Package: medsim (via r-universe)

October 25, 2026

Type Package

Title Simulation Infrastructure for Mediation Analysis

Version 0.6.0.9000

Maintainer Davood Tofighi <dtofighi@gmail.com>

Description Provides standardized infrastructure for conducting Monte Carlo simulation studies in mediation analysis. Features include environment-aware execution (local vs HPC cluster), parallel processing with progress reporting, ground truth caching, automated result analysis, and publication-ready visualization. Designed to work seamlessly with the mediationverse ecosystem: 'medfit', 'probmed', 'RMediation', and 'medrobust'.

License GPL (>= 3)

URL https://data-wise.github.io/medsim/, https://github.com/Data-Wise/medsim

BugReports https://github.com/Data-Wise/medsim/issues

Depends R (>= 4.1.0)

Imports parallel, pbapply, ggplot2, dplyr

Suggests medfit (>= 0.2.0), medrobust, probmed, S7, RMediation, mice, mitml, mvtnorm, MASS, tictoc, here, renv, pak, testthat (>= 3.0.0), knitr, rmarkdown, quarto, covr, withr, gridExtra, patchwork, RColorBrewer

Remotes Data-Wise/medrobust, Data-Wise/probmed

Config/Needs/website pkgdown, quarto

Encoding UTF-8

Language en-US

Roxygen list(markdown = TRUE)

VignetteBuilder quarto

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

RoxygenNote 7.3.3

Repository https://data-wise.r-universe.dev

Date/Publication 2026-10-25 03:28:07 UTC

RemoteUrl https://github.com/Data-Wise/medsim

RemoteRef dev

RemoteSha a2659822b9be52af0d85ee925f15216773d74536

Contents

medsim_amputate . . . 3
medsim_analyze . . . 4
medsim_analyze_coverage . . . 6
medsim_analyze_performance . . . 8
medsim_analyze_power . . . 9
medsim_audit_results . . . 10
medsim_cache_clear . . . 12
medsim_cache_exists . . . 13
medsim_cache_info . . . 14
medsim_cache_init . . . 15
medsim_cache_list . . . 15
medsim_cache_load . . . 16
medsim_cache_save . . . 18
medsim_check_results . . . 19
medsim_combine_chunks . . . 20
medsim_compare_configs . . . 22
medsim_compare_methods . . . 22
medsim_config . . . 24
medsim_detect_cores . . . 27
medsim_detect_environment . . . 28
medsim_estimand . . . 29
medsim_estimate_speedup . . . 30
medsim_figures . . . 31
medsim_get_optimal_cores . . . 32
medsim_method_bounds . . . 33
medsim_method_gauge . . . 34
medsim_method_ipw . . . 35
medsim_method_mbco_mi . . . 36
medsim_method_mc_ci . . . 37
medsim_method_pmed_mbco . . . 38
medsim_method_sobol . . . 39
medsim_nsim_for_mcse . . . 40
medsim_plot_combined_panel . . . 41
medsim_plot_coverage . . . 42
medsim_plot_error_boxplot . . . 43
medsim_plot_se_vs_estimate . . . 45
medsim_plot_timing . . . 45

medsim_rnonnormal . . . 46
medsim_run . . . 47
medsim_run_chunk . . . 50
medsim_run_parallel . . . 52
medsim_scenario . . . 54
medsim_scenario_dm . . . 56
medsim_scenario_gauge . . . 57
medsim_scenario_missing . . . 58
medsim_scenario_missing_grid . . . 59
medsim_scenario_numeric . . . 60
medsim_scenario_pmed . . . 61
medsim_scenario_sobol . . . 62
medsim_scenarios_mediation . . . 63
medsim_summarize_branch_switch . . . 65
medsim_table_accuracy . . . 65
medsim_table_comparison . . . 66
medsim_table_coverage . . . 67
medsim_table_performance . . . 68
medsim_table_power . . . 68
medsim_table_timing . . . 69
medsim_tables . . . 70
medsim_tables_workflow . . . 70
medsim_validate_scenario . . . 71
medsim_workflow . . . 72
medsim_write_submit_script . . . 73
medsim_write_table . . . 74
print.medsim_analysis . . . 75
print.medsim_comparison . . . 75
print.medsim_config . . . 76
print.medsim_coverage . . . 76
print.medsim_power . . . 77
print.medsim_results . . . 77
print.medsim_scenario . . . 78
print.medsim_table . . . 78
summary.medsim_results . . . 79

Index **80**

`medsim_amputate` *Insert missing values under MCAR / MAR / MNAR*

Description

Thin, documented amputer aligning `mice::ampute` semantics to the medsim DGM contract. Returns the input frame with `NA`s inserted in `target` column(s); column names/order are preserved (DGM data contract).

