General structure of a simulation study

Using the example of estimating the statistical power of Student's independent *t*-test

Ian Hussey

02 Mai, 2024

Contents

Overview of tutorial

This tutorial teaches you about the essential components of a simulation study and the general steps involved in actually writing one. It does this using the example of a simulation that answers a question that is relevant to many social scientists:

The independent t-test assumes that participants are allocated equally to two groups. How much does an unbalanced design lower the statistical power of the test?

We will somewhat gloss over some important details, e.g., the specifics of how data is generated using pseudo random number generators, a more fine-grained treatment of how we write functions, and diagnostics of whether your simulation is performing well. We will return to these in future lessons.

Essential components & general steps

The essential *components* of a simulation are:

1. Generate pseudo-random data set with known properties
2. Analyse data with a statistical method
3. Repeat 1 & 2 many times ('iterations')
4. Collect and aggregate results across iterations
5. Make it an experiment: Systematically vary parameters in Step 1 (between factor) and/or compare different ways to do Step 2 (within factor)

However, the way you *build* a simulation is much more iterative than this final product. Each component must be both built and inspected by itself, and then checked to make sure that the components work together in the correct manner, like a fine Swiss watch.

The general steps for actually building one involve:

1. Generate a *single* pseudo-random data set with known properties, using hard-coded variables
2. Analyse this *single* data set with a statistical method, using hard-coded variables
3. Convert the data generation code to a function, making it as abstract as necessary [Component 1]
4. Convert the data analysis code to a function, making it as abstract as necessary [Component 2]
5. Ensure that experiment parameters, data generation function, and data analysis code play together nicely (i.e., can pass values between one another in a workflow, and save values appropriately [Component 4]). Do this using a small number of parameters and just one or two iterations.
6. Increase the number of iterations and parameters and actually run the simulation as a whole [Component 4 and 5]
7. Check the assumptions of the simulation have been adequately met
8. Interpret the results of the simulation

This tutorial focuses on the general practical steps, but always keep the essential components in mind as the guiding principles of why we're doing something.

Generate data and apply and analysis *once* using hard coded arguments

Generate some data for an independent t-test

condition: factor with two levels (control, intervention) score: numeric of normally distributed data (within each condition), with different means, and SD = 1.

To sample data from a normally distributed population, we use the pseudo-random number generator for normal data built into R: `rnorm()`. Note that many packages exist to help you generate different types of data more easily, including {MASS}, {faux}, {simstudy}, and {lavaan}. These can be very helpful, but here we'll do it manually for the moment.

```r
# dependencies
library(tidyr)
library(dplyr)
```

```
##
## Attache Paket: 'dplyr'

## Die folgenden Objekte sind maskiert von 'package:stats':
##
##     filter, lag

## Die folgenden Objekte sind maskiert von 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
library(forcats)
library(readr)
library(purrr)
library(ggplot2)
library(effsize)

# set the seed value for pseudo random number generators to make results reproducible
set.seed(42)

generated_data <-
  bind_rows(
    # given that cohen's d = (m2 - m1)/SD_pooled, simply setting SDs to 1 and mean_control to 0 lets us
    tibble(condition = "control",
           score = rnorm(n = 100, mean = 0.0, sd = 1)),
    tibble(condition = "intervention",
           score = rnorm(n = 100, mean = 0.5, sd = 1))
  )

head(generated_data)
```

```
## # A tibble: 6 x 2
##   condition  score
##   <chr>      <dbl>
## 1 control    1.37
## 2 control   -0.565
## 3 control    0.363
## 4 control    0.633
## 5 control    0.404
## 6 control   -0.106
```

Fit a Student's independent *t*-test

```r
t.test(formula = score ~ condition,
       data = generated_data,
       var.equal = TRUE,
       alternative = "two.sided")
```

```
##
##  Two Sample t-test
##
```

```
## data:  score by condition
## t = -2.7554, df = 198, p-value = 0.006409
## alternative hypothesis: true difference in means between group control and group intervention is not
## 95 percent confidence interval:
##  -0.6519651 -0.1080379
## sample estimates:
##      mean in group control mean in group intervention
##                 0.03251482                 0.41251629
```

Create a data generating function

Make the existing code more abstract

Move the values to variables

```r
# parameters for simulated data
n_control = 50
n_intervention = n_control
mean_control = 0
mean_intervention = 0.5
sd_control = 1
sd_intervention = 1

# generate data using these parameters
generated_data <-
  bind_rows(
    tibble(condition = "control",
           score = rnorm(n = n_control, mean = mean_control, sd = sd_control)),
    tibble(condition = "intervention",
           score = rnorm(n = n_intervention, mean = mean_intervention, sd = sd_intervention))
  )

# view the generated data
head(generated_data)
```

```
## # A tibble: 6 x 2
##   condition  score
##   <chr>      <dbl>
## 1 control   -2.00
## 2 control    0.334
## 3 control    1.17
## 4 control    2.06
## 5 control   -1.38
## 6 control   -1.15
```

Convert this to a function

Abstracting code into functions is a learned skill. For the moment, see if you can follow the logic of the function and where it came from in the hard coded original version.

```r
# define function
generate_data <- function(n_control, # the parameters are now function arguments
                          n_intervention,
                          mean_control,
                          mean_intervention,
                          sd_control,
                          sd_intervention) {

  data <-
    bind_rows(
      tibble(condition = "control",
             score = rnorm(n = n_control, mean = mean_control, sd = sd_control)),
      tibble(condition = "intervention",
             score = rnorm(n = n_intervention, mean = mean_intervention, sd = sd_intervention))
    )

  return(data)
}

# call the function with example arguments
generated_data <- generate_data(n_control = 50,
                                n_intervention = 50,
                                mean_control = 0,
                                mean_intervention = 0.5,
                                sd_control = 1,
                                sd_intervention = 1)

# view the generated data
head(generated_data)
```

```
## # A tibble: 6 x 2
##   condition    score
##   <chr>        <dbl>
## 1 control   -0.00462
## 2 control    0.760
## 3 control    0.0390
## 4 control    0.735
## 5 control   -0.146
## 6 control   -0.0579
```

Check the data is what you meant to data

In general, you should inspect the data thoroughly, check the data types, plot it, conduct other sanity checks, etc. For the moment, we apply just the most basic test: check that the analysis can be fit to data generated by the data generation function. For the moment, the specific results don't matter as much as whether it can accept the data. That is, your data generation will have to be aligned with your data analysis code (e.g., both make use of the variables score [continuous] and condition [factor with two levels]).

```
t.test(formula = score ~ condition,
       data = generated_data, # using the data generated in the previous chunk
       var.equal = TRUE,
       alternative = "two.sided")
```

```
##
##  Two Sample t-test
##
## data:  score by condition
## t = -3.9348, df = 98, p-value = 0.0001557
## alternative hypothesis: true difference in means between group control and group intervention is not
## 95 percent confidence interval:
```

```
##  -1.0364241 -0.3414872
## sample estimates:
##      mean in group control mean in group intervention
##                -0.06154135                 0.62741428
```

Create a data analysis function

Make the existing code more abstract

Now do the same abstraction for the analysis.

