when(
    name == "int_sd" ~ "part:operator",
    name == "oper_sd" ~ "operator",
    name == "part_sd" ~ "part",
    TRUE ~ "Residual"
  )) %>%
  right_join(sim_setup_tbl)

t %>%
  head(10) %>%
  select(variable, value, row_id, n_sims, n_part, n_oper, n_meas, int_sd, oper_sd,
part_sd, repeatab_sd, tol_pct_true) %>%
  kable(align = "c")
```

variable	value	row_id	n_sims	n_part	n_oper	n_meas	int_sd	oper_sd	part_sd	repeatab_sd	tol_pct_true
part:operator	1	1	200	10	3	2	1	1	1	1	3
operator	1	1	200	10	3	2	1	1	1	1	3
part	1	1	200	10	3	2	1	1	1	1	3
Residual	1	1	200	10	3	2	1	1	1	1	3
part:operator	1	1	200	10	3	2	1	1	1	1	3
operator	1	1	200	10	3	2	1	1	1	1	3
part	1	1	200	10	3	2	1	1	1	1	3
Residual	1	1	200	10	3	2	1	1	1	1	3
part:operator	1	1	200	10	3	2	1	1	1	1	3
operator	1	1	200	10	3	2	1	1	1	1	3

It would be cool to see all the data from all the simulations. Recall that the true values of sd for operator, part:operator interaction, and Residual were 1 and that part variation increased across the different sims. All tof this is plotted below, with the true values in red while the results of the simulation shown in black. It can be seen that the simulations group nicely around the true values and we can see the uncertainty.

```
sim_outcomes_tbl %>%
  ggplot(aes(x = part_sd, y = st_dev_estimate)) +
  geom_point(size = .4) +
  geom_point(data = t, aes(x = part_sd, y = value), color = "firebrick") +
  facet_wrap(~variable, scales = "free") +
  labs(
    title = "Simulation Results Across a Range of Possible Part Standard
Deviations",
    subtitle = "Gage R&R with n=10 parts, n=3 operators, n=2 measurements",
    x = "True Part Standard Deviation",
    y = "Estimate of Standard Deviation",
```

```
    caption = "red points mark true value of standard deviation for the effect"
  )
```



Let's see how the estimated percent tolerance lines up with the true values from the population. This tbl calculates the percent tol across all the simulations.

```
tol_outcomes_tbl <- sim_outcomes_tbl %>%
  filter(variable != "part") %>%
  group_by(sim_number, row_id) %>%
  summarize(grr_variance_est = sum(variance_estimate)) %>%
  right_join(sim_setup_tbl) %>%
  rowwise() %>%
  mutate(est_tol_pct = (grr_variance_est) / (part_sd)) %>%
  select(n_part, n_oper, n_meas, int_sd, oper_sd, part_sd, tol_pct_true,
est_tol_pct, everything())

