Monte Carlo simulation studies with mcstatsim

Imad El Badisy

2026-11-03

Contents

1 Overview **1**

2 A first simulation **2**

3 The `sim_func` contract **3**

4 Reproducibility **3**

5 When a replication fails **4**

6 Resuming a long run **4**

7 From replicates to a results table **4**
7.1 Rounding 5

8 The performance-measure functions **5**

9 Numerical diagnostics **6**

10 Visual diagnostics **6**

11 Case study: robust location estimators **9**

12 Practical notes **10**

13 Reference **10**

1 Overview

A Monte Carlo simulation study repeats the same experiment many times under several data-generating scenarios, then summarises how an estimator or a test behaves across those repetitions. `mcstatsim` organises such a study around a single higher-order function, `runsim()`:

- you write the **one replication** as an ordinary R function of its parameters,
- you list the **scenarios** as the rows of a data frame,
- `runsim()` maps the function over `scenarios x replications`, in parallel, and returns one tidy data frame of replicate-level results.

Everything else in the package operates on that data frame: `summarise_sim()` collapses it into a performance table, the `calc_*` functions compute the individual performance measures with Monte Carlo standard errors, and a handful of helpers and plots support diagnosis.

The only hard dependency is `functionals`. `ggplot2` is used by the plotting helpers and is optional; this vignette skips the figures if it is not installed.

```
library(mcstatsim)
```

2 A first simulation

Suppose we want to check the bias of the sample mean as an estimator of a normal mean. One replication draws a sample and returns the estimate:

```
sim_mean <- function(n, mu) {
  x <- rnorm(n, mean = mu)
  data.frame(estimate = mean(x))
}
```

The scenarios are a data frame whose column names match the arguments of `sim_func`:

```
grid <- expand.grid(n = c(10, 50), mu = c(0, 1))
grid
#>    n mu
#> 1 10  0
#> 2 50  0
#> 3 10  1
#> 4 50  1
```

Run 1000 replications per scenario:

```
res <- runsim(
  n = 1000,
  grid_params = grid,
  sim_func = sim_mean,
  seed = 101,
  num_cores = sim_cores,
  show_progress = FALSE
)
head(res)
#>    n mu estimate ID
#> 1 10  0  0.12649  1
#> 2 50  0 -0.04246  1
#> 3 10  1  1.57291  1
#> 4 50  1  0.81448  1
#> 5 10  0  0.53932  2
#> 6 50  0 -0.08296  2
```

The result has one row per replication per scenario. `runsim()` prepends the scenario columns (`n`, `mu`) that `sim_func` did not itself return, and adds an `ID` column with the replication index. Now summarise:

```
summarise_sim(
  res,
  by = c("n", "mu"),
  measures = list(bias = ~ calc_bias(estimate, mu[1]))
)
#>    n mu n_sim    bias bias_mcse
#> 1 10  0  1000  0.0066    0.0097
#> 2 10  1  1000  0.0061    0.0100
#> 3 50  0  1000 -0.0043    0.0044
#> 4 50  1  1000  0.0040    0.0045
```

`bias` is the estimated bias and `bias_mcse` its Monte Carlo standard error: the bias is within a couple of standard errors of zero in every scenario, as it should be for an unbiased estimator.

3 The `sim_func` contract

`sim_func` must

1. take arguments that are a subset of the `grid_params` column names,
2. carry out exactly **one** replication, and
3. return a `data.frame` (usually one row; several rows are allowed and are all kept).

It should **not** set a seed internally (see the next section) and should **not** draw plots. Anything constant within a scenario, such as the true parameter value, can be returned as a column so it is available to `summarise_sim()`:

```
sim_ci <- function(n, mu) {
  x <- rnorm(n, mean = mu)
  est <- mean(x)
  se <- sd(x) / sqrt(n)
  crit <- qt(0.975, df = n - 1)
  data.frame(
    est   = est,
    lo    = est - crit * se,
    hi    = est + crit * se,
    truth = mu           # constant within a scenario, handy for summarise_sim()
  )
}
```

The grid is normally built with `expand.grid()` for a full factorial design, but any data frame works, so you can also supply a hand-picked list of scenarios.

