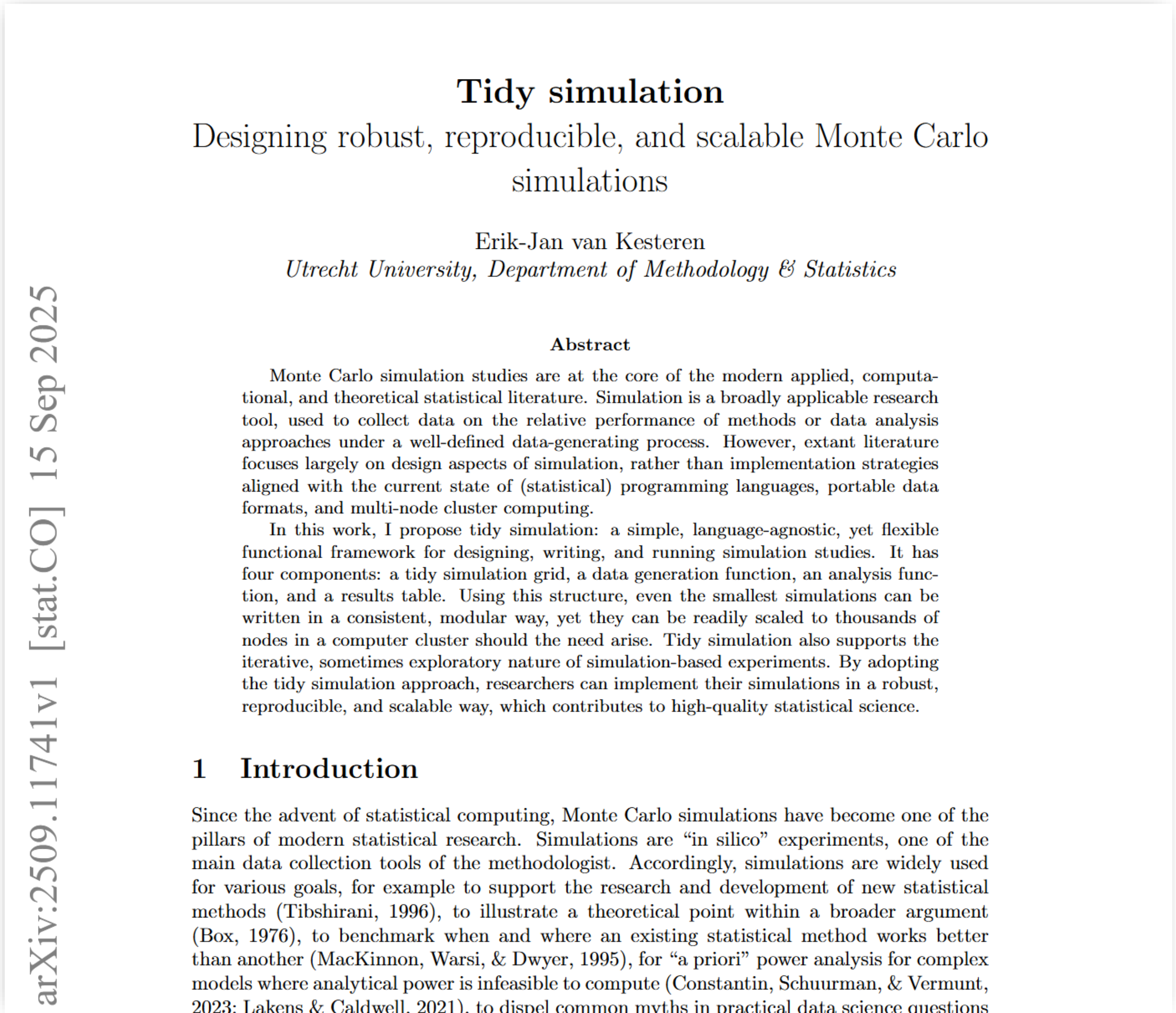
TIDY SIMULATION

Designing robust, reproducible, and scalable Monte Carlo simulations

Erik-Jan van Kesteren
Assistant prof Statistics & Data Science
Utrecht University, NL

arxiv.org/abs/2509.11741



arXiv:2509.11741v1 [stat.CO] 15 Sep 2025

Tidy simulation

Designing robust, reproducible, and scalable Monte Carlo simulations

Erik-Jan van Kesteren
Utrecht University, Department of Methodology & Statistics

Abstract

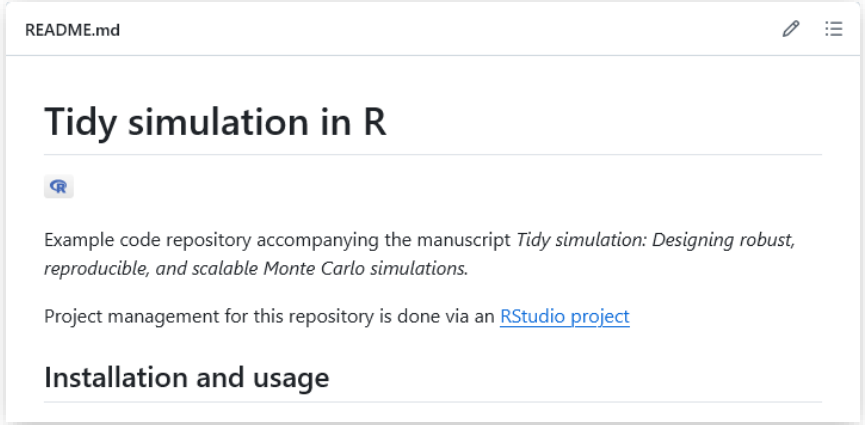
Monte Carlo simulation studies are at the core of the modern applied, computational, and theoretical statistical literature. Simulation is a broadly applicable research tool, used to collect data on the relative performance of methods or data analysis approaches under a well-defined data-generating process. However, extant literature focuses largely on design aspects of simulation, rather than implementation strategies aligned with the current state of (statistical) programming languages, portable data formats, and multi-node cluster computing.

In this work, I propose tidy simulation: a simple, language-agnostic, yet flexible functional framework for designing, writing, and running simulation studies. It has four components: a tidy simulation grid, a data generation function, an analysis function, and a results table. Using this structure, even the smallest simulations can be written in a consistent, modular way, yet they can be readily scaled to thousands of nodes in a computer cluster should the need arise. Tidy simulation also supports the iterative, sometimes exploratory nature of simulation-based experiments. By adopting the tidy simulation approach, researchers can implement their simulations in a robust, reproducible, and scalable way, which contributes to high-quality statistical science.

1 Introduction

Since the advent of statistical computing, Monte Carlo simulations have become one of the pillars of modern statistical research. Simulations are "in silico" experiments, one of the main data collection tools of the methodologist. Accordingly, simulations are widely used for various goals, for example to support the research and development of new statistical methods (Tibshirani, 1996), to illustrate a theoretical point within a broader argument (Box, 1976), to benchmark when and where an existing statistical method works better than another (MacKinnon, Warsi, & Dwyer, 1995), for "a priori" power analysis for complex models where analytical power is infeasible to compute (Constantin, Schuurman, & Vermunt, 2023; Lakens & Caldwell, 2021), to dispel common myths in practical data science questions

github.com/vankesteren/tidy_simulation



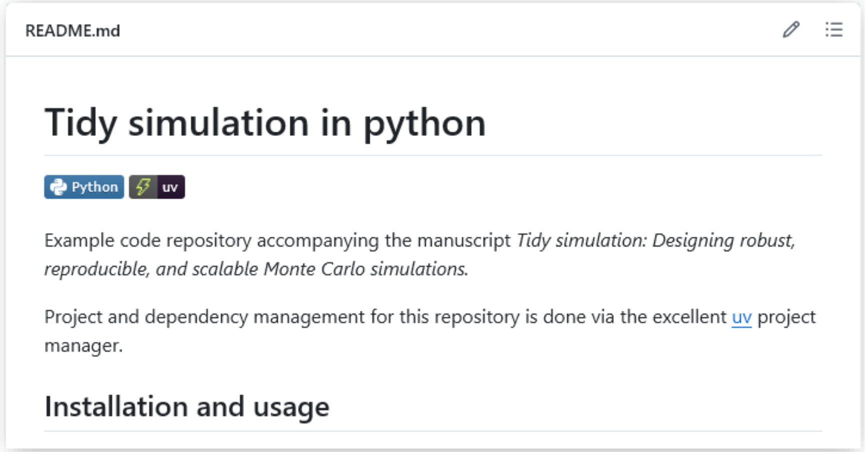
README.md

Tidy simulation in R

Example code repository accompanying the manuscript *Tidy simulation: Designing robust, reproducible, and scalable Monte Carlo simulations.*

Project management for this repository is done via an RStudio project

Installation and usage



README.md

Tidy simulation in python

Example code repository accompanying the manuscript *Tidy simulation: Designing robust, reproducible, and scalable Monte Carlo simulations.*

Project and dependency management for this repository is done via the excellent uv project manager.

