Code documentation for the simulation study: Comparison of confidence intervals summarizing the uncertainty of the combined estimate of a meta-analysis

Felix Hofmann

Contents

1 Overview

This document gives an overview over the code that implements the simulation study *Comparison of confidence intervals summarizing the uncertainty of the combined estimate of a meta-analysis.*

1.1 Terminology

In order to understand make the contents of this document more clear, the following terminology is used:

grid A `data.frame` with columns:

- `sampleSize` The sample size for all of the studies in the meta-analysis.
- `effect` The true effect θ in the meta-analysis. Here we use $\theta \in \{0.1, 0.2, 0.5\}$.
- `I2` Higgin's I^2, i. e. , the proportion of the variability in the meta-analysis that is explained by between-study heterogeneity rather than sampling error. Here we use $I^2 \in \{0, 0.3, 0.6, 0.9\}$.
- `k` The number of studies in a given meta-analysis data set. In this simulation we use $k \in \{3, 5, 10, 20, 50\}$.
- `heterogeneity` The heterogeneity model used in simulation. Currently, this is always `"additive"`.
- `dist` The distribution of the individual study effects. This is either `"Gaussian"` or `"t"`. The distribution always has mean θ.
- `bias` The publication bias. This is one of `"none"`, `"moderate"`, or `"strong"`.
- `large` The number of large studies in the meta-analysis. These large studies have $10 \cdot$ `sampleSize` participants. Currently, we use large $\in \{0, 1, 3\}$.

method With the term *method* we refer to a specific method used to calculate a confidence interval (CI) or point estimate. The methods of interest in this simulation are currently *Edgington* (Edgington, 1972) and *Fisher* (Fisher, 1932). As reference methods we use *Hartung-Knapp* (IntHout et al., 2014), *Henmi-Copas* (Henmi and Copas, 2010), and the well-known *Random Effects* model.

parameters In the context of this document, we use the term *parameters* to refer to a set of values that determines the simulation and assessment process within the simulation. Usually this is one row of `grid` (see above). In the code base, this variable appears as `pars`.

N The number of times we repeat the simulation and assessment of the confidence intervals and point estimates for all parameters. In the code base, this variable appears as `N`.

heterogeneity estimation method The method used to estimate the between-study heterogeneity τ^2. Currently, we use the four methods `"none"`, `"DL"`, `"PM"`, and `"REML"`. Method `"none"` means that we set $\tau^2 \stackrel{!}{=} 0$, while method `"DL"` refers to the method of DerSimonian and Laird (1986), `"PM"` denotes the method of Paule and Mandel (1982), and `"REML"` refers to the method of Harville (1977).

performance measure With the term *performance measures* we refer to the measures we use to assess the CI(s) and point estimate(s) for one single method and one single meta-analysis data set. As performance measures for CIs, we currently record the width of the CI (`width`), the score of the CI (`score`), the coverage of the true effect, i. e. , $\mathbb{1}_{\theta \in \text{CI}}$ (`coverage_true`), and the number of confidence intervals (`n`). Regarding the point estimate(s), we only record the estimate itself.

summary measure We define a *summary measure* as a measure that summarizes various records of a *performance measure* for a given method and fixed parameters. For CIs we use the mean to summarise the performance measures. For point estimates, we use mean squared error (MSE), bias, and variance. Moreover, we calculate the relative frequencies for the number of CIs.

1.2 Simulation details

The aim of this simulation is to assess the performance of novel estimation methods (*Fisher*, *Edgington*) for point estimates and confidence intervals in meta-analyses in comparison to established methods (*Hartung-Knapp*, *Henmi-Copas*, *Random Effects*).