4 Reproducibility

Passing `seed` makes a run reproducible **and** independent of the number of cores. Internally `runsim()` creates one independent `"L'Ecuyer-CMRG"` random number stream per job and assigns streams by position, so a given `(replication, scenario)` cell always draws from the same stream:

```
a <- runsim(200, grid, sim_mean, seed = 7, num_cores = 1, show_progress = FALSE)
b <- runsim(200, grid, sim_mean, seed = 7, num_cores = 2, show_progress = FALSE)
identical(a$estimate, b$estimate)
#> [1] TRUE
```

The seed used is stored on the result:

```
attr(a, "seed")
#> [1] 7
```

Without `seed`, `runsim()` leaves the session RNG untouched and successive runs differ. Do not call `set.seed()` inside `sim_func`: it would tie every replication of a scenario to the same stream and destroy the independence the Monte Carlo standard errors rely on.

5 When a replication fails

Long studies often have a few replications where a model does not converge or a solver errors. By default `runsim()` stops on the first error (`on_error = "stop"`). With `on_error = "warn"` or `"omit"` the failing jobs are dropped and recorded instead:

```
flaky <- function(n, mu) {
  x <- rnorm(n, mu)
  if (mean(x) < -0.5) stop("solver failed to converge")
  data.frame(est = mean(x))
}

res_f <- runsim(300, expand.grid(n = 15, mu = c(-1, 0, 1)), flaky,
                seed = 4, num_cores = sim_cores, show_progress = FALSE,
                on_error = "omit")
nrow(res_f)
#> [1] 599
```

The failures are attached as an attribute and `failure_summary()` aggregates them per scenario:

```
failure_summary(res_f)
#>    n mu n_failed failure_rate             example_error
#> 1 15 -1      293       0.9767 solver failed to converge
#> 2 15  0        8       0.0267 solver failed to converge
```

Warnings raised inside `sim_func` are captured too (parallel back ends would otherwise swallow them) and land in `attr(res_f, "warnings")`.

6 Resuming a long run

For studies that take hours, pass `save_path`. `runsim()` then runs the replications in chunks and rewrites an `.rds` checkpoint after each chunk. Calling `runsim()` again with the same `save_path` and matching `n`, `grid_params` and `seed` loads the completed replications and runs only the rest:

```
ckpt <- tempfile(fileext = ".rds")

# first call: imagine this is interrupted partway through
r1 <- runsim(400, grid, sim_mean, seed = 9, save_path = ckpt,
             checkpoint_every = 50, num_cores = sim_cores, show_progress = FALSE)

# a later call with the same settings resumes instead of restarting
r2 <- runsim(400, grid, sim_mean, seed = 9, save_path = ckpt,
             checkpoint_every = 50, num_cores = sim_cores, show_progress = FALSE)

identical(r1$estimate, r2$estimate)
#> [1] TRUE
file.remove(ckpt)
#> [1] TRUE
```

Because streams are keyed by position, a resumed run reproduces an uninterrupted one exactly.

7 From replicates to a results table

`summarise_sim(data, by, measures)` groups the replicate-level data by the scenario columns in `by` and evaluates each element of `measures` per group. A measure is either

- a one-sided formula whose right-hand side is evaluated with the group's columns in scope, e.g. `~ calc_coverage(lo, hi, truth[1])`, or
- a function of the group data frame, e.g. `function(d) calc_coverage(d$lo, d$hi, d$truth[1])`.

A quantity that is constant within a scenario (a true parameter value) is a column of length equal to the number of replications, so index it with `[1]`.

```
res_ci <- runsim(1000, expand.grid(n = c(15, 60), mu = 0), sim_ci,
                 seed = 202, num_cores = sim_cores, show_progress = FALSE)

perf <- summarise_sim(
  res_ci,
  by = "n",
  measures = list(
    bias     = ~ calc_bias(est, truth[1]),
    coverage = ~ calc_coverage(lo, hi, truth[1]),
    width    = ~ calc_width(lo, hi)
  )
)
perf
#>    n n_sim    bias bias_mcse coverage coverage_mcse  width width_mcse
#> 1 15  1000  0.0039    0.0083    0.955        0.0066 1.0772     0.0065
#> 2 60  1000 -0.0045    0.0042    0.953        0.0067 0.5123     0.0015
```

Each `calc_*` measure contributes an estimate column and a `*_mcse` column. The `n_sim` column reports the number of distinct replications behind each row.