tol_outcomes_tbl %>%
  select(n_part, n_oper, n_meas, int_sd, oper_sd, part_sd, tol_pct_true,
est_tol_pct, sim_number, row_id) %>%
  head(10) %>%
  kable(align = "c")
```

n_part	n_oper	n_meas	int_sd	oper_sd	part_sd	tol_pct_true	est_tol_pct	sim_number	row_id
10	3	2	1	1	1	3	1.723929	1	1
10	3	2	1	1	1	3	2.447401	2	1
10	3	2	1	1	1	3	5.098323	3	1
10	3	2	1	1	1	3	1.882222	4	1
10	3	2	1	1	1	3	1.702724	5	1
10	3	2	1	1	1	3	2.381758	6	1
10	3	2	1	1	1	3	1.619832	7	1
10	3	2	1	1	1	3	2.986048	8	1

n_part	n_oper	n_meas	int_sd	oper_sd	part_sd	tol_pct_true	est_tol_pct	sim_number	row_id
10	3	2	1	1	1	3	2.440299	9	1
10	3	2	1	1	1	3	2.441025	10	1

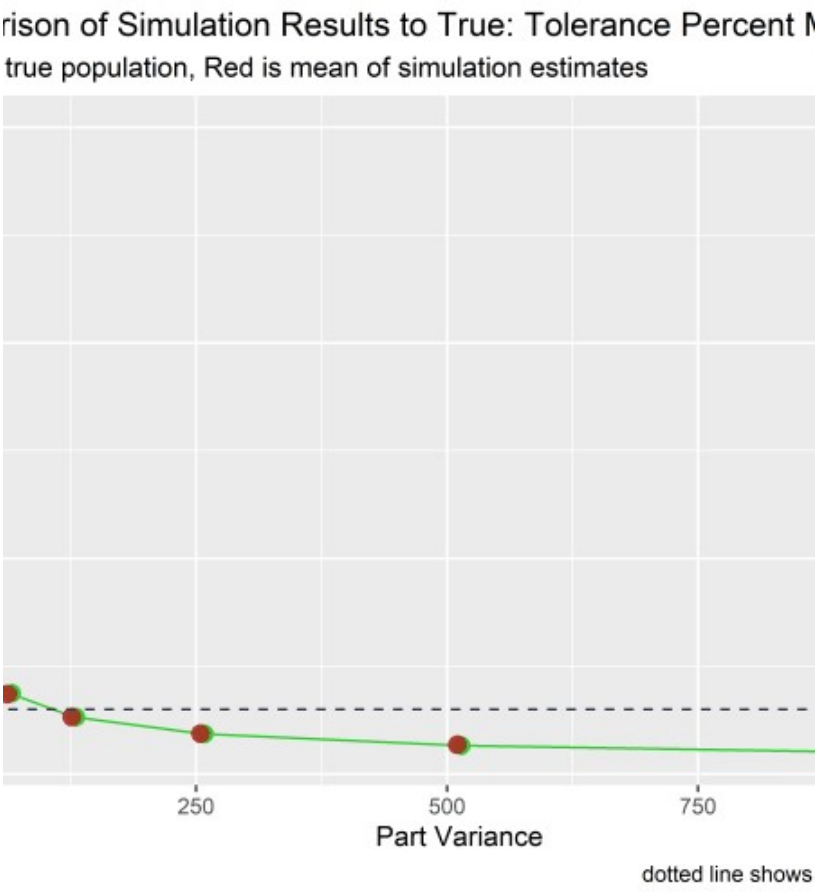
This summary tbl captures the mean from simulations conducted at each level.

```
tol_est_tbl <- tol_outcomes_tbl %>%
  group_by(row_id, part_var) %>%
  summarize(mean_est_tol_pct = mean(est_tol_pct))

tol_est_tbl %>% kable(align = "c")
```

row_id	part_var	mean_est_tol_pct
1	1	2.9902181
2	2	2.1352549
3	4	1.5002527
4	8	1.0354094
5	16	0.7362949
6	32	0.5330749
7	64	0.3708208
8	128	0.2657111
9	256	0.1880981
10	512	0.1372343
11	1024	0.0937739

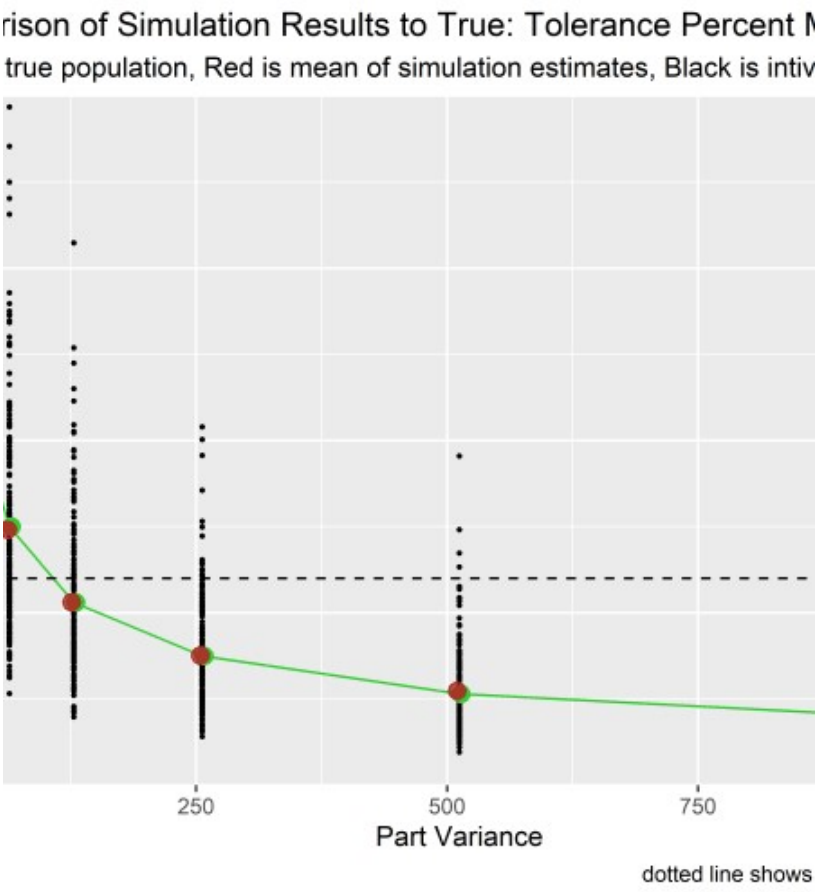
Visualize the mean from the simulations vs. the true tol percent:

```
tol_outcomes_tbl %>%
  ggplot(aes(x = part_var, y = est_tol_pct)) +
  #  geom_point(size = .5) +
  geom_point(aes(x = part_var, y = tol_pct_true), size = 3, color = "limegreen",
position = position_nudge(2), alpha = .85) +
  geom_line(aes(x = part_var, y = tol_pct_true), color = "limegreen") +
  geom_point(data = tol_est_tbl, aes(x = part_var, y = mean_est_tol_pct), size = 3,
color = "firebrick", position = position_nudge(-2), alpha = .85) +
  geom_hline(yintercept = .3, lty = 2, color = "#2c3e50") +
  scale_y_continuous(labels = scales::percent) +
  labs(
    title = "Comparison of Simulation Results to True: Tolerance Percent Metric",
    subtitle = "Green is true population, Red is mean of simulation estimates",
    x = "Part Variance",
    y = "Tolerance Percent",
    caption = "dotted line shows 30% acceptance limit"
  )
```



Very nice agreement between the simulations and the true population values. But the above plot just shows the mean. How much uncertainty is expected across experiments at the same settings? Here I show a subset of data below the 100% level (just to keep the plot a little less messy).

```
tol_outcomes_tbl %>%
  ggplot(aes(x = part_var, y = est_tol_pct)) +
  geom_point(size = .5) +
  geom_point(aes(x = part_var, y = tol_pct_true), size = 3, color = "limegreen",
position = position_nudge(2), alpha = .85) +
  geom_line(aes(x = part_var, y = tol_pct_true), color = "limegreen") +
  geom_point(data = tol_est_tbl, aes(x = part_var, y = mean_est_tol_pct), size = 3,
color = "firebrick", position = position_nudge(-2), alpha = .85) +
  geom_hline(yintercept = .3, lty = 2) +
  scale_y_continuous(
    labels = scales::percent,
    expand = expansion(),
    limits = c(0, 1)
  ) +
  labs(
    title = "Comparison of Simulation Results to True: Tolerance Percent Metric",
    subtitle = "Green is true population, Red is mean of simulation estimates,
Black is intividual estimates",
    x = "Part Variance",
    y = "Tolerance Percent",
    caption = "dotted line shows 30% acceptance limit"
  )
```



Pretty interesting. There is quite a bit of uncertainty here and there are quite a few cases where individual simulations fail just due to chance. The probability of this happening is worst when the part variance is around 100-400% of the operator and repeatability sd.

From this point one could start tweaking any values of interest to see how the uncertainty of the performance metric is affected. For example, you might look at the optimal number of operators or replicates to give the best chance of passing when the true population value would pass. Pretty powerful stuff!