Rather than just printing all the results of the t test to the console, extract the p value specifically as a column in a tibble. In general, you usually want to extract the results of analyses in `tidy data` format. Note that the {parameters} and {broom} packages can be very helpful for doing this for you for many common types of analysis, but here we're do it manually.

```r
res_t_test <- t.test(formula = score ~ condition,
                     data = generated_data,
                     var.equal = TRUE,
                     alternative = "two.sided")

res <- tibble(p = res_t_test$p.value)

res
```

```
## # A tibble: 1 x 1
##          p
##      <dbl>
## 1 0.000156
```

Convert this to a function

```r
analyse_data <- function(data) {

  res_t_test <- t.test(formula = score ~ condition,
                       data = data,
                       var.equal = TRUE,
                       alternative = "two.sided")

  res <- tibble(p = res_t_test$p.value)

  return(res)
}
```

Compare original hard-coded version with functionalised version

Original manual code, copied from above

```r
set.seed(42)

generated_data <-
  bind_rows(
    tibble(condition = "control",
           score = rnorm(n = 100, mean = 0.0, sd = 1)),
    tibble(condition = "intervention",
           score = rnorm(n = 100, mean = 0.5, sd = 1))
  )

t.test(formula = score ~ condition,
       data = generated_data,
       var.equal = TRUE,
       alternative = "two.sided")
```

```
##
##  Two Sample t-test
##
## data:  score by condition
## t = -2.7554, df = 198, p-value = 0.006409
## alternative hypothesis: true difference in means between group control and group intervention is not
## 95 percent confidence interval:
##  -0.6519651 -0.1080379
## sample estimates:
##      mean in group control mean in group intervention
##                 0.03251482                 0.41251629
```

New code using functions, copied from above

```
set.seed(42)

generated_data <- generate_data(n_control = 50,
                                n_intervention = n_control,
                                mean_control = 0,
                                mean_intervention = 0.5,
                                sd_control = 1,
                                sd_intervention = 1)

results <- analyse_data(generated_data)

results
```

```
## # A tibble: 1 x 1
##         p
##     <dbl>
## 1 0.00297
```

Do it lots of times and summarize across them

```
set.seed(42)

# define the number of iterations
iterations <- 100

# declare a vector to store each iteration's results in
p_values <- numeric(iterations)

# use for loop to repeat its content many times
# for the 'i'-th element of the series 1 to `iterations` (i.e., 1, 2, 3, ... iterations), run the follo
```

```r
for(i in 1:iterations){
  # generate data
  generated_data <- generate_data(n_control = 50,
                                  n_intervention = 50,
                                  mean_control = 0,
                                  mean_intervention = 0.5,
                                  sd_control = 1,
                                  sd_intervention = 1)

  # analyse data
  results <- analyse_data(generated_data)

  # store result of this iteration
  # the i-th element of the iterations vector will be replaced with the p value of the current iteratio
  p_values[i] <- results$p
}

# summarize individual analysis results across iterations to find stimulation results
mean(p_values < .05)
```

```
## [1] 0.77
```

Congratulations, you've just run your first simulation. Let's remind ourselves what it did.

- Data generating process (i.e., population effect): true effect Cohen's d = 0.50 (i.e., difference in means = 0.50, both SDs = 1), with 50 participants per condition.
- Data analysis: independent t-test p value.
- Iterations: 100
- Summary across iterations: For 50 participants per condition, when the true population effect exists (is non zero, i.e., Cohen's d = .50), an independent t-test produces statistically significant p values in 77% of cases. This is the definition of statistical power: the proportion of cases where effects that do exist are detected (i.e., true-positive results).

What if I want my simulation to do more complex things?

The above code isn't enough because we want our simulation to be an experiment with multiple between-conditions. How might we accomplish this?

We could extend the method above, but it gets tricky very quickly.

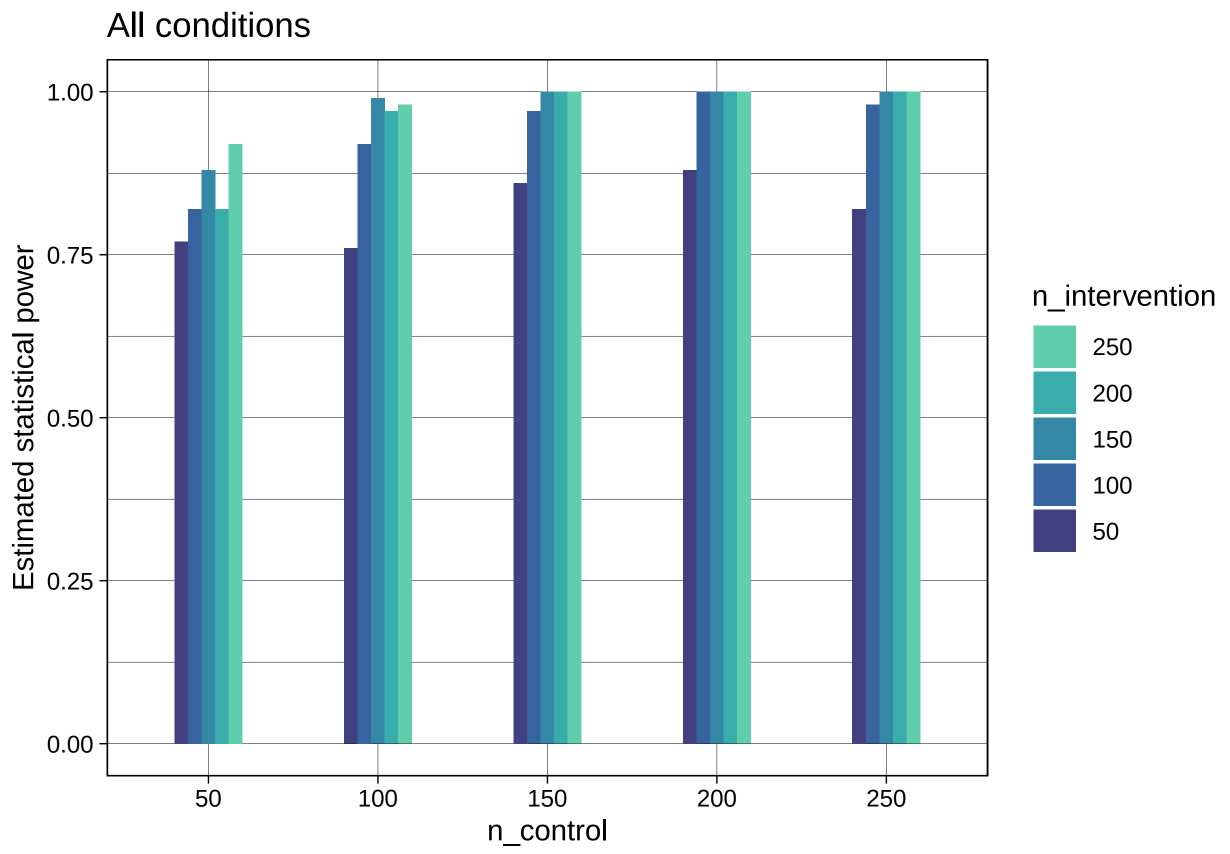
For example, if I wanted to see how changing the number of participants in (a) the control condition and (b) the intervention condition so that either can be anywhere between 50 and 250 (in steps of 50), including having different number between them (e.g., control = 200, intervention = 100), then I'll have to either:

- Repeat my code a lot. This is always a bad idea. Repeating simple tasks we're bad at doing ourselves is the reason we invented computers.
- Abstract the code further in some way, such as using more for-loops.