Installation and usage

Why this work?

- I maintain our department’s simulation server
- Students and staff alike run into problems
- Often about robustness, reproducibility, and scalability

There is a didactic need for structure

Why this work? Disclaimer

I will present a “framework” for implementing statistical simulations

Others have had similar ideas! I try to refer to them, but I might have missed some.

- If you know of others, let me know!

TL;DR

1. Simulation grid
2. Generation function
3. Analysis function
4. Results table

Statistical simulations

Statistical simulation

- Data collection method for methodologists and statisticians
- Augment theory / mathematical analysis
- General process:
  1. Simulate data from a known generative distribution
  2. Apply methods of interest (baseline vs fancy model)
  3. Compare the results to determine which is better

A statistical simulation is an experiment on your computer

You should think about it design it, run it, and analyze it as such

Aims	What is the research question?
Data-generating mechanism	How is the stochastic data generated?
Estimand & Method	Which method do we apply to the generated data?
Performance measures	Which metric are we interested in comparing and presenting?

Morris, T. P., White, I. R., & Crowther, M. J. (2019). Using simulation studies to evaluate statistical methods. Statistics in medicine, 38 (11), 2074–2102

Classic example

- RCT with a binary treatment and two measurement occasions: pre- and post-treatment
- Data collection is very expensive: small sample size
- What power to detect the treatment effect?
- Different linear model options:
  - Outcome: change score (post-pre) or post-treatment
  - Covariate: pre-treatment or no correction.

Classic example

Aims	Which method is best at detecting treatment effect?
Data-generating mechanism	Gaussian data with treatment effect sizes from 0.0 to 1.0
Estimand & Method	▪ post uncorrected ▪ post corrected ▪ change-score uncorrected ▪ change-score corrected
Performance measures	Power: proportion of true positives

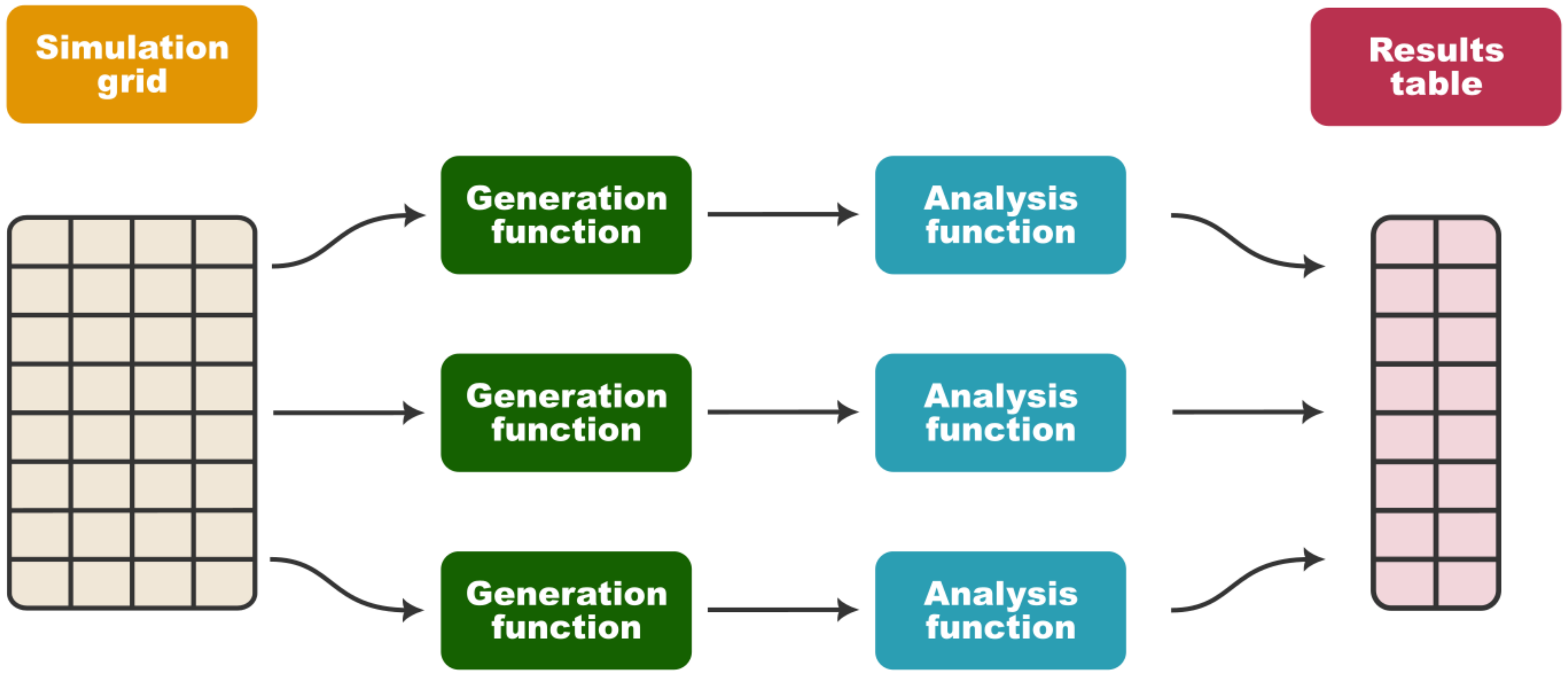
Tidy simulation

Tidy simulation

- **Tidy** as in original idea of **tidy data** (Wickham, 2014) as an underlying data structure
- NOT as in tidyverse, tidymodels, etc., syntax / implementations
- Language-agnostic framework! You want python, Julia, ...? No problem!

Four steps

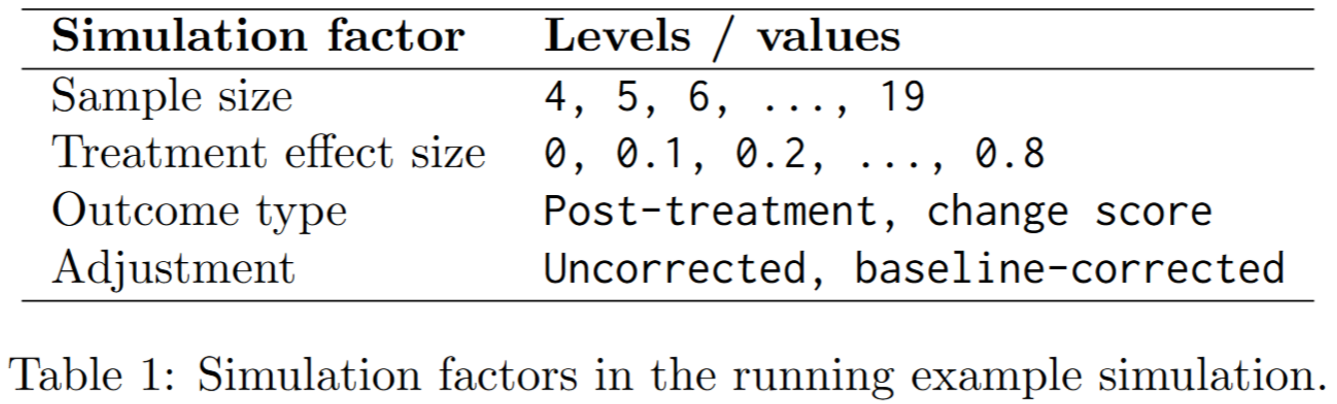
1. Simulation grid
2. Generation function
3. Analysis function
4. Results table


Simulation grid
Generation function
Generation function
Generation function
Analysis function
Analysis function
Analysis function
Results table

1

Simulation grid

First, determine the factors to vary (simulation conditions) from the design



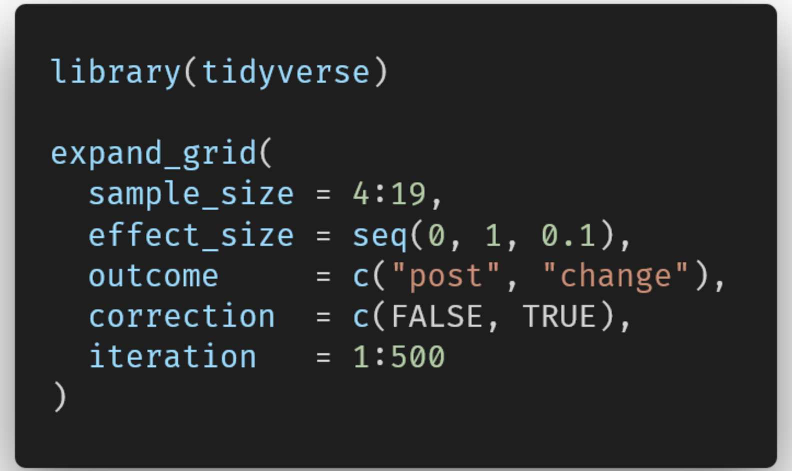
Simulation factor	Levels / values
Sample size	`4, 5, 6, ..., 19`
Treatment effect size	`0, 0.1, 0.2, ..., 0.8`
Outcome type	`Post-treatment, change score`
Adjustment	`Uncorrected, baseline-corrected`

Table 1: Simulation factors in the running example simulation.