Thus, we have designed the simulation procedure as shown in Figure 1. The implementation of the procedure can be found in the in function `sim()`, which is defined in file `simulation.R`. This process needs two inputs, a *grid of parameters* and N, the number times we repeat the simulation and assessment of the CIs and point estimates for each set of parameters defined in the `grid`. With these two inputs, the simulation starts the first step by initialising a list (variable `o` in the code) in which the summary measures are ultimately stored. The second step is to extract a set of parameters (variable `pars` in the code) that is later used for the simulation and assessment of the CIs and point estimates. The third step consists of the initialization of two further variables (`av`, `p_accept` in the code) that store the performance measures of one simulated data set. Fourth, the following process is repeated N times:

1. Simulate a meta-analysis using the current parameters.
2. Estimate the between-study heterogeneity τ^2 using all heterogeneity estimation methods `"none"`, `"DL"`, `"PM"`, and `"REML"`.
3. For each estimate of τ^2, calculate the CIs and point estimates using all methods, i. e. , *Edgington*, *Fisher*, *Hartung-Knapp*, *Henmi-Copas*, and *Random Effects*.
4. Calculate the performance measures for resulting CI and point estimate.
5. Store the performance measures in variables `av` and `p_accept`.

Fifth, the summary measures are calculated from the estimates using the same heterogeneity estimation method and CI construction method. Sixth, the performance measures are stored in the output list `o`. Seventh, the simulation did not go through all parameters sets yet, the process goes back to step two with new parameters. Otherwise, the simulation process stops and returns the output list `o`.