7.1 Rounding

The result-producing functions take a `digits` argument (default `4`). It rounds the returned numbers for readability; `summarise_sim()` rounds the measure columns only, leaving the `by` columns and `n_sim` alone. Pass `digits = NULL` for full precision:

```
summarise_sim(res_ci, by = "n",
              measures = list(bias = ~ calc_bias(est, truth[1])),
              digits = NULL)
#>    n n_sim    bias bias_mcse
#> 1 15  1000  0.0039    0.0083
#> 2 60  1000 -0.0045    0.0042
```

Measures built from `calc_*` are already rounded to those functions' own `digits`; write `~ calc_bias(est, truth[1], digits = NULL)` inside the measure if you need more precision than that.

8 The performance-measure functions

Each `calc_*` function takes vectors of replicate-level quantities and returns a named list with the measure and its Monte Carlo standard error.

Function	Inputs	Returns
`calc_bias(est, true)`	estimates, true value	`bias`, `bias_mcse`
`calc_variance(est)`	estimates	`variance`, `variance_mcse`
`calc_mse(est, true)`	estimates, true value	`mse`, `mse_mcse`
`calc_rmse(est, true)`	estimates, true value	`rmse`, `rmse_mcse`
`calc_coverage(lo, hi, true)`	interval bounds, true value	`coverage`, `coverage_mcse`
`calc_width(lo, hi)`	interval bounds	`width`, `width_mcse`

Function	Inputs	Returns
`calc_rejection_rate(p, alpha)`	p-values	`rejection_rate`, `rejection_rate_mcse`
`calc_relative_bias(est, true)`	estimates, non-zero true value	`rel_bias`, `rel_bias_mcse`
`calc_relative_mse(est, true)`	estimates, non-zero true value	`rel_mse`, `rel_mse_mcse`
`calc_relative_rmse(est, true)`	estimates, non-zero true value	`rel_rmse`, `rel_rmse_mcse`

They can be used directly, outside a simulation:

```
set.seed(3)
theta_hat <- rnorm(500, mean = 2.1, sd = 0.4)
calc_bias(theta_hat, true_param = 2)
#> $bias
#> [1] 0.121
#>
#> $bias_mcse
#> [1] 0.0185
calc_rmse(theta_hat, true_param = 2)
#> $rmse
#> [1] 0.4296
#>
#> $rmse_mcse
#> [1] 0.0166
```

9 Numerical diagnostics

`check_estimates()` scans the numeric columns of a result for `NA`, `NaN` and `Inf`, per scenario, so a handful of blown-up replications are not folded silently into a summary:

```
divergent <- function(n, mu) {
  x <- rnorm(n, mu)
  data.frame(est = mean(x), ratio = mu / (mu - 1))   # Inf when mu == 1
}
res_d <- runsim(50, expand.grid(n = 20, mu = c(0, 1, 2)), divergent,
                seed = 5, num_cores = sim_cores, show_progress = FALSE)
check_estimates(res_d, by = "mu")
#>   mu column n_bad n_total prop_bad
#> 1  1  ratio    50      50        1
```

`mcse_target()` answers "how many replications do I need?". Given the Monte Carlo standard error reached so far and the replication count that produced it, it returns the count needed for a target standard error:

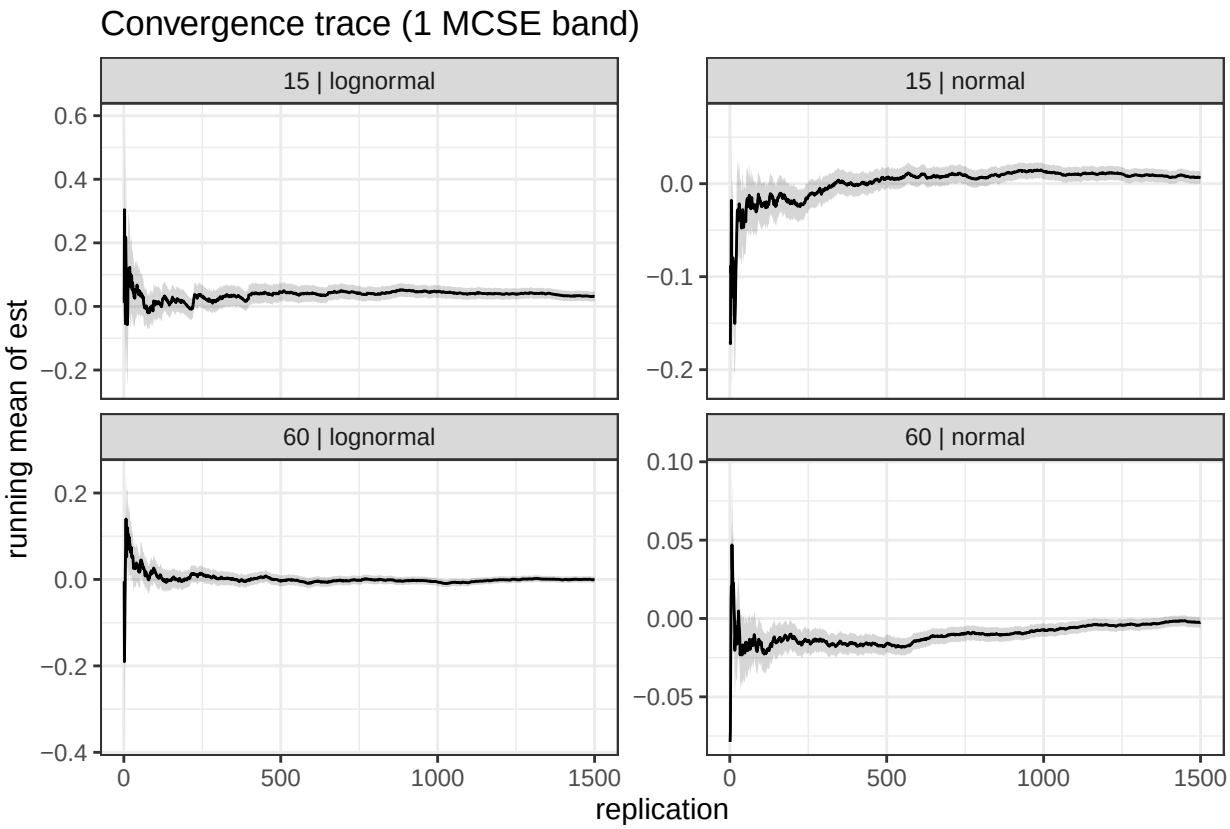
```
mcse_target(mcse_current = perf$coverage_mcse, n_current = perf$n_sim,
            target = 0.005)
#> [1] 1743 1796
```

10 Visual diagnostics

The three plotting helpers return `ggplot` objects and require the suggested `ggplot2` package.