1

Simulation grid

Tidy data frame with settings for each iteration



```r
library(tidyverse)

expand_grid(
  sample_size = 4:19,
  effect_size = seq(0, 1, 0.1),
  outcome     = c("post", "change"),
  correction  = c(FALSE, TRUE),
  iteration   = 1:500
)
```

1

Simulation grid

Tidy data frame with settings for each iteration

```python
from polarsgrid import expand_grid

simulation_grid = expand_grid(
  sample_size=[4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
  effect_size=[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
  outcome=["post", "change"],
  correction=[False, True],
  iteration=list(range(500)),
)
```

1

Simulation grid

Tidy data frame with settings for each iteration

	Generation factors		*Analysis factors*		
Row ID	Sample size	Effect size	Outcome	Correction	Iteration
1	4	0.0	post	false	1
2	5	0.0	post	false	1
3	6	0.0	post	false	1
⋮	⋮	⋮	⋮	⋮	
17600	19	1.0	change	true	250
17601	4	0.0	post	false	251
17602	5	0.0	post	false	251
⋮	⋮	⋮	⋮	⋮	
351998	17	1.0	change	true	500
351999	18	1.0	change	true	500
352000	19	1.0	change	true	500

1

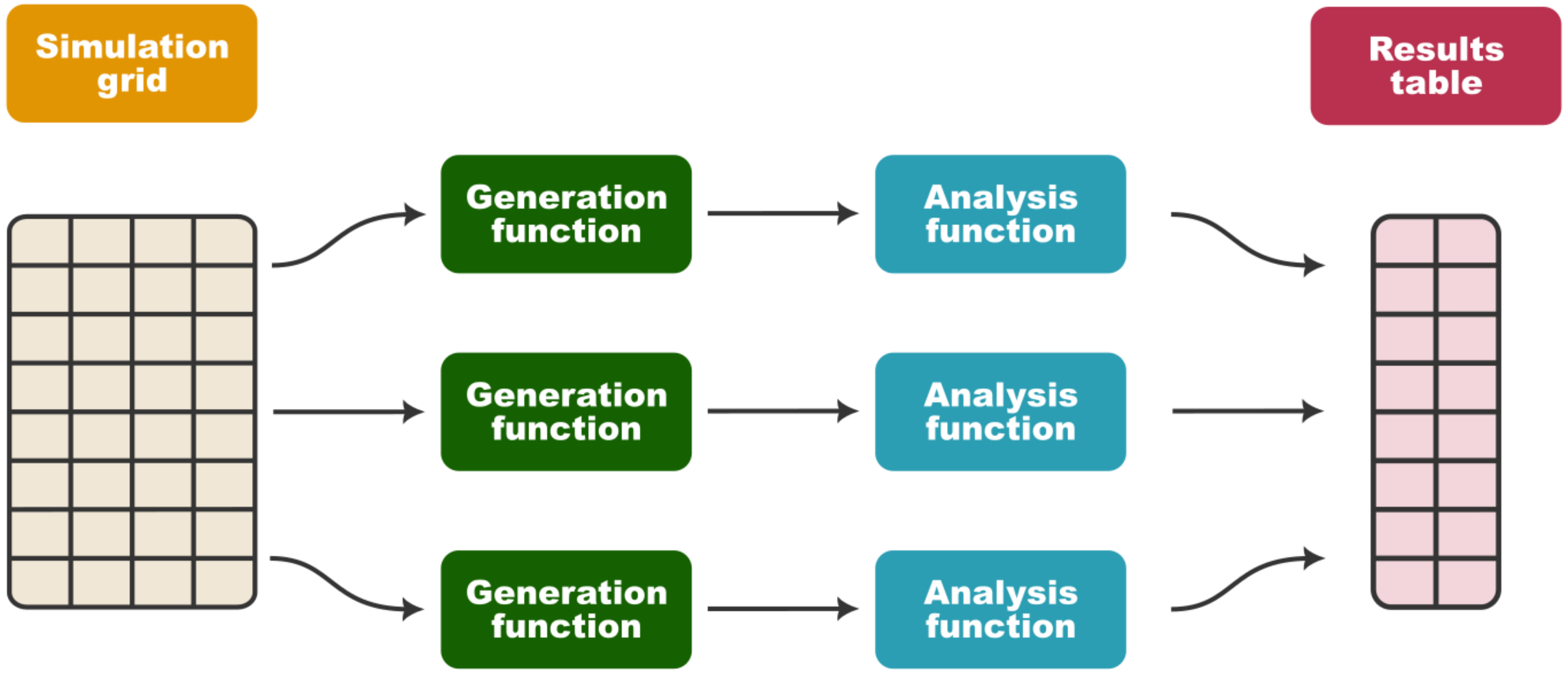
Simulation grid

- If data generation takes long: repeated measures experimental design
- "Wide" dataset with multiple analysis factors on each row
- Can be transformed back to "long" data for analysis (pivot_longer / unpivot)

1

Simulation grid

- Post-processing is allowed!
- If needed: add extra metadata
  - random seed
  - temporary storage directory
- Simulation grid is backbone of your analyses


Simulation grid
Results table
Generation function
Analysis function
Generation function
Analysis function
Generation function
Analysis function