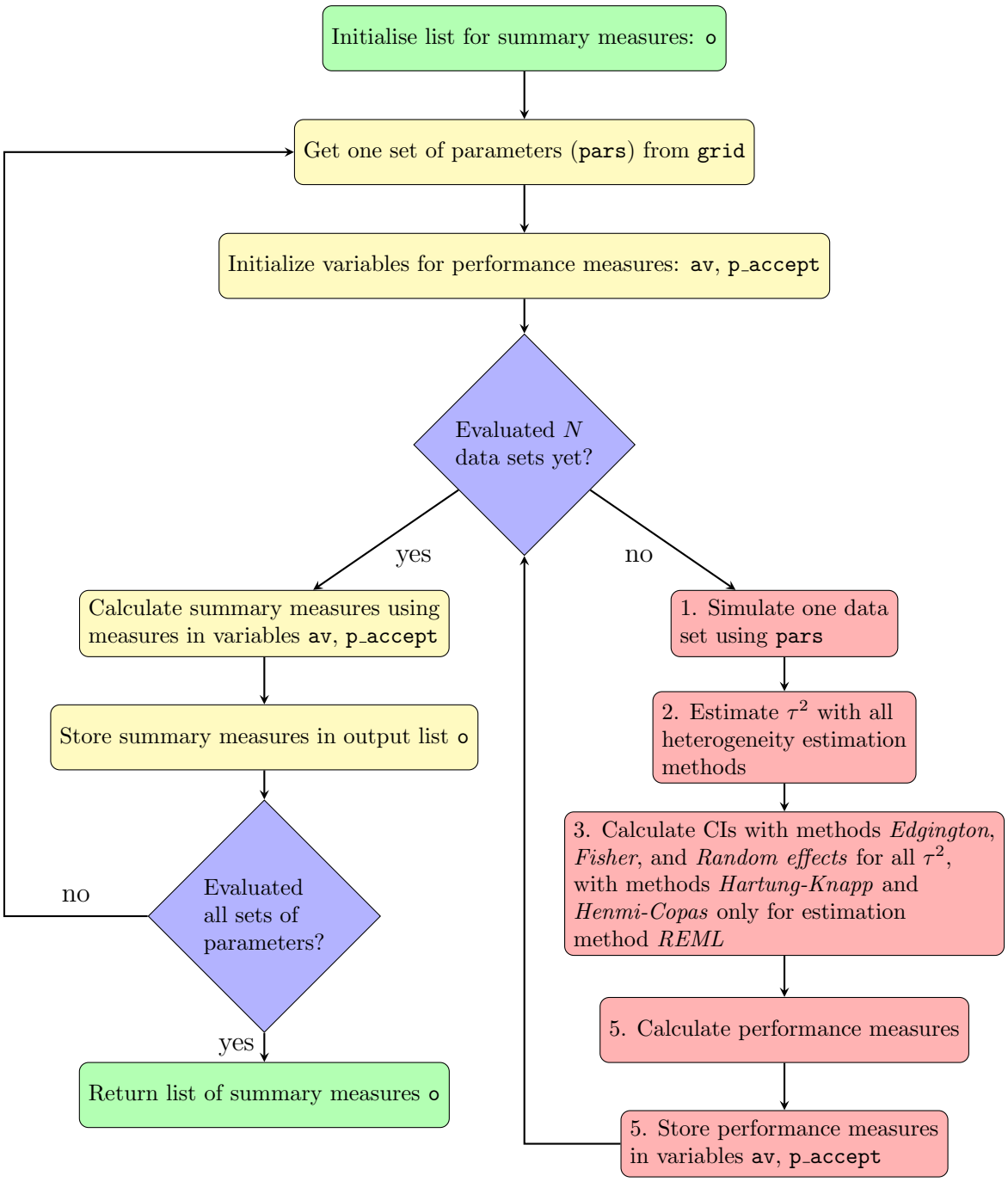

Figure 1: Flow chart of the simulation procedure. The colors indicate different scopes of the procedure, i. e. , green nodes are only executed once in total, yellow nodes are executed once for each set of parameters, and red nodes are executed N times for each set of parameters. The blue diamonds represent decision nodes.

2 Structure of the code base

The code is split into seven different files that all implement different aspects of the simulation process in Subsection 1.2. A list of these seven files is shown below.

- *simulation.R*
- *utils.R*
- *study_simulation.R*
- *studies2cis.R*
- *cis2measures.R*
- *measures2summary.R*
- *plot_results.R*

Each of these files contains functionality related to one particular step in the simulation. The main file is *simulation.R*, which sources all of the other scripts,with the exception of *plot_results.R*. The sourced files, i. e. , *utils.R*, *study_simulation.R*, *studies2cis.R*, *cis2measures.R*, and *measures2summary.R* only contain function definitions, no other objects. A more detailed overview of the contents of each file can be found in the following subsections.

2.1 simulation.R

This is the main file of the simulation. The code in this file does the following:

1. Load necessary libraries.
2. Source the other files that implement parts of the simulation.
3. Define the function `sim()`, which contains the logic for the green, yellow, and blue nodes shown in Figure 1.
4. Define the object `grid`, a `data.frame` containing all parameter combinations.
5. Set the number of iterations `N`.
6. Set the number of cores `cores`, the simulation should be run on.
7. Starts the simulation

All other functionality is defined in the other files or the *R*-package *confMeta*.

2.2 utils.R

The file `utils.R` defines two helper functions that are used in various other functions throughout the simulation.

`rep2()` A simple utility function that wraps the base *R* function `rep()`. The main difference is that it accepts vector inputs for arguments `each` or `times`.

`error_function()` A logger function that is only used within `tryCatch()` blocks. In case of errors, the function saves the error message to a file called *error.txt*. It also stores the current parameters (passed through argument `args`) to a file called *pars.rds*, and the last object received (argument `error_obj`) to a file called *error.rds*.

Note: parallel execution
The simulation uses the *foreach*-package (Microsoft and Weston, 2022) as a parallel backend with the *doRNG*-package for ensuring that the random numbers generated inside different `for` loops are independent and reproducible. This setup has, however, the drawback that without manual error handling, the simulation continues exection until all scenarios finished, even if some of the loops throw errors. This is, of course, rather undesirable since the simulation might run for a long time even if there are errors already in the first few iterations. Thus, all functions that simulate studies, calculate CIs, calculate performance measures, or calculate summary measures are wrapped in a `tryCatch()` block that calls `error_function()` in case of an error. The `error_function()` will then create a the file *error.txt*. The simulation then checks in each iteration whether this file exists, and if so, just writes `"skipped"` into the output list `"o"` without computing anything. This, however, means that before restarting the simulation, the *error.txt* file must be deleted.