Usage

```
medsim_amputate(
  data,
  target,
  mechanism = c("MCAR", "MAR", "MNAR"),
  prop = 0.2,
  predictors = NULL,
  weights = NULL,
  type = c("RIGHT", "LEFT", "MID", "TAIL")
)
```

Arguments

data	data.frame with columns X, M, Y (+ optional covariates).
target	Character vector of column(s) to set NA (e.g. "M", or c("M","Y")).
mechanism	One of "MCAR", "MAR", "MNAR". MCAR: constant prob; MAR: logistic on observed predictors; MNAR: logistic including the target itself (self-mechanism).
prop	Target overall missingness proportion in target.
predictors	Columns driving missingness for MAR/MNAR (defaults to all non-target).
weights	Optional named weights for the missingness logistic.
type	Amputation tail type, mice::ampute-style: one of "RIGHT","LEFT","MID","TAIL".

Value

data with NAs inserted in target; names/order preserved.

Examples

```
set.seed(1)
d <- data.frame(X = rnorm(300), M = rnorm(300), Y = rnorm(300))
d_mar <- medsim_amputate(d, target = "M", mechanism = "MAR", prop = 0.3)
mean(is.na(d_mar$M))
```

---

medsim_analyze *Analyze Simulation Results*

Description

Computes comprehensive accuracy metrics for simulation results including bias, mean absolute error (MAE), root mean squared error (RMSE), and relative efficiency.

Usage

```
medsim_analyze(results, metrics = "all", by_scenario = TRUE)
```

Arguments

`results`	A medsim_results object from `medsim_run()`
`metrics`	Character vector: which metrics to compute. Options: • "bias": Mean estimation error • "mae": Mean absolute error • "rmse": Root mean squared error • "median_ae": Median absolute error • "max_ae": Maximum absolute error • "relative_bias": Bias as percentage of truth • "all": Compute all metrics (default)
`by_scenario`	Logical: Compute metrics separately for each scenario (default TRUE)

Details

Metrics Computed:

For each estimated parameter with known ground truth:

- **Bias**: Mean estimation error (estimate - truth)
- **MAE**: Mean absolute error, robust to outliers
- **RMSE**: Root mean squared error, penalizes large errors
- **Median AE**: Median absolute error, very robust
- **Max AE**: Maximum absolute error, identifies worst case
- **Relative Bias**: Bias as percentage of true value

Interpretation:

- Bias close to 0: unbiased estimator
- MAE/RMSE small: accurate estimator
- RMSE > MAE: presence of outliers/large errors
- Relative bias: standardized bias metric

Value

A list with class "medsim_analysis" containing:

- `accuracy`: data.frame with accuracy metrics
- `by_scenario`: data.frame with metrics by scenario (if by_scenario=TRUE)
- `summary`: Overall summary statistics

See Also

`medsim_analyze_coverage()`, `medsim_compare_methods()`

Examples

```
## Not run:
# Run simulation
results <- medsim_run(method, scenarios, config)

# Analyze accuracy
analysis <- medsim_analyze(results)

# View overall metrics
print(analysis$accuracy)

# View by-scenario metrics
print(analysis$by_scenario)

# Custom metrics
analysis <- medsim_analyze(
  results,
  metrics = c("bias", "mae", "rmse"),
  by_scenario = FALSE
)

## End(Not run)
```

`medsim_analyze_coverage` *Analyze Coverage Rates*

Description

Computes coverage rates for confidence intervals from simulation results. Requires that the simulation method returns confidence interval bounds (e.g., `ci_lower`, `ci_upper`).

Usage

```
medsim_analyze_coverage(
  results,
  ci_levels = c(0.9, 0.95, 0.99),
  ci_suffix = "_ci",
  by_scenario = TRUE,
  estimand = NULL
)
```

Arguments

`results`	A medsim_results object from `medsim_run()`
`ci_levels`	Numeric vector: nominal confidence levels to check (default: c(0.90, 0.95, 0.99))

ci_suffix	Character: suffix for CI columns (default: "_ci") E.g., if parameter is "indirect", looks for "indirect_ci_lower" and "indirect_ci_upper"
by_scenario	Logical: Compute coverage separately for each scenario (default TRUE)
estimand	Optional medsim_estimand() descriptor. When supplied with kind "interval", dispatches to interval-aware coverage computation. Default NULL uses standard CI-column-based coverage.

Details

Coverage Interpretation:

Coverage rate is the proportion of confidence intervals that contain the true parameter value. For a 95% CI, expect ~95% coverage in large samples.