2

Generation function

Function that generates a single dataset to be analyzed

Input generation factors from 1 row of grid

Output single (ideally tidy) simulated dataset

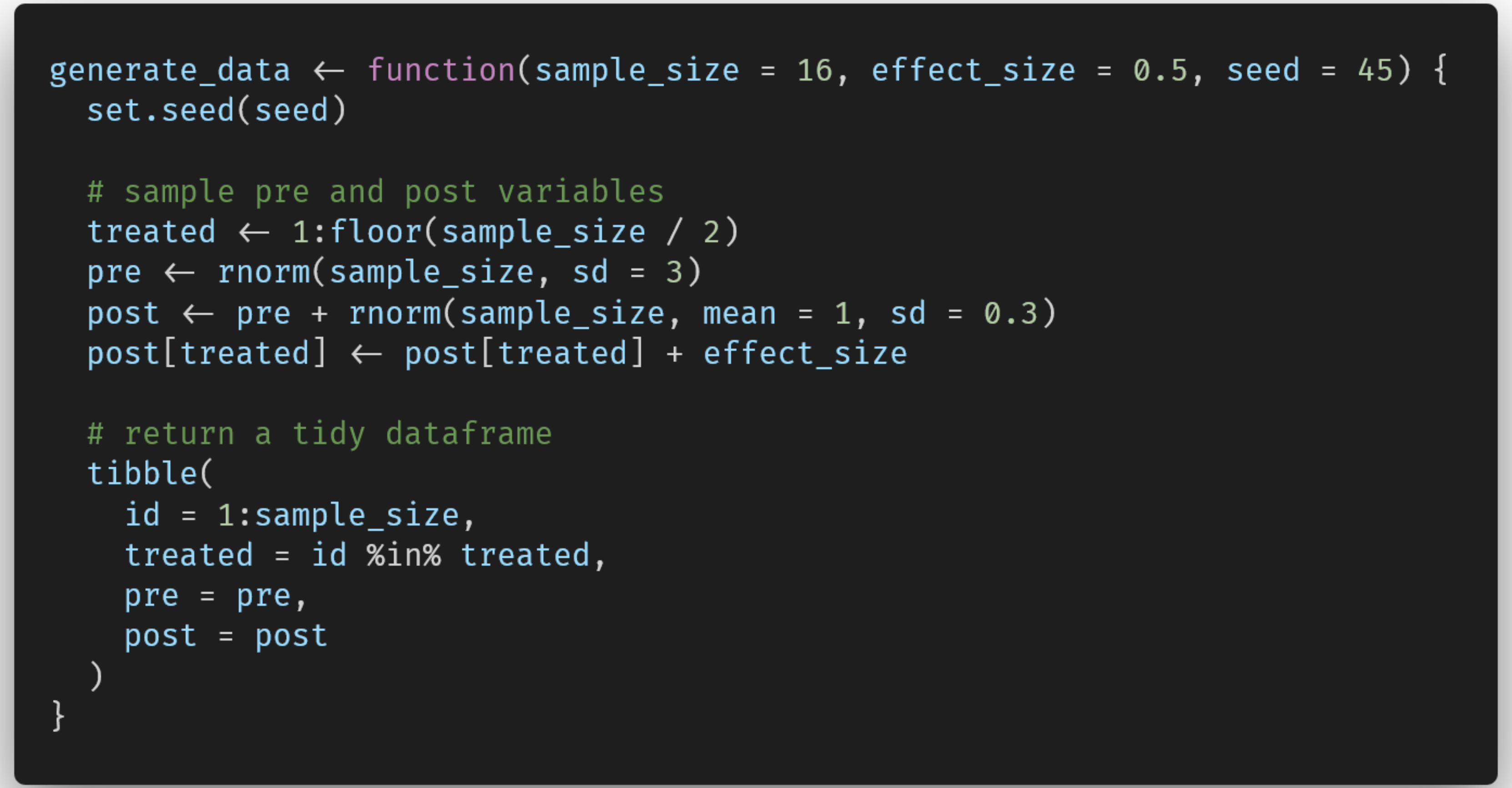
2

Generation function

- This may run many thousands of times
- Make it robust and efficient:
  - Use industry-standard random generators
    stats::rnorm(), scipy or numpy, Distributions.jl
  - Modularize when necessary
  - Should be able to handle any combination of input parameters & error appropriately

2

Generation function



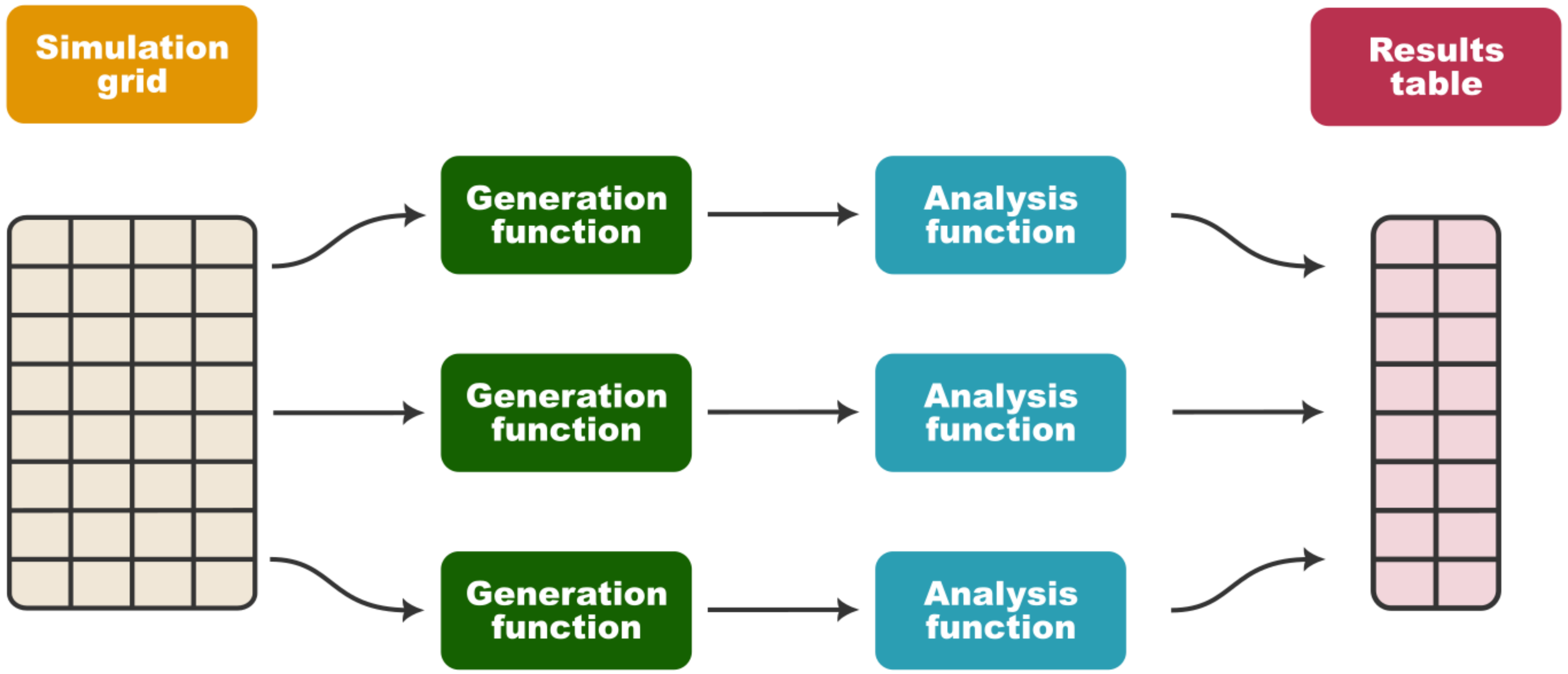
```r
generate_data ← function(sample_size = 16, effect_size = 0.5, seed = 45) {
  set.seed(seed)

  # sample pre and post variables
  treated ← 1:floor(sample_size / 2)
  pre ← rnorm(sample_size, sd = 3)
  post ← pre + rnorm(sample_size, mean = 1, sd = 0.3)
  post[treated] ← post[treated] + effect_size

  # return a tidy dataframe
  tibble(
    id = 1:sample_size,
    treated = id %in% treated,
    pre = pre,
    post = post
  )
}
```

2

Generation function

```python
def generate_data(N: int, P: int, seed: int):
    np.random.seed(seed)
    X = np.random.standard_normal((N, P))
    b = np.random.normal(1, 1, P)
    y = X @ b + np.random.normal(0, 1, N)
    return X, y, b
```

Keep it simple (simple → robust)


Simulation grid
Results table
Generation function
Analysis function
Generation function
Analysis function
Generation function
Analysis function

3

Analysis function

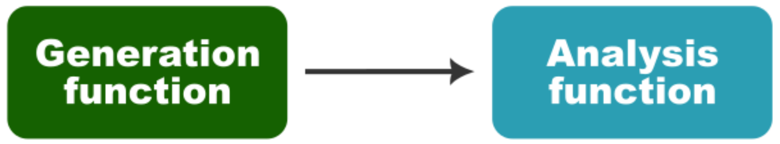
Function that computes metric of interest from dataset and analysis factors

Input 1 Data from data generation factors

Input 2 Analysis factors from 1 row of grid

Output single number or set of numbers

- The data flowing between generation function and analysis function form an interface (API)



- Keeping the API stable means you can change the components iteratively and everything “just works”

3

Analysis function

- This again needs to be really robust
- Be particularly careful about data-dependent errors
- Non-convergence, Heywood cases in SEM, missing data problems…