2.3 study_simulation.R

The script *study_simulation.R* contains all functions related to the simulation of a meta-analysis data set. Such a data set consists of a number of individual studies that are simulated as described in the simulation protocol.
The file contains the definitions of the following functions.

`simRE()` Simulates a meta-analysis data set. The inputs for this function are `k`, `sampleSize`, `effect`, `I2`, `heterogeneity`, `dist`, and `large`. For a more detailed explanation on what these arguments mean, see Subsection 1.1 under **grid**. The function returns a matrix with columns `theta`, `se`, and `delta` which correspond to the observed study-specific effects, the corresponding standard errors, and the true study-specific effects, respectively.

`pAccept()` Calculates the acceptance probability for a given study under publication bias. The function is only called if the argument `bias` in function `simREbias()` is either `"moderate"` or `"strong"`.

`simREbias()` Simulates a meta-analysis data. Under the hood this function just calls `simRE()` in order to simulate meta-analyses with `k` studies. In a second step, the publication bias is incorporated if `bias` $\neq$ `"none"`. This is achieved by calculating the acceptance probabilities using `pAccept()` and sampling from a binomial distribution which studies to keep. If the number of accepted studies is smaller than `k`, this process is repeated until the meta-analysis has exactly `k` studies. In the third step, large studies are sampled and added to the meta-analysis if `large` $\neq 0$. It outputs a matrix with the same format as `simRE()` but it also attaches an attribute `p_accept` to the matrix indicating the mean acceptance probability for each study in the meta-analysis. This attribute contains the value that is stored in as a performance measure in the variable `p_accept` mentioned in Figure 1.

`sim_effects()` This is the main function for simulating studies. Internally, it calls `simREbias()` and if that errors, runs `error_function()` such that the error is documented.

In the simulation, we exclusively use the function `sim_effects()` as shown below:

```r
# define the grid
grid <- expand.grid(
    # sample size of trial
    sampleSize = 50,
    # average effect, impacts selection bias
    effect = c(0.1, 0.2, 0.5),
    # Higgin's I^2 heterogeneity measure
    I2 = c(0, 0.3, 0.6, 0.9),
    # number of studies
    k = c(3, 5, 10, 20, 50),
    # The heterogeneity model that the studies are simulated from
    heterogeneity = c("additive"),
    # distribution
    dist = c("Gaussian", "t"),
    # bias
    bias = c("none", "moderate", "strong"),
    # number of large studies
    large = c(0, 1, 2),
    stringsAsFactors = FALSE
)
# get a set of parameters
print(pars <- grid[324, ])

##     sampleSize effect  I2 k heterogeneity dist   bias large
## 324         50    0.5 0.9 5      additive    t strong     0

# simulate studies
print(studies <- sim_effects(pars, i = 1))

##          theta        se     delta
## [1,] 0.7169084 0.1956171 0.2514940
## [2,] 0.6030504 0.1936712 1.0862761
## [3,] 0.4720733 0.1947064 0.3638587
## [4,] 1.2048328 0.1852003 1.0694965
## [5,] 0.9940716 0.1958838 0.9317017
## attr(,"heterogeneity")
## [1] "additive"
## attr(,"effect")
## [1] 0.5
## attr(,"p_accept")
## [1] 0.9787999
```

2.4 studies2cis.R

The file *studies2cis.R* contains the functionality to convert a simulated meta-analysis data set into CIs, the point estimate, and the maximum p-value function, if applicable. Since we needed

to implement this functionality for all different methods, this file contains a comparatively large number of functions. Thus, instead of listing every single function and describing what it does, I will rather focus on the structure of how the necessary function is implemented.

2.4.1 Estimating heterogeneity

In order to estimate the between-study heterogeneity τ^2 with all of the heterogeneity estimation method, the simulation includes the function `get_tau2()`.