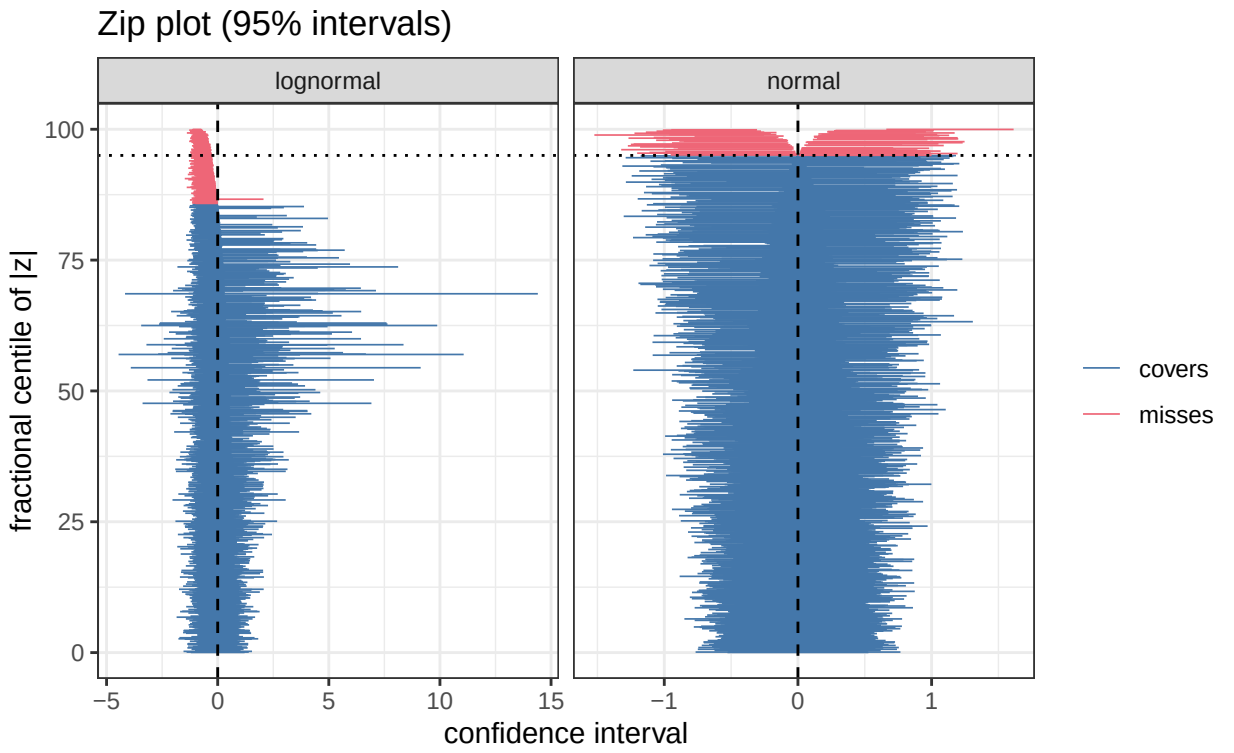
`plot_convergence()` shows the running mean of a quantity against the replication index, with a Monte Carlo standard-error band, one panel per scenario. It shows whether a run used enough replications:

```
plot_convergence(res2, "est", by = c("n", "dist"))
```



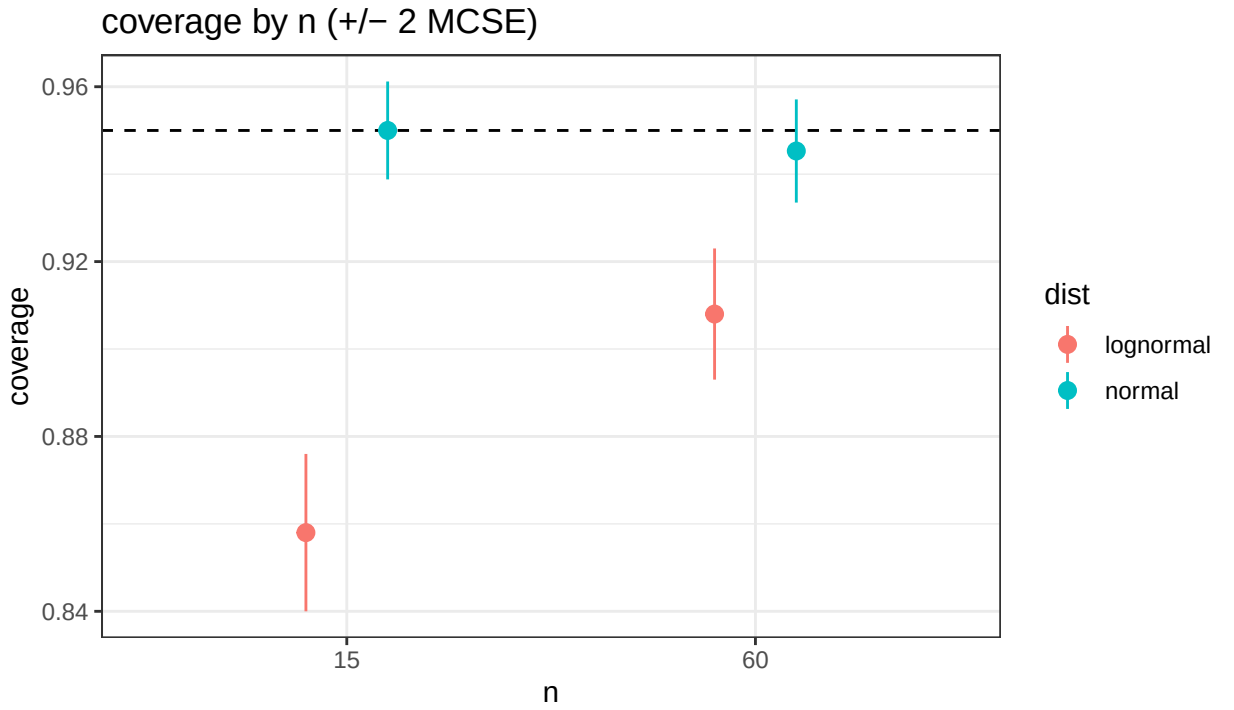
`plot_zip()` draws the zip plot of Morris, White and Crowther (2019). Within each scenario the confidence intervals are sorted by how extreme their z-statistic is and coloured by whether they cover; a well-calibrated procedure fills the panel up to the nominal line in the "covers" colour:

```
plot_zip(res2[res2$n == 15, ], "lo", "hi", "truth", by = "dist")
```