If $Pr(\text{error in one run}) = 0.00001$, then

$$Pr(\text{error in 500k runs}) = 1 - 0.99999^{500000} = \mathbf{99.3\%}$$

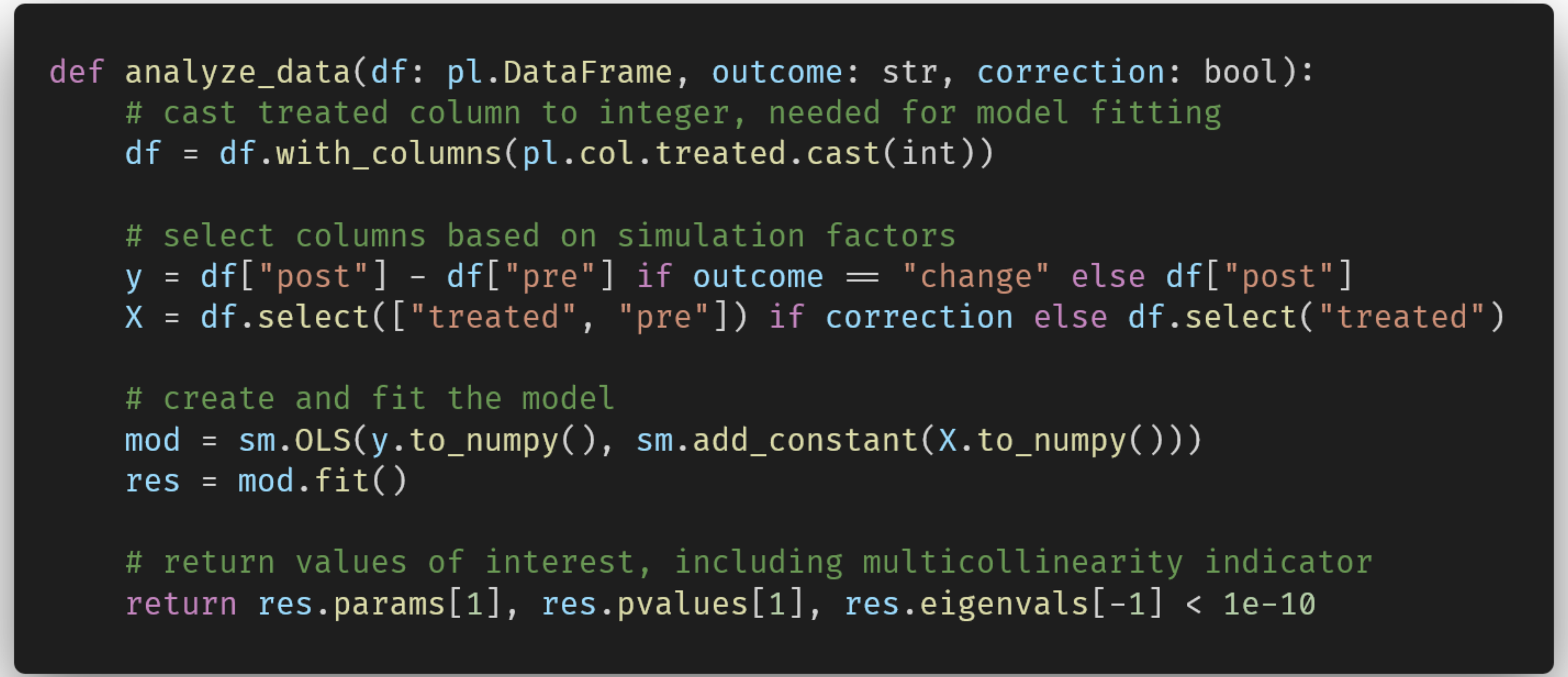
3

Analysis function

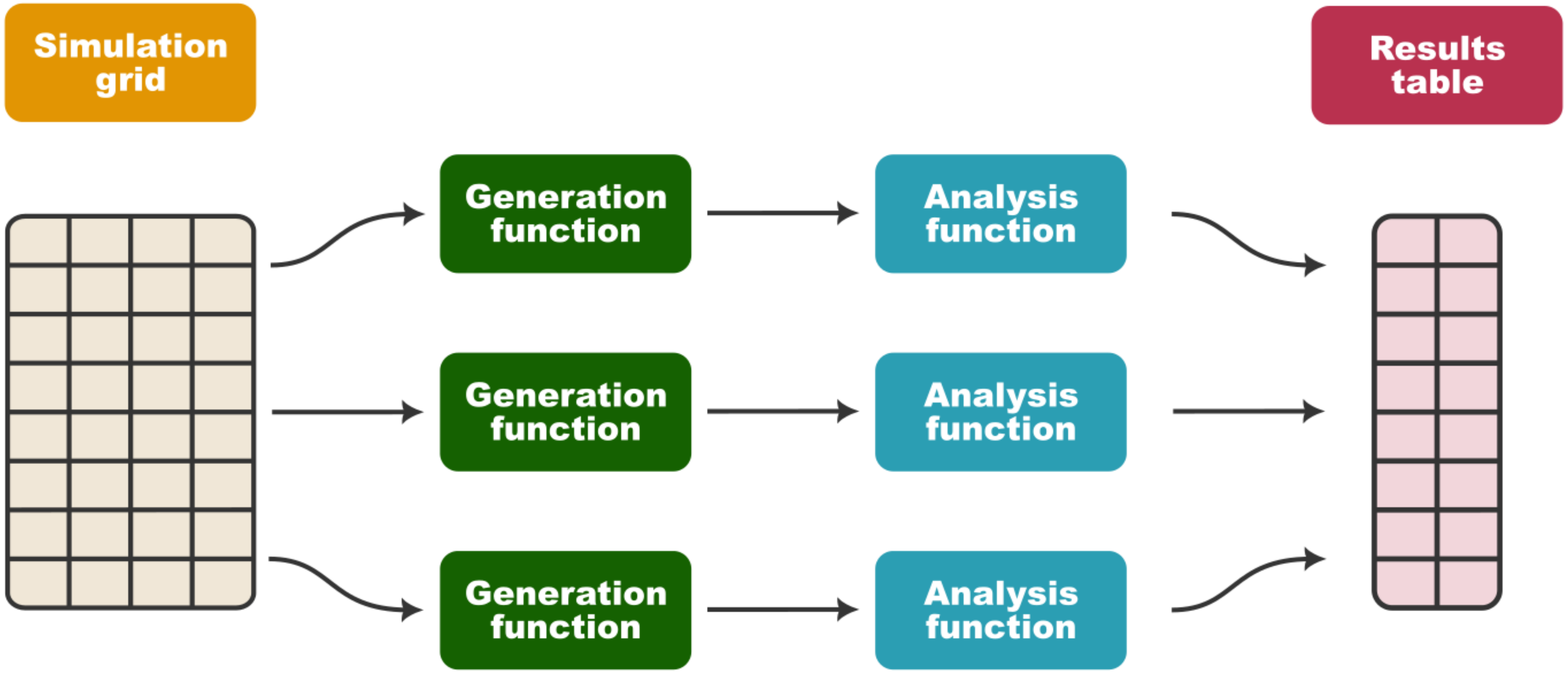
- Test your function against edge cases!
- Be aware of sources of errors
- Handle them so that simulation can continue
- Record the errors (outcome of interest?)

3

Analysis function



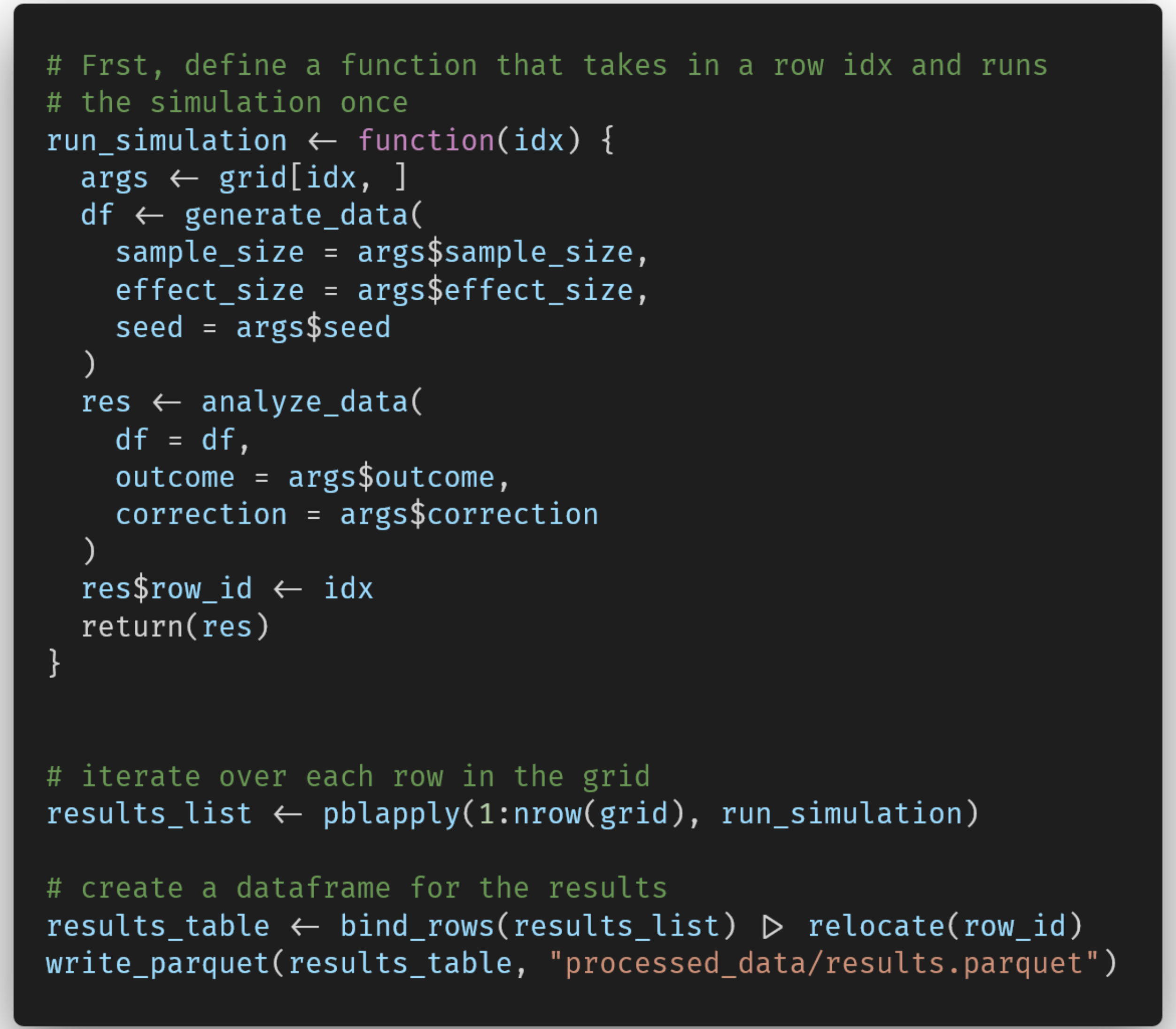
```python
def analyze_data(df: pl.DataFrame, outcome: str, correction: bool):
    # cast treated column to integer, needed for model fitting
    df = df.with_columns(pl.col.treated.cast(int))

    # select columns based on simulation factors
    y = df["post"] - df["pre"] if outcome == "change" else df["post"]
    X = df.select(["treated", "pre"]) if correction else df.select("treated")

    # create and fit the model
    mod = sm.OLS(y.to_numpy(), sm.add_constant(X.to_numpy()))
    res = mod.fit()

    # return values of interest, including multicollinearity indicator
    return res.params[1], res.pvalues[1], res.eigenvals[-1] < 1e-10
```