Before you look at this code, know that it's much harder to understand and much harder to write too. I'm showing you this method so that you know it exists and is the most basic way of doing it - not because we are going to do it in any of the rest of this course.

```r
set.seed(42)

# define the number of iterations
iterations <- 100
```

```r
n_control_conditions <- seq(from = 50, to = 250, by = 50)
n_intervention_conditions <- seq(from = 50, to = 250, by = 50)

# initialize results list
simulation_results <- list()

# counter for appending results
result_counter <- 1

# use nested for loops to iterate over conditions
for(k in n_intervention_conditions){ # for each of the 'k'-th members of the n_intervention_conditions
  for(j in n_control_conditions){ # ... and for each of the 'j'-th members of the n_control_conditions
    for(i in 1:iterations){ # ... and for each value of 'i' in the sequence 1 to iterations, run the fo

      # generate data
      generated_data <- generate_data(n_control = j,  # current value of j
                                      n_intervention = k,  # current value of k
                                      mean_control = 0,
                                      mean_intervention = 0.5,
                                      sd_control = 1,
                                      sd_intervention = 1)

      # analyse data
      results <- analyse_data(generated_data)

      # save results for this iteration as the 'result_counter'-th member of the simulation_results lis
      simulation_results[[result_counter]] <- list(
        n_control = j, # current value of j
        n_intervention = k, # current value of k
        p_value = results$p  # current value the t test's p value
      )

      # increment the iteration counter
      result_counter <- result_counter + 1
    }
  }
}

# convert the list-of-lists into a data frame, as its easier to wrangle and plot
simulation_results_df <- bind_rows(simulation_results)

# summarize individual analysis results across iterations to find stimulation results
# ie plot as a bar plot
simulation_results_df |>
  mutate(n_control = as.factor(n_control),
         n_intervention = as.factor(n_intervention)) |>
  group_by(n_control, n_intervention) |>
  summarize(power = mean(p_value < .05), .groups = "drop") |>
  ggplot(aes(n_control, power, fill = n_intervention)) +
  geom_col(position = position_dodge(width = 0.4), width = 0.4) +
  scale_fill_viridis_d(option = "mako", begin = 0.3, end = 0.8,
                       guide = guide_legend(reverse = TRUE)) +
  theme_linedraw() +
  ggtitle("All conditions") +
  ylab("Estimated statistical power")
```



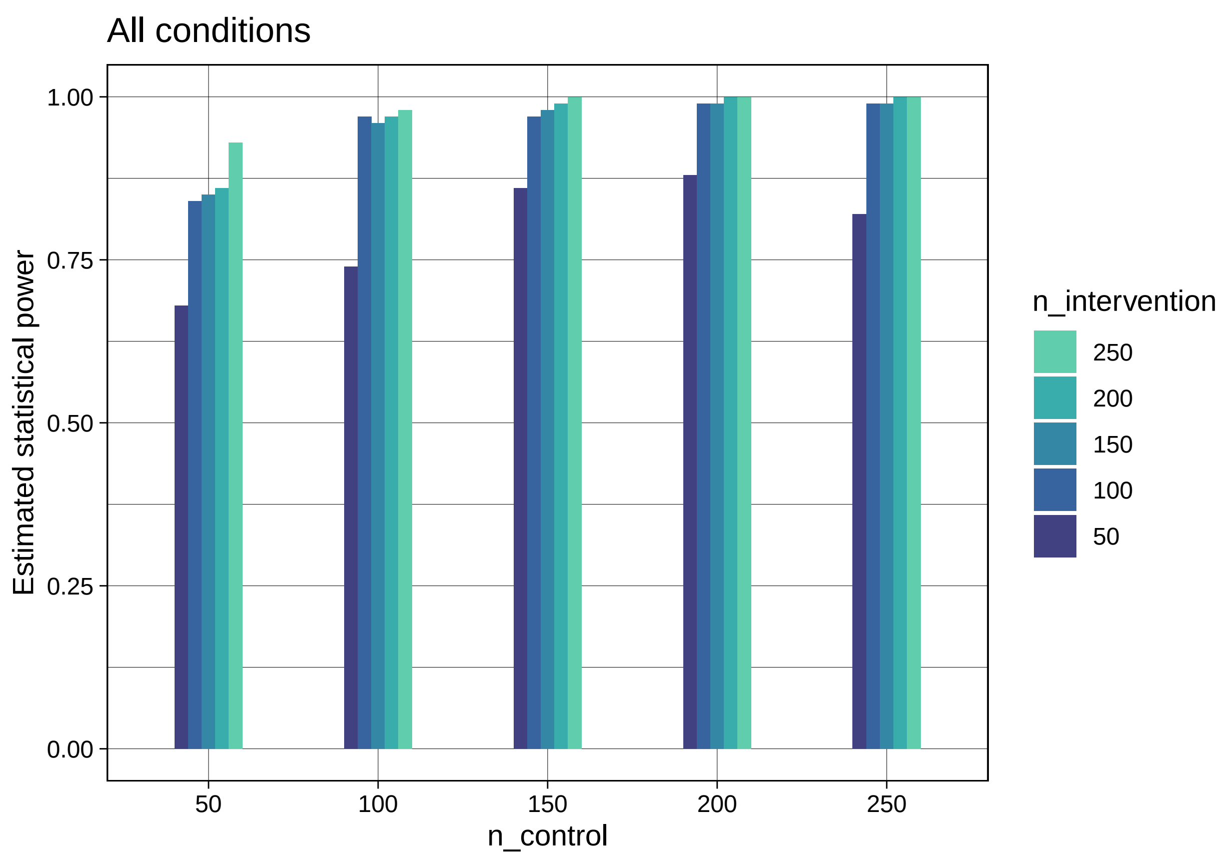
A better way using {dplyr} and {purrr}

There are many other ways to implement simulations, using solutions written in base R, purrr, and other packages such as SimDesign and simhelpers.

My goal here is to teach you an approach that is both flexible and intuitive for those who understand dplyr well already. It prioritizes retaining all objects created within the simulation so that they are visible and checkable, whereas other solutions make intermediate steps invisible and abstract which is often difficult for new learners.

```
# define experiment parameters
# and use expand_grid (the {tidyr} version of expand.grid) to create a data frame of all their permutat
# note that each variable can have just one value or multiple values
experiment_parameters_grid <- expand_grid(
  n_control = seq(from = 50, to = 250, by = 50),
  n_intervention = seq(from = 50, to = 250, by = 50),
  mean_control = 0,
  mean_intervention = 0.5,
  sd_control = 1,
  sd_intervention = 1,
  iteration = 1:100 # note this is a series not an integer, i.e., "1:100" not "100", as "100" would mea
)
```

```r
simulation <-
  # "using the experiment parameters..."
  experiment_parameters_grid |>

  # "...generate data using the data generating function and the parameters relevant to data generation
  # NB pmap() passes a list of parameters to the function listed as its second argument (generate_data)
  # when used inside a mutate call like this, its saying "for each row, create a new column (generated_
  # by calling a function (generate_data) and use the following list of columns as its arguments.
  # ensure the list orders the arguments in the same order as the function takes them, as they are pass
  mutate(generated_data = pmap(list(n_control,
                                    n_intervention,
                                    mean_control,
                                    mean_intervention,
                                    sd_control,
                                    sd_intervention),
                               generate_data)) |>

  # "... then apply the analysis function to the generated data using the parameters relevant to analys
  # again ensure they are listed in the correct order, as they are passed to the function in this order
  mutate(analysis_results = pmap(list(generated_data),
                                 analyse_data))


# "... then summarise results across iterations"
simulation_summary <- simulation |>
  # convert this column in the df that is a nested data frame into columns in the df
  unnest(analysis_results) |>
  # convert these variables into factors for plotting
  mutate(n_control = as.factor(n_control),
         n_intervention = as.factor(n_intervention)) |>
  # for each level of n_control and n_intervention, calculate power (proportion of iterations where sig
  group_by(n_control,
           n_intervention) |>
  summarize(power = mean(p < .05), .groups = "drop")

# plot results
ggplot(simulation_summary, aes(n_control, power, fill = n_intervention)) +
  geom_col(position = position_dodge(width = 0.4), width = 0.4) +
  scale_fill_viridis_d(option = "mako", begin = 0.3, end = 0.8,
                       guide = guide_legend(reverse = TRUE)) +
  theme_linedraw() +
  ggtitle("All conditions") +
  ylab("Estimated statistical power")
```



- Inspect the parameters grid to see how it contains a row for each iteration to be run, containing all the parameters for that iteration.
- View the `simulation` data frame: notice that it saves the data and results from each iteration as a "nested data frame" column. You can click on each cell to open it and see the data frame contained.
- It's really important to understand the pmap() call here. It allows us to effectively run the following code on each row, while avoiding using `rowwise()` because that can be much slower and cause issues later. However, it can help to see it implemented using rowwise() (without pmap()) to understand it. If it helps you to understand it, you can write your own simulations this way for the moment but note that they'll be slower to run.