`plot_performance()` turns a `summarise_sim()` table into a point-range of a measure and its Monte Carlo error across the design:

```
plot_performance(perf2, "coverage", x = "n", colour = "dist", ref = 0.95)
```



11 Case study: robust location estimators

We compare three estimators of a symmetric centre, the sample mean, the 20% trimmed mean and the median, when a fraction `eps` of the data is replaced by a wide contaminating normal. The true centre is 0 for all three.

```
sim_location <- function(n, eps, estimator) {
  contaminated <- rbinom(n, 1, eps) == 1
  x <- rnorm(n)
  x[contaminated] <- rnorm(sum(contaminated), sd = 8)

  est <- switch(estimator,
    mean    = mean(x),
    trimmed = mean(x, trim = 0.2),
    median  = median(x)
  )
  data.frame(estimate = est, truth = 0)
}

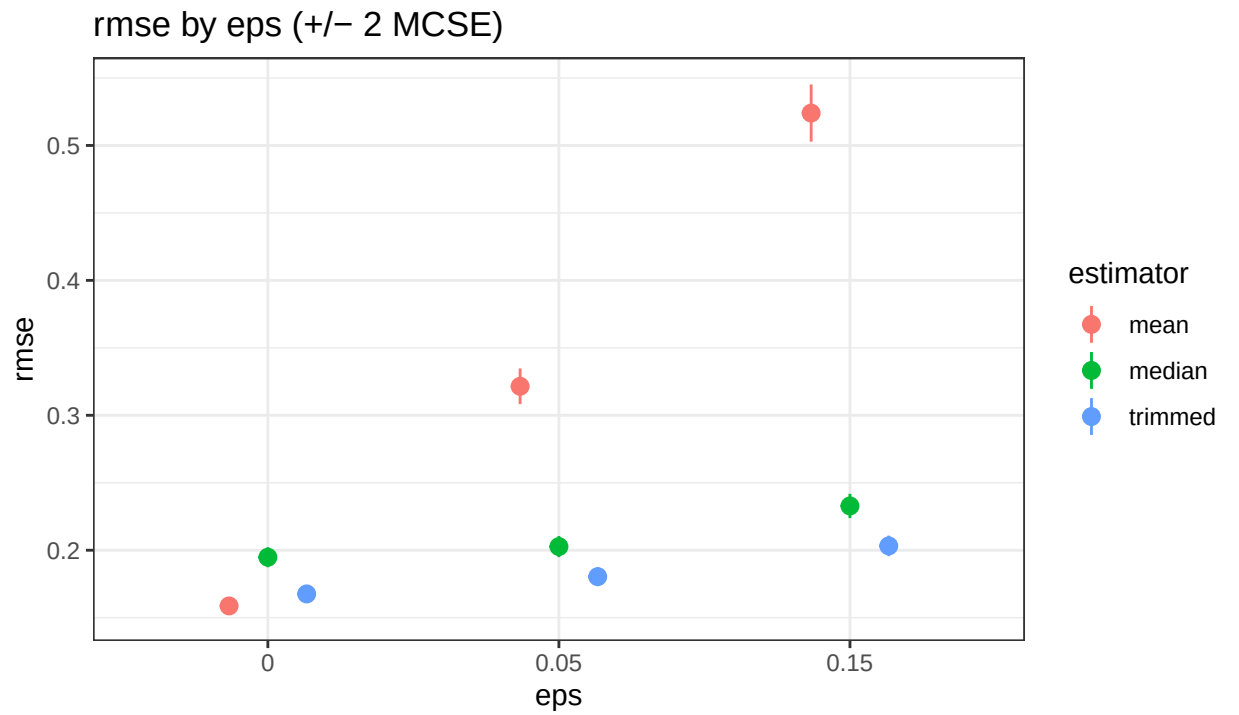
design <- expand.grid(
  n = 40,
  eps = c(0, 0.05, 0.15),
  estimator = c("mean", "trimmed", "median"),
  stringsAsFactors = FALSE
)
```

```
sim_res <- runsim(2000, design, sim_location, seed = 2024,
                  num_cores = sim_cores, show_progress = FALSE)
```

```
perf_loc <- summarise_sim(
  sim_res,
  by = c("estimator", "eps"),
  measures = list(
    bias = ~ calc_bias(estimate, truth[1]),
    rmse = ~ calc_rmse(estimate, truth[1])
  )
)
perf_loc[order(perf_loc$eps, perf_loc$rmse), ]
#>   estimator  eps n_sim    bias bias_mcse   rmse rmse_mcse
#> 1      mean 0.00  2000 -0.0025    0.0035 0.1587    0.0031
#> 7   trimmed 0.00  2000  0.0003    0.0037 0.1676    0.0032
#> 4    median 0.00  2000  0.0001    0.0044 0.1948    0.0037
#> 8   trimmed 0.05  2000  0.0000    0.0040 0.1804    0.0035
#> 5    median 0.05  2000 -0.0053    0.0045 0.2027    0.0039
#> 2      mean 0.05  2000 -0.0031    0.0072 0.3215    0.0066