Simulation grid
Results table
Generation function
Generation function
Generation function
Analysis function
Analysis function
Analysis function

4

Results table

- Running the *generation function* and the *analysis function* on each row of the *grid* yields the *results table*
- Contains row index and metrics of interest
- An “embarrassingly parallel” program: apply, map, or loop



```r
# Frst, define a function that takes in a row idx and runs
# the simulation once
run_simulation <- function(idx) {
  args <- grid[idx, ]
  df <- generate_data(
    sample_size = args$sample_size,
    effect_size = args$effect_size,
    seed = args$seed
  )
  res <- analyze_data(
    df = df,
    outcome = args$outcome,
    correction = args$correction
  )
  res$row_id <- idx
  return(res)
}


# iterate over each row in the grid
results_list <- pblapply(1:nrow(grid), run_simulation)

# create a dataframe for the results
results_table <- bind_rows(results_list) |> relocate(row_id)
write_parquet(results_table, "processed_data/results.parquet")
```

4

Results table

Row ID	Estimate	p-value	Converged
1	0.01	0.973	true
2	0.12	0.522	true
3	-0.11	0.104	true
⋮	⋮	⋮	⋮
351998	1.03	0.001	true
351999	0.99	0.019	true
352000	0.96	0.002	true

And now the magic happens

Tidy data ensures tidy analysis

- Join (merge) the grid and the results on row id

	Generation factors		*Analysis factors*		
Row ID	Sample size	Effect size	Outcome	Correction	Iteration
1	4	0.0	post	false	1
2	5	0.0	post	false	1
3	6	0.0	post	false	1
⋮	⋮	⋮	⋮	⋮	
17600	19	1.0	change	true	250
17601	4	0.0	post	false	251
17602	5	0.0	post	false	251
⋮	⋮	⋮	⋮	⋮	
351998	17	1.0	change	true	500
351999	18	1.0	change	true	500
352000	19	1.0	change	true	500

Row ID	Estimate	p-value	Converged
1	0.01	0.973	true
2	0.12	0.522	true
3	-0.11	0.104	true
⋮	⋮	⋮	⋮
351998	1.03	0.001	true
351999	0.99	0.019	true
352000	0.96	0.002	true

- This creates a tidy “analysis” data frame

Tidy data ensures tidy analysis

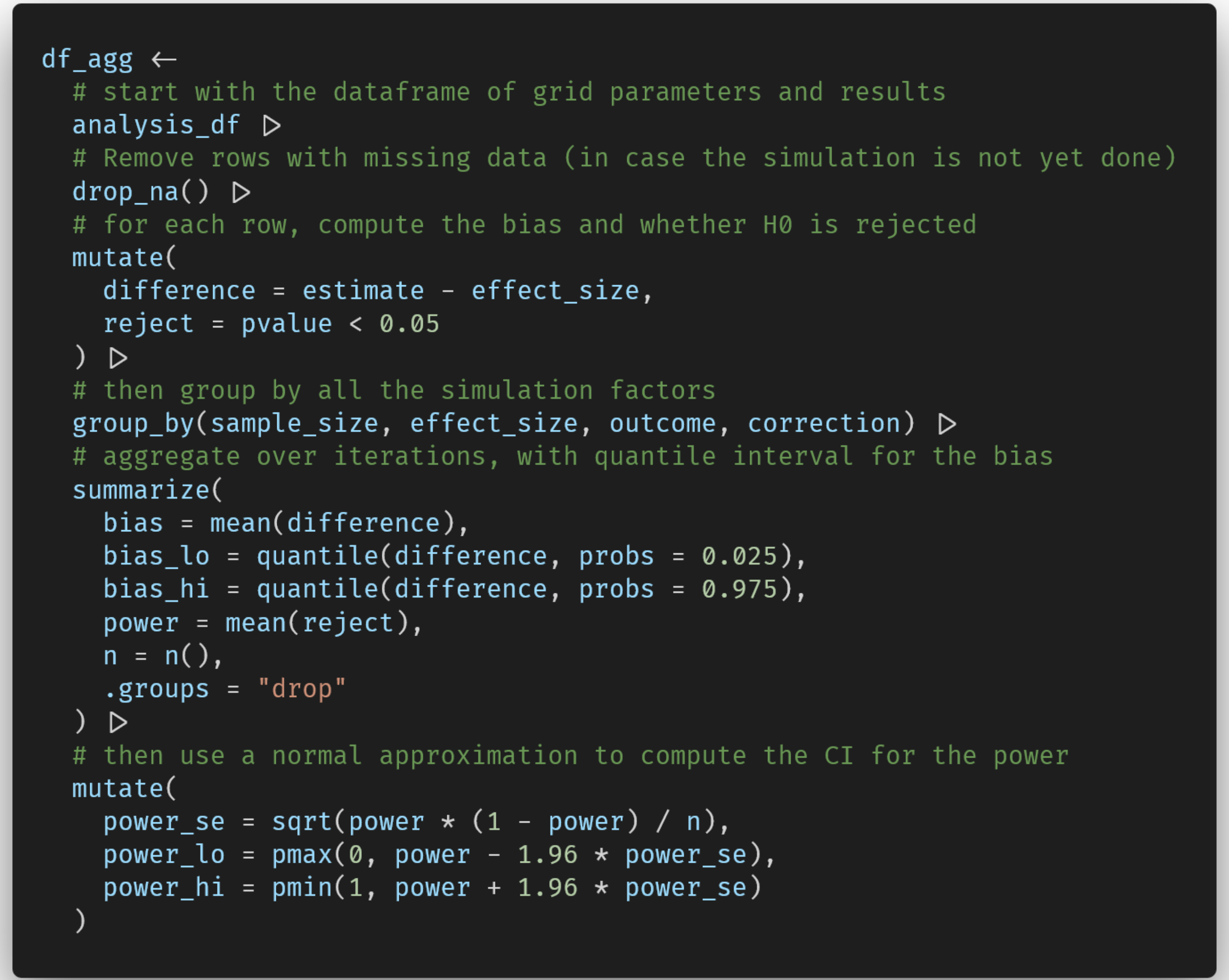
```r
# load the grid
simulation_grid ← read_parquet("processed_data/grid.parquet")

# load the results
results_table ← read_parquet("processed_data/results.parquet")

# combine them using a left join
analysis_df ← left_join(
  x = simulation_grid,
  y = results_table,
  by = join_by(row_id)
)
```