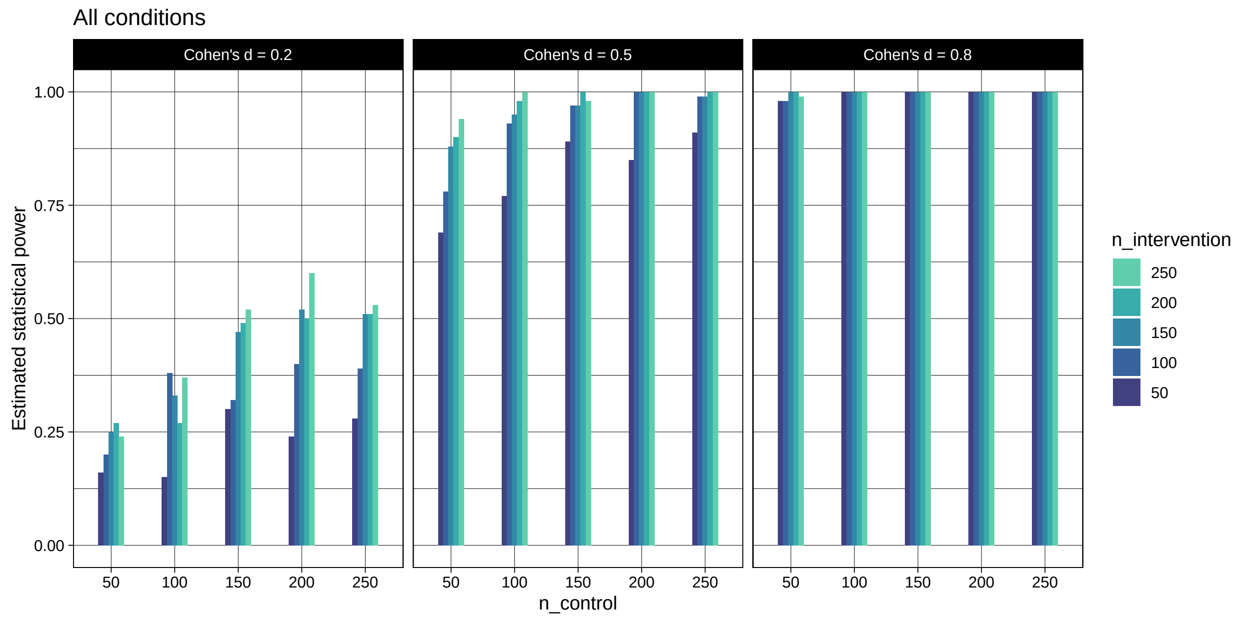
```
simulation_alt <-
  # "using the experiment parameters..."
  experiment_parameters_grid |>

  # "...generate data using the data generating function and the parameters relevant to data generation
  rowwise() |> # without pmap(), you need a rowwise() call to use only the values from this row as inpu
  # note how the output of generate_data() has to be converted to a list() to be saved in a single cell
  mutate(generated_data = list(generate_data(n_control = n_control,
                                             n_intervention = n_intervention,
                                             mean_control = mean_control,
                                             mean_intervention = mean_intervention,
                                             sd_control = sd_control,
                                             sd_intervention = sd_intervention))) |>

  # "... then apply the analysis function to the generated data using the parameters relevant to analys
  mutate(analysis_results = list(analyse_data(data = generated_data))) |>
  ungroup() # once the operations that need to be completed rowwise() are complete you should ungroup()
```

The flexibility of this approach

This approach where we (a) define a grid of all the parameters and iterations we want to run, (b) map our data generation and analysis functions onto each row, saving nested data frames each time, is extremely scalable. By this, I mean that we can examine any other values of these parameters that we like, or indeed change some parameters from being static single parameters to having multiple ones (i.e., experimentally manipulate them), without having to write additional for-loops to handle them.

For example, let's simulate more than one true effect size and see how power changes with true effect size too.

```
# define experiment parameters
experiment_parameters_grid <- expand_grid(
  n_control = seq(from = 50, to = 250, by = 50),
  n_intervention = seq(from = 50, to = 250, by = 50),
  mean_control = 0,
  mean_intervention = c(0.2, 0.5, 0.8), # I've added additional values here for small, medium, & large
  sd_control = 1,
  sd_intervention = 1,
  iteration = 1:100
)

simulation <-
  # using the experiment parameters
  experiment_parameters_grid |>

  # generate data using the data generating function and the parameters relevant to data generation
  mutate(generated_data = pmap(list(n_control,
                                    n_intervention,
                                    mean_control,
                                    mean_intervention,
                                    sd_control,
                                    sd_intervention),
                               generate_data)) |>

  # apply the analysis function to the generated data using the parameters relevant to analysis
  mutate(analysis_results = pmap(list(generated_data),
                                 analyse_data))


# summarise results across iterations
simulation_summary <- simulation |>
  unnest(analysis_results) |>
  mutate(n_control = as.factor(n_control),
         n_intervention = as.factor(n_intervention),
         true_effect = paste("Cohen's d =", mean_intervention)) |>
  group_by(n_control,
           n_intervention,
           true_effect) |>
  summarize(power = mean(p < .05), .groups = "drop")

# plot summary
ggplot(simulation_summary, aes(n_control, power, fill = n_intervention)) +
  geom_col(position = position_dodge(width = 0.4), width = 0.4) +
  scale_fill_viridis_d(option = "mako", begin = 0.3, end = 0.8,
                       guide = guide_legend(reverse = TRUE)) +
  theme_linedraw() +
  ggtitle("All conditions") +
  ylab("Estimated statistical power") +
  facet_wrap(~ true_effect)
```