2.4.2 Reference methods

Since we use external libraries for the calculation of CIs and point estimates for methods *Random effects* (*meta*), *Hartung-Knapp* (*meta*) and *Henmi-Copas* (*metafor*), the file contains wrapper functions for these libraries. All of these libraries follow the same pattern, i. e. , they provide a function of the simulated estimates and the simulated standard estimates in the meta-analysis data set and return a resulting object, from which the CI and point estimate can be extracted. Thus, the extraction of the CI and estimate is essentially only a function of the desired method, the simulated study effects, and the corresponding standard errors.

This idea is implemented in the function `get_classic_intervals()`, which takes three arguments, `methods`, `thetahat`, `se`. Here, `methods` can be any combination of `"hk_ci"` (for *Hartung-Knapp*), `"hc_ci"` (for *Henmi-Copas*), `"reml_ci"` (for *Random effects*). The arguments `thetahat` and `se` are just the simulated studies and the corresponding standard errors from the simulation step. Internally, `get_classic_intervals()` determines, what kind of object it needs to fit for each of the different `methods` and how to extract the desired statistics.

The internal implementation consists of functions `get_classic_obj_xx()` where `xx` is either `hc`, `reml` or `hc`. There is also a function for `bm`, which corresponds to the *bayesmeta* method, but that is currently not used. All of these functions accept an argument `thetahat` and `se` and return an object. Moreover, these functions are summarised in `get_classic_obj()` which, depending on the `method` argument calls one of the former functions.

Similarly, there is a function `get_classic_interval()`, which calls, depending on the `method` argument, one of the functions `get_classic_yy_xx()`, where `yy` is either `ci` (for CI) or `pi` (for prediction interval (PI)) and `xx` is again one of `hc`, `hk`, or `reml`. This can be used as shown in the following code chunk.

```
# fit an object using Hartung-Knapp and extract the confidence interval
get_classic_interval(
    # alternatively, this could also be "hk_pi" for prediction interval
    method = "hk_ci",
    obj = get_classic_obj(
        method = "hk",
        thetahat = studies[, "theta"],
        se = studies[, "se"]
    )
)
```

```
##     lower     upper  estimate
## 0.4293056 1.1731024 0.8012040
```

2.4.3 Methods of interest

The implementation of the p-value functions and the algorithm that calculates the CIs and point estimates can be found in the package *confMeta*. However, this simulation also provides several wrappers for the confMeta functionality.
The most two most basic functions are `get_p_value_functions()`, which returns a list of all the p-value functions that should be present in the simulation, and `get_p_value_args()`, which returns a list of argument configurations that are passed to the p-value functions.
The function `get_new_ci_gamma()` calculates a grid from the lists returned by the two functions above, loops over this list and calculates for each argument/function combination the CI and the point estimate.
The function `get_new_intervals()` combines all the above functions and serves as the main function to calculates everything of interest related to the methods of interest.

2.4.4 Assembling the output

The main function for this step in the simulation is called `calc_ci()` which itself wraps `sim2CIs()`. The latter calculates the CIs and point estimates for the reference methods and the methods of interests for all possible argument configurations and returns a list with elements

CIs A `data.frame` containing the confidence interval for each method/argument combination. It also contains the variables `is_new`, `is_ci`, and `is_pi`, which all contain logical values `TRUE` or `FALSE`. These are important in the next step to determine which performance maesures need to be calculated for a given CI.

estimates A `data.frame` containing the point estimates for each method/argument combination.

p_max A `data.frame` containing the maximum of the p-value function for each method/argument combination.

model The simulation model that the meta-analysis data set was simulated from. This is the same as `pars$heterogeneity` and is independent of the `heterogeneity` argument of any p-value function used here.

theta The simulated effects of the individual studies in the meta-analysis data set. This is the same as the column `"theta"` from the meta-analysis data set.

delta The study-specific effects in the meta-analysis data set. This is the same as the column `"delta"` from the meta-analysis data set.

effect The true effect used for the simulation of the meta-analysis data set. This is the same as `pars$effect` for the current iteration.