Tidy data ensures tidy analysis

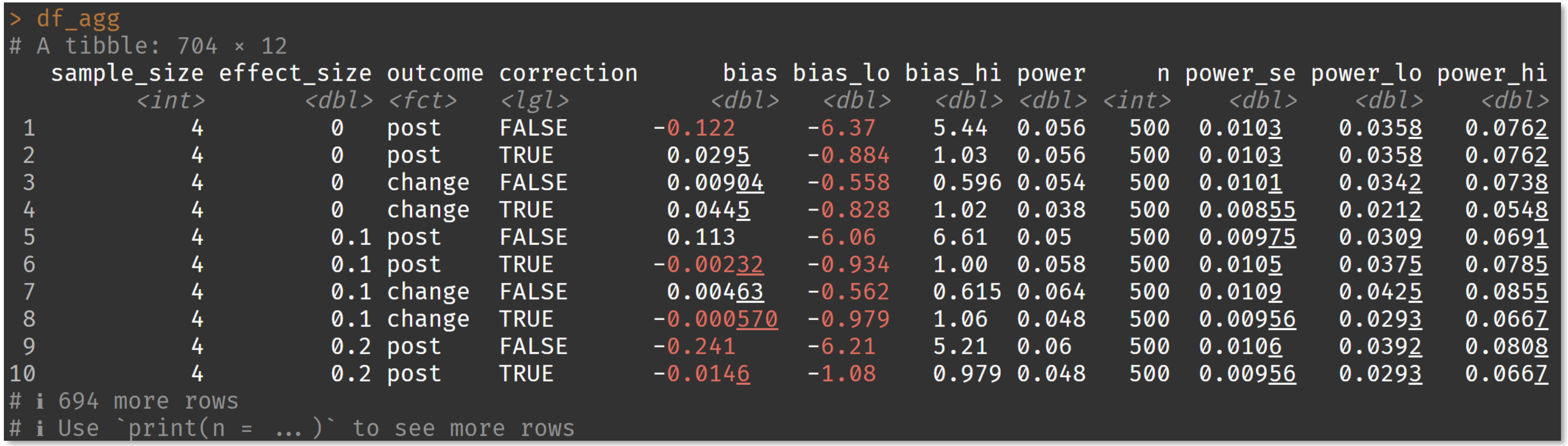
Now you can:

- Group by / aggregate (summarize) to create summary tables
- Plot using the grammar of graphics
- Run statistical models to test hypotheses about conditions/factors (e.g., GLM, ANOVA)

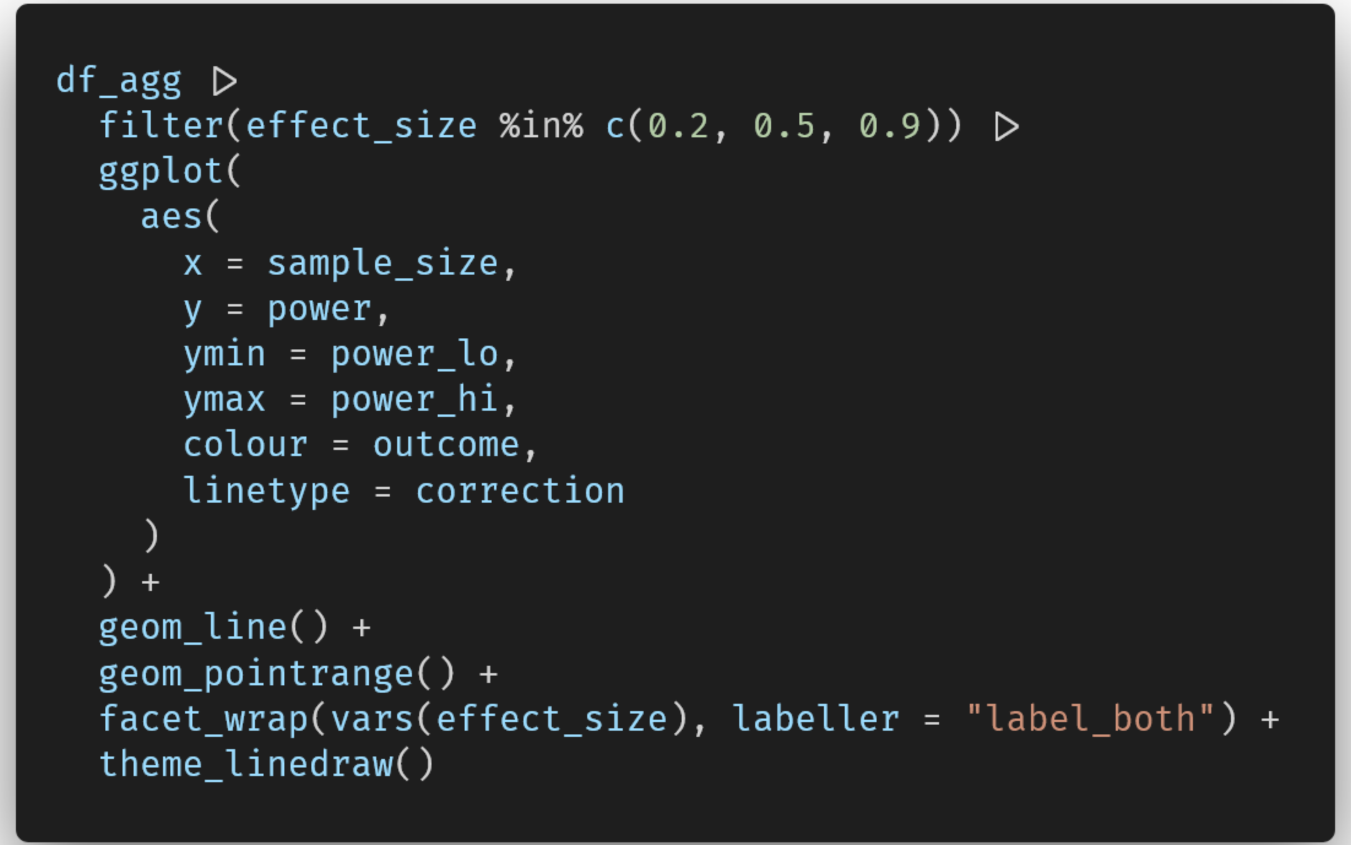
Because this is like data collected from an experiment!



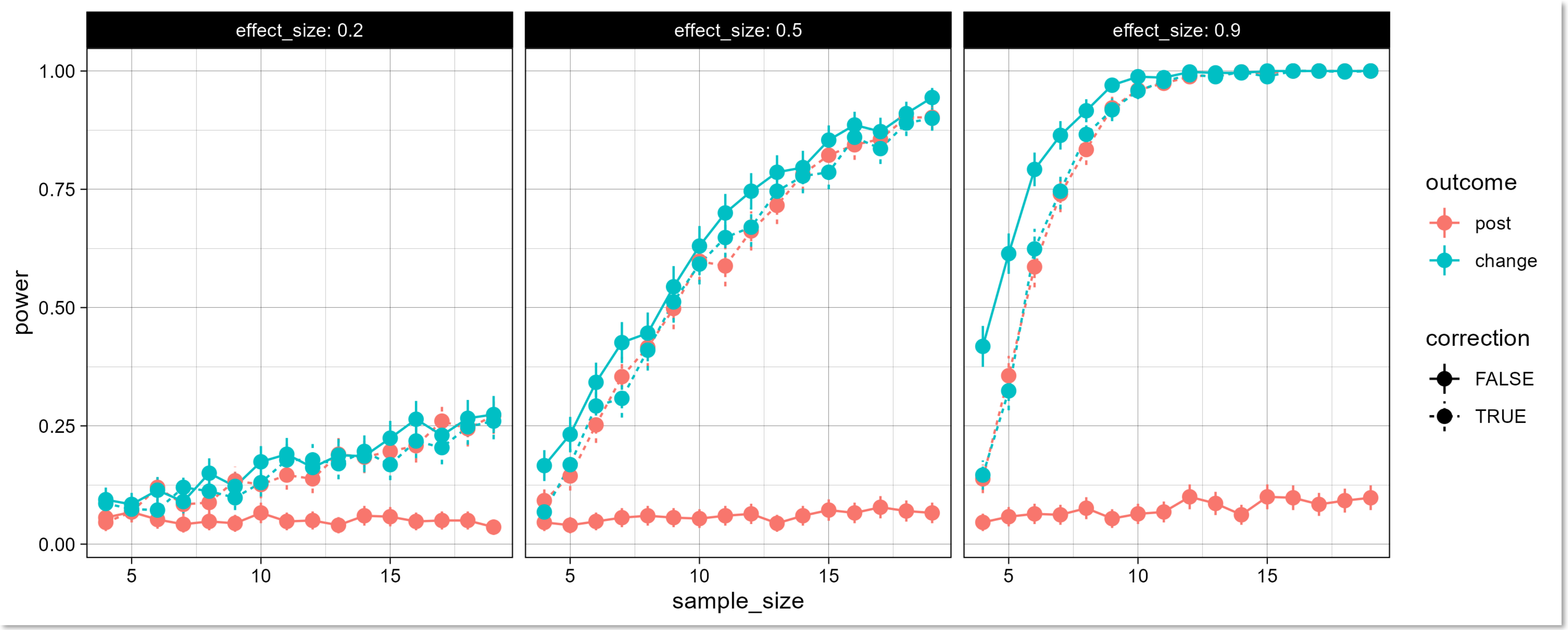
```r
df_agg <-
  # start with the dataframe of grid parameters and results
  analysis_df |>
  # Remove rows with missing data (in case the simulation is not yet done)
  drop_na() |>
  # for each row, compute the bias and whether H0 is rejected
  mutate(
    difference = estimate - effect_size,
    reject = pvalue < 0.05
  ) |>
  # then group by all the simulation factors
  group_by(sample_size, effect_size, outcome, correction) |>
  # aggregate over iterations, with quantile interval for the bias
  summarize(
    bias = mean(difference),
    bias_lo = quantile(difference, probs = 0.025),
    bias_hi = quantile(difference, probs = 0.975),
    power = mean(reject),
    n = n(),
    .groups = "drop"
  ) |>
  # then use a normal approximation to compute the CI for the power
  mutate(
    power_se = sqrt(power * (1 - power) / n),
    power_lo = pmax(0, power - 1.96 * power_se),
    power_hi = pmin(1, power + 1.96 * power_se)
  )
```