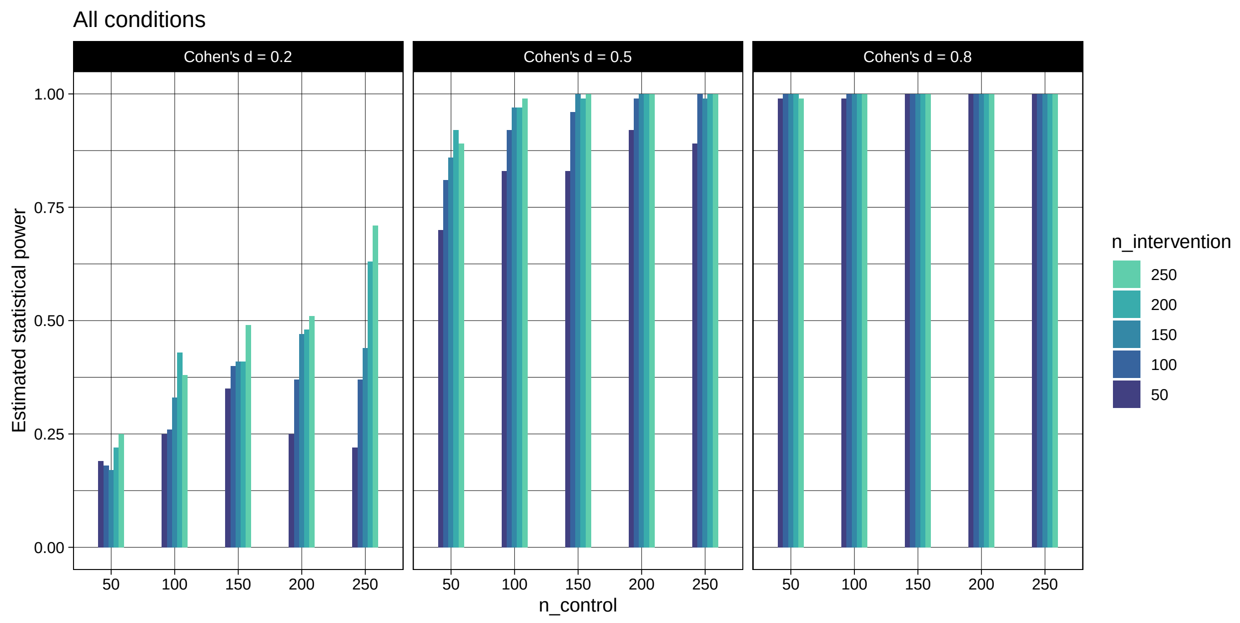
Extending this simulation

We saw above that this approach allows us to easily simulate additional conditions simply by including them in our expand_grid parameters. If we want to simulate additional conditions or collect additional data from, we can also easily modify our data generation or data analysis functions to do this, (often) without having to change other components of the simulation, or without having to change them much (e.g., by adding additional parameters to the expand grid).

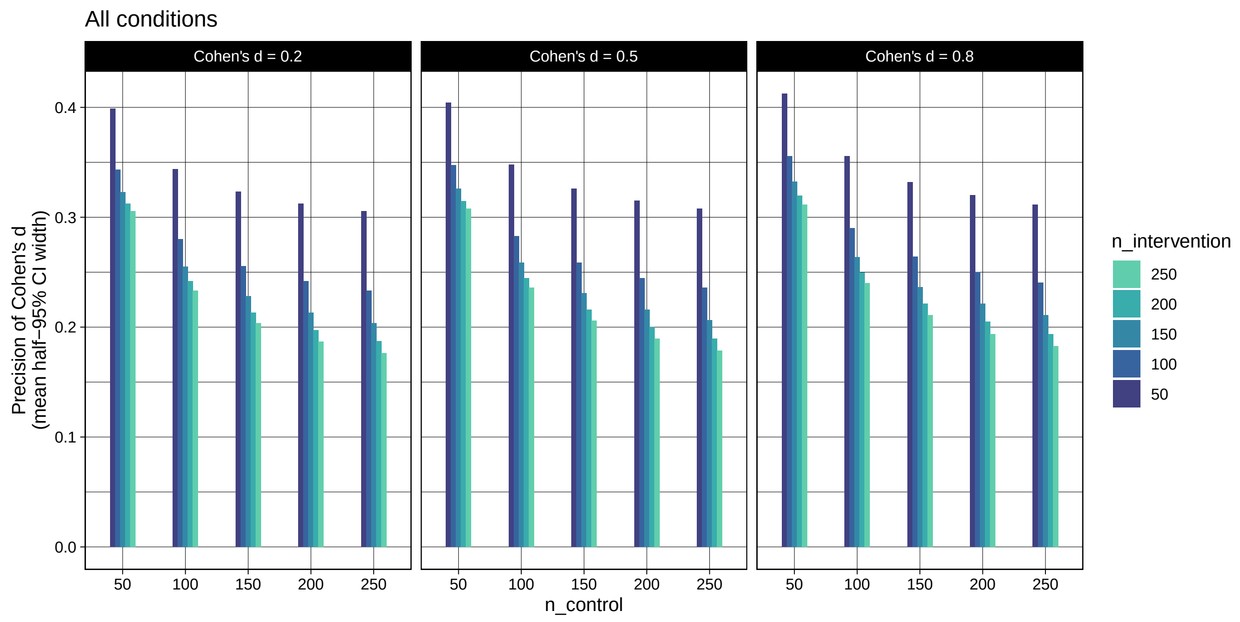
For example, if we want our analysis to save the p values but also the Cohen's d effect sizes, we can modify the function to do this.

```
# redefine the data generation function
generate_data <- function(n_control, # the parameters are now function arguments
                          n_intervention,
                          mean_control,
                          mean_intervention,
                          sd_control,
                          sd_intervention) {

  data <-
    bind_rows(
      tibble(condition = "control",
             score = rnorm(n = n_control, mean = mean_control, sd = sd_control)),
      tibble(condition = "intervention",
             score = rnorm(n = n_intervention, mean = mean_intervention, sd = sd_intervention))
    ) |>
  # new addition: control's factor levels must be ordered so that intervention is the first level and c
```

```r
  # this ensures that positive cohen's d values refer to intervention > control and not the other way a
  mutate(condition = fct_relevel(condition, "intervention", "control"))

  return(data)
}

# redefine the analysis function
analyse_data <- function(data) {
  # dependencies
  require(effsize)

  res_t_test <- t.test(formula = score ~ condition,
                       data = data,
                       var.equal = TRUE,
                       alternative = "two.sided")

  res_cohens_d <- effsize::cohen.d(formula = score ~ condition,  # new addition: also fit cohen's d
                                   within = FALSE,
                                   data = data)

  res <- tibble(p = res_t_test$p.value,
                cohens_d = res_cohens_d$estimate,  # new addition: save cohen's d and its 95% CIs to th
                cohens_d_ci_lower = res_cohens_d$conf.int["lower"],
                cohens_d_ci_upper = res_cohens_d$conf.int["upper"])

  return(res)
}


# the code for running the simulation remains unchanged:

# define experiment parameters
experiment_parameters_grid <- expand_grid(
  n_control = seq(from = 50, to = 250, by = 50),
  n_intervention = seq(from = 50, to = 250, by = 50),
  mean_control = 0,
  mean_intervention = c(0.2, 0.5, 0.8), # small, medium, large Cohen's d
  sd_control = 1,
  sd_intervention = 1,
  iteration = 1:100
)

# run simulation
simulation <-
  # using the experiment parameters
  experiment_parameters_grid |>

  # generate data using the data generating function and the parameters relevant to data generation
  mutate(generated_data = pmap(list(n_control,
                                    n_intervention,
                                    mean_control,
                                    mean_intervention,
                                    sd_control,
                                          sd_intervention),
                                     generate_data)) |>

  # apply the analysis function to the generated data using the parameters relevant to analysis
  mutate(analysis_results = pmap(list(generated_data),
                                       analyse_data))


# summarise across iterations
simulation_summary <- simulation |>
  unnest(analysis_results) |>
  mutate(n_control = as.factor(n_control),
         n_intervention = as.factor(n_intervention),
         true_effect = paste("Cohen's d =", mean_intervention)) |>
  group_by(n_control,
           n_intervention,
           true_effect) |>
  summarize(power = mean(p < .05), # power: proportion of iterations where significant results were fou
            # new: mean half-CI-width. ie with this sample size, how wide are its 95% CIs, ± the point
            mean_cohens_d_precision = mean((cohens_d_ci_upper - cohens_d_ci_lower)/2),
            .groups = "drop")

# plot power
ggplot(simulation_summary, aes(n_control, power, fill = n_intervention)) +
  geom_col(position = position_dodge(width = 0.4), width = 0.4) +
  scale_fill_viridis_d(option = "mako", begin = 0.3, end = 0.8,
                       guide = guide_legend(reverse = TRUE)) +
  theme_linedraw() +
  ggtitle("All conditions") +
  ylab("Estimated statistical power") +
  facet_wrap(~ true_effect)
```



```r
# new plot: precision of cohen's d
ggplot(simulation_summary, aes(n_control, mean_cohens_d_precision, fill = n_intervention)) +
  geom_col(position = position_dodge(width = 0.4), width = 0.4) +
  scale_fill_viridis_d(option = "mako", begin = 0.3, end = 0.8,
                       guide = guide_legend(reverse = TRUE)) +
  theme_linedraw() +
  ggtitle("All conditions") +
  ylab("Precision of Cohen's d\n(mean half-95% CI width)") +
  facet_wrap(~ true_effect)
```



Putting it all together

Now let's bring all the code together and run the simulation for real. We'll improve two things while we do it.