2.5 cis2measures.R

The file *cis2measures.R* contains functionality to convert the output of function `calc_ci()` to the performance measures stated in Subsection 1.1. The main function here is called `calc_measures()`, which wraps `CI2measures()` and, in case of an error, runs the `error_function()`.

Internally, we first determine, which performance measure needs to be calculated for a given CI. This is done by looking at the combination of booleans in the variables `is_new`, `is_ci`, and `is_pi` in function `get_measures()`. This function returns a list, that contains all the functions needed to calculate the performance measures for the given CI. Currently, we use the following functions to calculate the performance measures:

`calc_coverage_true()` Calculates the coverage of the true effect θ. Thus, the return is 1, if $\theta \in$ CI and 0 otherwise. This is done for all CIs.

`calc_width()` Calculates the width of the CI. This is done for all CIs.

`calc_score()` Calculates the score of the CI. This is done for all CIs.

`calc_n()` Calculates the number of confidence intervals. This is only done for CIs from methods of interest, i. e. , where `is_new == TRUE`.

`calc_estimate()` Simply returns the point estimate itself.

The main output of the function `calc_ci()` is a `data.frame` with columns

value The value of the performance measure.

measure The name of the performance measure.

method The name of the CI construction method.

is_ci Whether the current interval is a CI.

is_pi Whether the current interval is a PI.

is_new Whether the construction method is a method of interest.

2.6 measures2summary.R

This script contains the functionality needed to summarise the performance measures to summary measures. Again, the wrapper function is `calc_summary_measures()` which calls `get_summary_measures()` or, in case of an error, `error_function()`.

Internally, we use the following functions to summarise the performance measures.

`get_mean_stats()` Calculates the mean across all performance measures of the same method.

`get_gamma_stats()` Calculates the maximum, third quartile, median, mean, first quartile, and minimum across the performance measure n, i. e. , the number of CIs.

`get_n_stats()` Calculates how often the number of CIs is equal to $\{0, \ldots, 9\}$ or > 9.

`get_bias_var_stats()` Calculates the mean squared error, bias and variance from the point estimates of each method.

`get_paccept_stats()` Calculates the mean of the average acceptance probability. This is only done in case the current parameters have a bias which is not `"none"`.

2.7 plot_results.R

This script does is not a part of the simulation itself, but it visualizes the results of the simulation used to visualise the results.

References

DerSimonian, R. and Laird, N. (1986). Meta-analysis in clinical trials. *Controlled Clinical Trials*, 7(3):177–188. 2

Edgington, E. S. (1972). An Additive Method for Combining Probability Values from Independent Experiments. *The Journal of Psychology*, 80(2):351–363. Publisher: Routledge _eprint: https://doi.org/10.1080/00223980.1972.9924813. 2

Fisher, R. A. (1932). *Statistical Methods for Research Workers*. Oliver & Boyd, Edinburgh, 4 edition. 2

Harville, D. A. (1977). Maximum Likelihood Approaches to Variance Component Estimation and to Related Problems. *Journal of the American Statistical Association*, 72(358):320–338. Publisher: [American Statistical Association, Taylor & Francis, Ltd.]. 2

Henmi, M. and Copas, J. B. (2010). Confidence intervals for random effects meta-analysis and robustness to publication bias. *Statistics in Medicine*, 29(29):2969–2983. 2

IntHout, J., Ioannidis, J. P., and Borm, G. F. (2014). The Hartung-Knapp-Sidik-Jonkman method for random effects meta-analysis is straightforward and considerably outperforms the standard DerSimonian-Laird method. *BMC Medical Research Methodology*, 14(25). 2

Microsoft and Weston, S. (2022). *foreach: Provides Foreach Looping Construct*. R package version 1.5.2. 6

Paule, R. C. and Mandel, J. (1982). Consensus Values and Weighting Factors. *Journal of Research of the National Bureau of Standards (1977)*, 87(5):377–385. 2