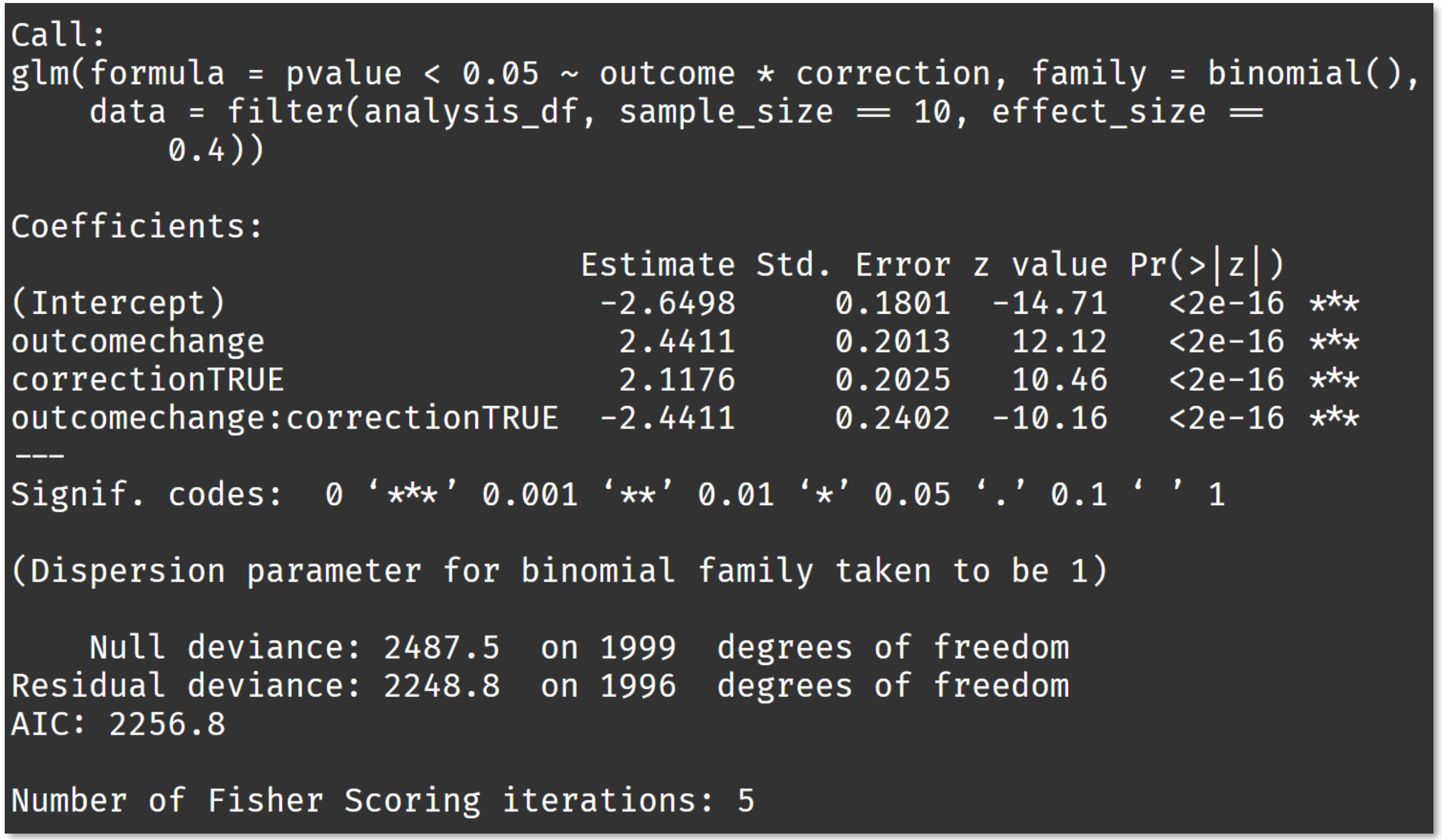


```
> df_agg
# A tibble: 704 × 12
   sample_size effect_size outcome correction      bias bias_lo bias_hi power     n power_se power_lo power_hi
         <int>       <dbl> <fct>   <lgl>          <dbl>   <dbl>   <dbl> <dbl> <int>    <dbl>    <dbl>    <dbl>
 1           4         0   post    FALSE      -0.122     -6.37    5.44  0.056   500  0.0103    0.0358   0.0762
 2           4         0   post    TRUE        0.0295    -0.884   1.03  0.056   500  0.0103    0.0358   0.0762
 3           4         0   change  FALSE       0.00904   -0.558   0.596 0.054   500  0.0101    0.0342   0.0738
 4           4         0   change  TRUE        0.0445    -0.828   1.02  0.038   500  0.00855   0.0212   0.0548
 5           4         0.1 post    FALSE       0.113     -6.06    6.61  0.05    500  0.00975   0.0309   0.0691
 6           4         0.1 post    TRUE       -0.00232   -0.934   1.00  0.058   500  0.0105    0.0375   0.0785
 7           4         0.1 change  FALSE       0.00463   -0.562   0.615 0.064   500  0.0109    0.0425   0.0855
 8           4         0.1 change  TRUE       -0.000570  -0.979   1.06  0.048   500  0.00956   0.0293   0.0667
 9           4         0.2 post    FALSE      -0.241     -6.21    5.21  0.06    500  0.0106    0.0392   0.0808
10           4         0.2 post    TRUE       -0.0146    -1.08    0.979 0.048   500  0.00956   0.0293   0.0667
# ℹ 694 more rows
# ℹ Use `print(n = ...)` to see more rows
```



```r
df_agg |>
  filter(effect_size %in% c(0.2, 0.5, 0.9)) |>
  ggplot(
    aes(
      x = sample_size,
      y = power,
      ymin = power_lo,
      ymax = power_hi,
      colour = outcome,
      linetype = correction
    )
  ) +
  geom_line() +
  geom_pointrange() +
  facet_wrap(vars(effect_size), labeller = "label_both") +
  theme_linedraw()
```


effect_size: 0.2
effect_size: 0.5
effect_size: 0.9
power
1.00
0.75
0.50
0.25
0.00
5
10
15
sample_size
outcome
post
change
correction
FALSE
TRUE



```
Call:
glm(formula = pvalue < 0.05 ~ outcome * correction, family = binomial(),
    data = filter(analysis_df, sample_size == 10, effect_size ==
        0.4))

Coefficients:
                             Estimate Std. Error z value Pr(>|z|)
(Intercept)                   -2.6498     0.1801  -14.71   <2e-16 ***
outcomechange                  2.4411     0.2013   12.12   <2e-16 ***
correctionTRUE                 2.1176     0.2025   10.46   <2e-16 ***
outcomechange:correctionTRUE  -2.4411     0.2402  -10.16   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 2487.5  on 1999  degrees of freedom
Residual deviance: 2248.8  on 1996  degrees of freedom
AIC: 2256.8

Number of Fisher Scoring iterations: 5
```

Conclusion

Conclusion

- Tidy simulation enables robust, reproducible, and scalable simulations
- Make your sim modular with grid, generation, analysis, and results

arxiv.org/abs/2509.11741