Issue 1

You might have noticed that sometimes the results of the simulations above have strange results between the sample-size conditions. E.g., some conditions with higher sample sizes have slightly lower estimated power. This shouldn't be the case, as high sample sizes = more power. Why might this be the case?

Because, just like in an experiment with human participants, your sample size matters. We have only collected data from 100 iterations, and so the standard errors around each estimate will still be quite large. In the final simulation below, we'll set `iterations` to `1000` so that we have a lot more (simulated) data to base our inferences on. Note that this may take a minute or two to run depending on how fast your computer is. In general, simulations can be computationally intensive to run as you're applying an analysis not once but thousands of times.

Issue 2

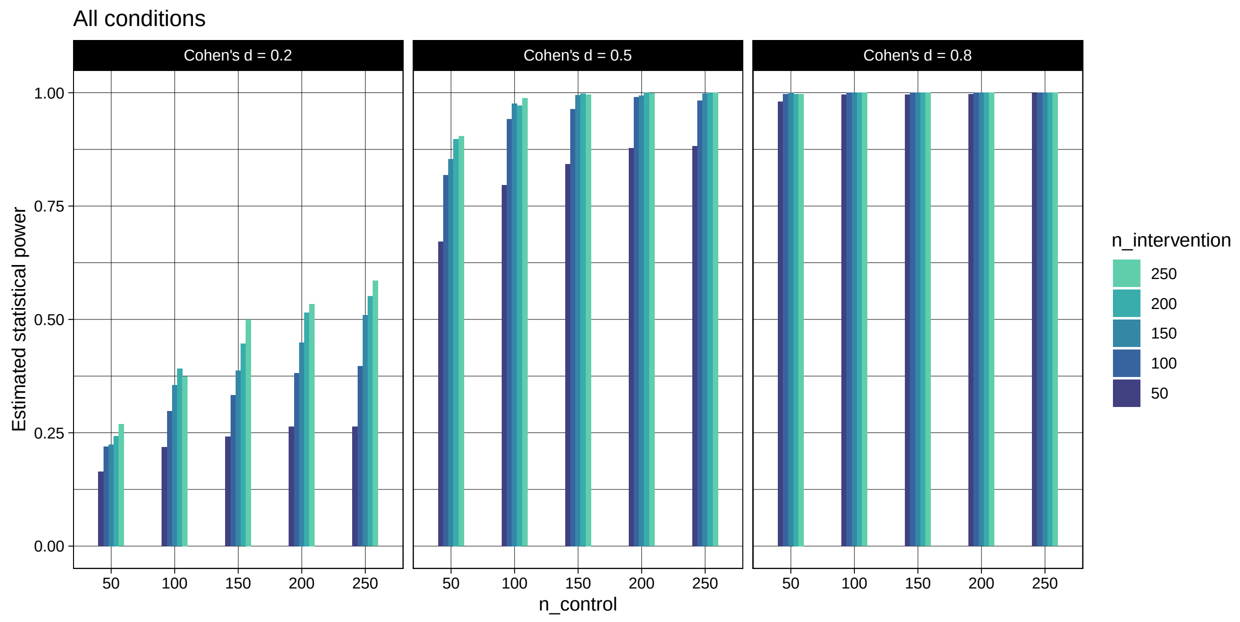
When you build simulations iteratively like we did above, its easy to lose track of which code is actually necessary to the final simulation vs. which bits were part of your own learning process. If you don't clean up your workflow every now and again, you will end up with spaghetti code that other people can't use, understand, or debug - and 'other people' often includes you yourself a few months from now.

So, in the below chunk, I pull together all the necessary code into (and no unnecessary code), and use `rm(list`