#> 9   trimmed 0.15  2000  0.0035    0.0045 0.2032    0.0038
#> 6    median 0.15  2000 -0.0115    0.0052 0.2328    0.0045
#> 3      mean 0.15  2000 -0.0051    0.0117 0.5240    0.0106
```

All three estimators are essentially unbiased (the contamination is symmetric), but their root mean squared error diverges as contamination grows: at `eps = 0.15` the sample mean is far worse than the trimmed mean and the median, while at `eps = 0` the mean is (slightly) the most efficient. This bias-free efficiency trade-off is the classic argument for robust estimators.

```r
plot_performance(perf_loc, "rmse", x = "eps", colour = "estimator")
```



12 Practical notes

- **Cores.** `num_cores` defaults to one less than the machine's core count. On Windows the parallel back end still works; very short replications may run faster with `num_cores = 1` because of worker start-up cost.
- **Chunking.** `checkpoint_every` trades disk writes against lost work on a crash. Use a small value for expensive replications, a larger one for cheap ones.
- **RNG hygiene.** Let `runsim()` own the randomness through `seed`. If a replication needs several independent random components, just draw them in sequence inside `sim_func`; they share the job's stream and stay independent across replications.
- **Keep plotting out of `sim_func`.** Return numbers from `sim_func` and draw figures afterwards from the result data frame.
- **Standard errors.** Report the `*_mcse` columns alongside every performance estimate, and use `mcse_target()` before committing to a replication count.

13 Reference

Morris, T. P., White, I. R., and Crowther, M. J. (2019). Using simulation studies to evaluate statistical methods. *Statistics in Medicine*, 38(11), 2074-2102.