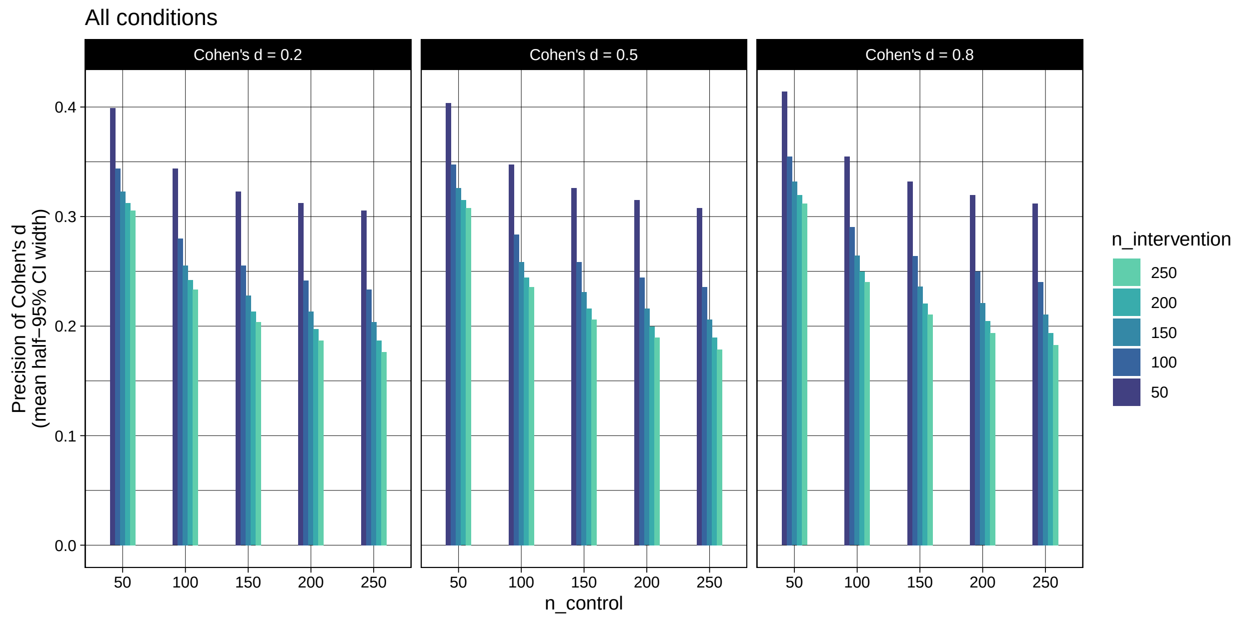
= ls()) to remove all previously saved objects from our R environment to ensure that we aren't carrying over old versions of functions. This ensures that the contents of this chunk alone represent a working simulation.

```r
# remove all objects from environment ----
rm(list = ls())


# dependencies ----
# repeated here for the sake of completeness

library(tidyr)
library(dplyr)
library(forcats)
library(readr)
library(purrr)
library(ggplot2)
library(effsize)


# set the seed ----
# for the pseudo random number generator to make results reproducible
set.seed(42)


# define data generating function ----
generate_data <- function(n_control,
                          n_intervention,
                          mean_control,
                          mean_intervention,
                          sd_control,
                          sd_intervention) {

  data <-
    bind_rows(
      tibble(condition = "control",
             score = rnorm(n = n_control, mean = mean_control, sd = sd_control)),
      tibble(condition = "intervention",
             score = rnorm(n = n_intervention, mean = mean_intervention, sd = sd_intervention))
    ) |>
    # control's factor levels must be ordered so that intervention is the first level and control is th
    # this ensures that positive cohen's d values refer to intervention > control and not the other way
    mutate(condition = fct_relevel(condition, "intervention", "control"))

  return(data)
}


# define data analysis function ----
analyse_data <- function(data) {
  # dependencies
  require(effsize)

  res_t_test <- t.test(formula = score ~ condition,
                       data = data,
                       var.equal = TRUE,
                       alternative = "two.sided")

  res_cohens_d <- effsize::cohen.d(formula = score ~ condition,  # new addition: also fit cohen's d
                                   within = FALSE,
                                   data = data)

  res <- tibble(p = res_t_test$p.value,
                cohens_d = res_cohens_d$estimate,  # new addition: save cohen's d and its 95% CIs to th
                cohens_d_ci_lower = res_cohens_d$conf.int["lower"],
                cohens_d_ci_upper = res_cohens_d$conf.int["upper"])

  return(res)
}


# define experiment parameters ----
experiment_parameters_grid <- expand_grid(
  n_control = seq(from = 50, to = 250, by = 50),
  n_intervention = seq(from = 50, to = 250, by = 50),
  mean_control = 0,
  mean_intervention = c(0.2, 0.5, 0.8), # small, medium, large Cohen's d
  sd_control = 1,
  sd_intervention = 1,
  iteration = 1:1000 # increased number of iterations for more stable estimates. NB real stimulation ar
)


# run simulation ----
simulation <-
  # using the experiment parameters
  experiment_parameters_grid |>

  # generate data using the data generating function and the parameters relevant to data generation
  mutate(generated_data = pmap(list(n_control,
                                    n_intervention,
                                    mean_control,
                                    mean_intervention,
                                    sd_control,
                                    sd_intervention),
                               generate_data)) |>

  # apply the analysis function to the generated data using the parameters relevant to analysis
  mutate(analysis_results = pmap(list(generated_data),
                                 analyse_data))


# # optional: save results to disk
write_rds(simulation, "simulation_1_results.rds")
# # read from disk in future if it exists already
simulation <- read_rds("simulation_1_results.rds")
```

```r
# summarise simulation results over the iterations ----
## estimate power
## ie what proportion of p values are significant (< .05)
simulation_summary <- simulation |>
  unnest(analysis_results) |>
  mutate(n_control = as.factor(n_control),
         n_intervention = as.factor(n_intervention),
         true_effect = paste("Cohen's d =", mean_intervention)) |>
  group_by(n_control,
           n_intervention,
           true_effect) |>
  summarize(power = mean(p < .05),
            mean_cohens_d_precision = mean((cohens_d_ci_upper - cohens_d_ci_lower)/2),
            .groups = "drop")

# plot power
ggplot(simulation_summary, aes(n_control, power, fill = n_intervention)) +
  geom_col(position = position_dodge(width = 0.4), width = 0.4) +
  scale_fill_viridis_d(option = "mako", begin = 0.3, end = 0.8,
                       guide = guide_legend(reverse = TRUE)) +
  theme_linedraw() +
  ggtitle("All conditions") +
  ylab("Estimated statistical power") +
  facet_wrap(~ true_effect)
```



```r
# plot precision
ggplot(simulation_summary, aes(n_control, mean_cohens_d_precision, fill = n_intervention)) +
  geom_col(position = position_dodge(width = 0.4), width = 0.4) +
  scale_fill_viridis_d(option = "mako", begin = 0.3, end = 0.8,
                       guide = guide_legend(reverse = TRUE)) +
  theme_linedraw() +
  ggtitle("All conditions") +
  ylab("Precision of Cohen's d\n(mean half-95% CI width)") +
  facet_wrap(~ true_effect)
```

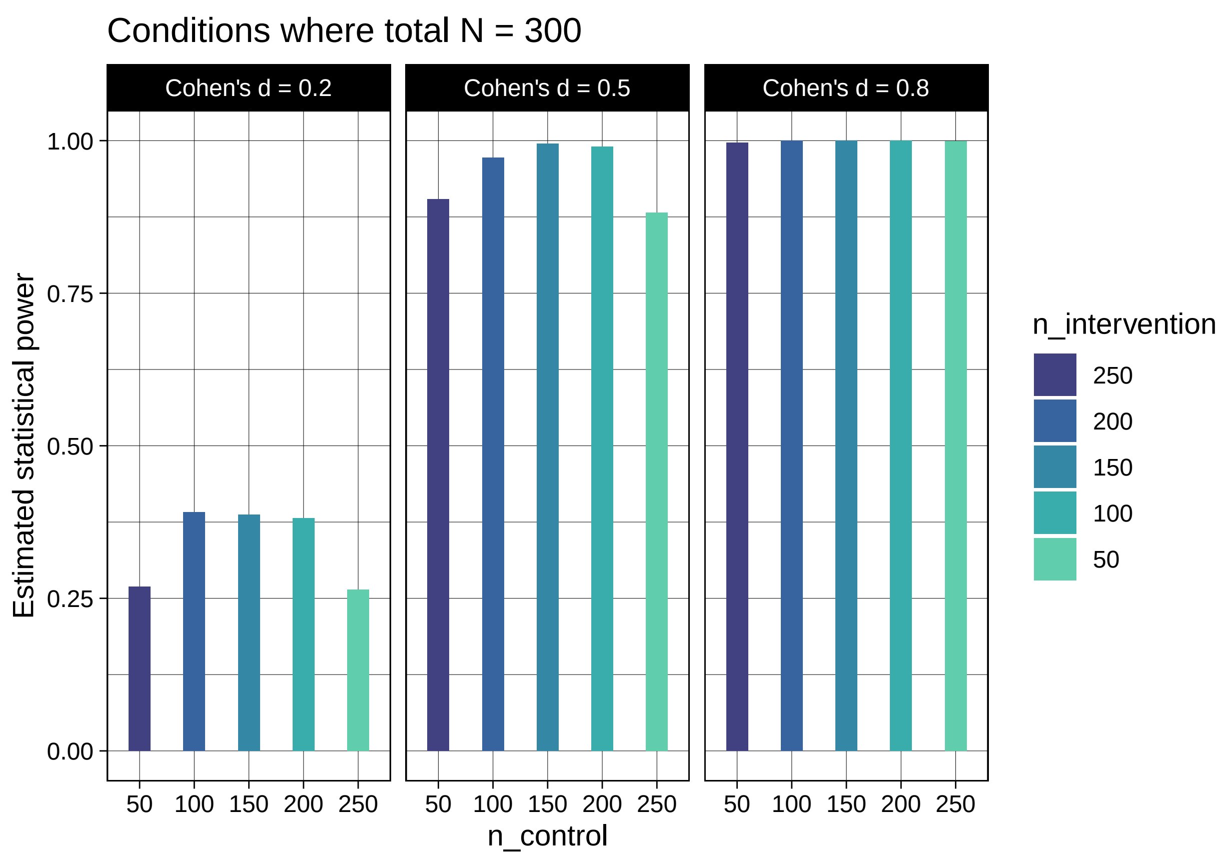


Make inferences across experimental conditions

Now that we've learned something about how to build simulations generally, what can we learn from this stimulation specifically?

Let's plot only those simulations where the total sample size between the two conditions is n = 300. By doing this, we are effectively simulating the impact of (un)balanced sample sizes on an independent t-test's statistical power. That is, all the columns refer to the same total sample size, but they differ in the number allotted to the control vs intervention conditions.

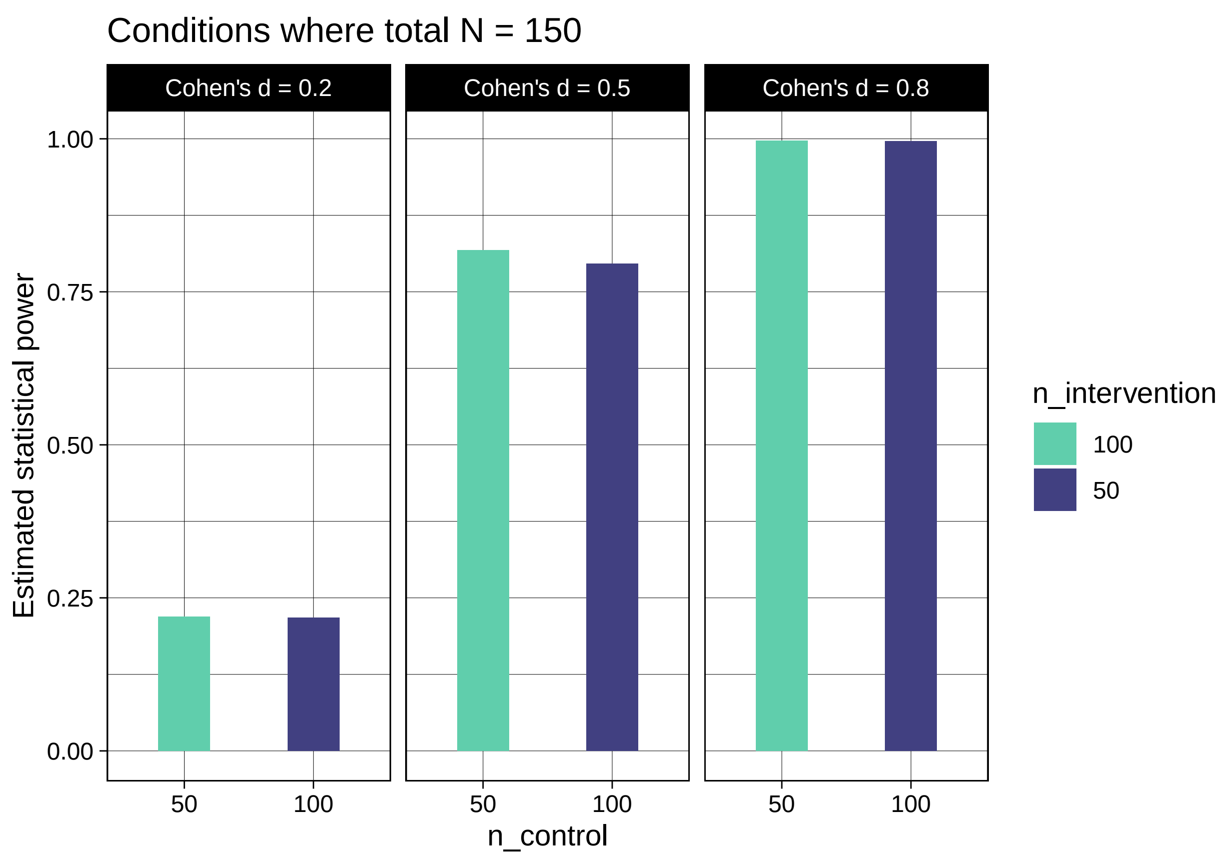
```r
simulation |>
  unnest(analysis_results) |>
  filter(n_control + n_intervention == 300) |> # subset only the conditions where total n = 300
  mutate(n_control = as.factor(n_control),
         n_intervention = as.factor(n_intervention),
         true_effect = paste("Cohen's d =", mean_intervention)) |>
  group_by(n_control,
           n_intervention,
           true_effect) |>
  summarize(power = mean(p < .05), .groups = "drop") |>
  ggplot(aes(n_control, power, fill = n_intervention)) +
  geom_col(position = position_dodge(width = 0.4), width = 0.4) +
  scale_fill_viridis_d(option = "mako", begin = 0.3, end = 0.8, direction = -1,
                       guide = guide_legend(reverse = TRUE)) +
  theme_linedraw() +
  ggtitle("Conditions where total N = 300") +
  ylab("Estimated statistical power") +
  facet_wrap(~ true_effect)
```



This shows that power is highest when the two conditions have balanced sample sizes, and tells us something about how much power decreases when sample sizes are unbalanced.

Note however that larger sample sizes are always preferable over smaller ones! I have seen students make this mistake recently. Don't ever not have a larger sample size if you can. E.g., compare power in the above conditions where total n = 300 with the below conditions where total n = 150, you gain more power from additional unbalanced participants than you lost due to imbalance.

```
simulation |>
  unnest(analysis_results) |>
  filter(n_control + n_intervention == 150) |> # subset only the conditions where total n = 150
  mutate(n_control = as.factor(n_control),
         n_intervention = as.factor(n_intervention),
         true_effect = paste("Cohen's d =", mean_intervention)) |>
  group_by(n_control,
           n_intervention,
           true_effect) |>
  summarize(power = mean(p < .05), .groups = "drop") |>
  ggplot(aes(n_control, power, fill = n_intervention)) +
  geom_col(position = position_dodge(width = 0.4), width = 0.4) +
  scale_fill_viridis_d(option = "mako", begin = 0.3, end = 0.8,
                       guide = guide_legend(reverse = TRUE)) +
  theme_linedraw() +
  ggtitle("Conditions where total N = 150") +
  ylab("Estimated statistical power") +
  facet_wrap(~ true_effect)
```



Wrapping up

This lesson taught you about the essential components of a simulation study, and introduced you to the way in which simulations are actually built in order to obtain each of those components (i.e., gradually and with lots of checking along the way).

It then used this simulation to answer a quantitative methods research question about the statistical power of a Student's *t*-test when the sample sizes are unbalanced.

To test your learning in this lesson, complete the assignment named "2_general_structure_of_a_simulation___assignment.Rmd I will later pass out a solution to this assignment that you can study and compare your solution to.

Session info

```r
sessionInfo()
```

```
## R version 4.3.2 (2023-10-31 ucrt)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 11 x64 (build 22631)
##
## Matrix products: default
##
##
## locale:
## [1] LC_COLLATE=German_Switzerland.utf8  LC_CTYPE=German_Switzerland.utf8
## [3] LC_MONETARY=German_Switzerland.utf8 LC_NUMERIC=C
## [5] LC_TIME=German_Switzerland.utf8
##
## time zone: Europe/Zurich
## tzcode source: internal
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] effsize_0.8.1 ggplot2_3.5.1 purrr_1.0.2   readr_2.1.5   forcats_1.0.0
## [6] dplyr_1.1.4   tidyr_1.3.1
##
## loaded via a namespace (and not attached):
##  [1] gtable_0.3.5      highr_0.10        compiler_4.3.2    tidyselect_1.2.1
##  [5] scales_1.3.0      yaml_2.3.8        fastmap_1.1.1     R6_2.5.1
##  [9] labeling_0.4.3    generics_0.1.3    knitr_1.46        tibble_3.2.1
## [13] munsell_0.5.1     pillar_1.9.0      tzdb_0.4.0        rlang_1.1.3
## [17] utf8_1.2.4        xfun_0.43         viridisLite_0.4.2 cli_3.6.2
## [21] withr_3.0.0       magrittr_2.0.3    digest_0.6.35     grid_4.3.2
## [25] rstudioapi_0.16.0 hms_1.1.3         lifecycle_1.0.4   vctrs_0.6.5
## [29] evaluate_0.23     glue_1.7.0        farver_2.1.1      fansi_1.0.6
## [33] colorspace_2.1-0  rmarkdown_2.26    tools_4.3.2       pkgconfig_2.0.3
## [37] htmltools_0.5.8.1
```