- Coverage < nominal: CI too narrow (anti-conservative)
- Coverage ~= nominal: CI has correct width
- Coverage > nominal: CI too wide (conservative)

Column Naming Conventions:

The function looks for columns named:

- {parameter}_ci_lower and {parameter}_ci_upper (default)
- Or custom suffix: {parameter}{ci_suffix}_lower and {parameter}{ci_suffix}_upper

Value

A list with class "medsim_coverage" containing:

- coverage: data.frame with coverage rates, plus n_failed/failure_rate (method() failures, conditional on data_generator() having succeeded) and n_dgp_failed/dgp_failure_rate (data_generator() failures, unconditional – the two rates use different denominators and do not sum, by design; see .medsim_failure_split())
- by_scenario: data.frame with coverage by scenario (if by_scenario=TRUE)
- summary: Overall summary statistics

See Also

medsim_analyze(), medsim_analyze_power()

Examples

```
## Not run:
# Method that returns CIs
my_method <- function(data, params) {
  # ... estimation
  list(
    indirect = estimate,
    indirect_ci_lower = ci[1],
    indirect_ci_upper = ci[2]
  )
}
```

```
results <- medsim_run(my_method, scenarios, config)

# Analyze coverage
coverage <- medsim_analyze_coverage(results)

print(coverage$coverage)
#>   parameter  coverage_90  coverage_95  coverage_99
#>   indirect   0.902        0.951        0.991

## End(Not run)
```

`medsim_analyze_performance` *ADEMP performance summary (bias, SEs, RMSE) with Monte Carlo SEs*

Description

Per scenario, computes bias, empirical SE, mean model SE, and RMSE for a parameter, each with its Monte Carlo SE (Morris, White & Crowther 2019).

Usage

```
medsim_analyze_performance(results, param = "indirect")
```

Arguments

`results`	A `medsim_results` object from `medsim_run()` or `medsim_combine_chunks()`, whose `$truth` table is merged in by scenario; or a results data.frame that already has `<param>` and `<param>_truth` columns. Either way, a `<param>_se` column (optional) supplies the model SE. For a `medsim_results` object, a scenario with no truth for `param` gets `NA` bias and RMSE, and a warning names it.
`param`	Parameter name (default "indirect").

Details

Truth is taken as the first converged row's `<param>_truth` value and assumed constant within scenario. Rows with NA estimates are excluded from computations; if all estimates are NA, numeric columns return NA.

Value

A data.frame, one row per scenario.

`medsim_analyze_power` *Analyze Statistical Power*

Description

Computes empirical power (rejection rate under alternative hypothesis) from simulation results. Requires that the simulation method returns p-values.

Usage

```
medsim_analyze_power(
  results,
  alpha = 0.05,
  p_suffix = "_p",
  null_value = 0,
  by_scenario = TRUE
)
```

Arguments

`results`	A medsim_results object from `medsim_run()`
`alpha`	Numeric: significance level (default: 0.05)
`p_suffix`	Character: suffix for p-value columns (default: "_p") E.g., if parameter is "indirect", looks for "indirect_p"
`null_value`	Numeric: null hypothesis value (default: 0)
`by_scenario`	Logical: Compute power separately for each scenario (default TRUE)

Details

Power Interpretation:
Power is the probability of correctly rejecting the null hypothesis when it is false (i.e., detecting a true effect).

- Power = proportion of replications where p < alpha
- Higher power = better ability to detect effects
- Power depends on: effect size, sample size, alpha level

Notes:
This function computes **empirical power** from simulations, not theoretical power. For scenarios where the null is true (e.g., indirect effect = 0), the rejection rate represents Type I error rate, not power.

Value

A list with class "medsim_power" containing:

- `power`: data.frame with power rates
- `by_scenario`: data.frame with power by scenario (if by_scenario=TRUE)
- `summary`: Overall summary statistics

See Also

medsim_analyze(), medsim_analyze_coverage()

Examples

```
## Not run:
# Method that returns p-values
my_method <- function(data, params) {
  # ... estimation and testing
  list(
    indirect = estimate,
    indirect_p = p_value
  )
}

results <- medsim_run(my_method, scenarios, config)

# Analyze power
power <- medsim_analyze_power(results, alpha = 0.05)

print(power$power)
#>   parameter  power    n_valid
#>   indirect   0.847    1000

## End(Not run)
```

`medsim_audit_results` *Audit a combined results object for chunked-run integrity violations*

Description

Standalone entry point for Gate A of the chunked-run integrity layer (SPEC-medsim-chunked-run-gates-2026-07-31): run the same audit medsim_combine_chunks() performs, on an already-combined `medsim_results` object. Checks, per scenario:

1. **Contiguity** – `replication` ids form a gapless, duplicate-free `1..max` run, with the same `max` in every scenario (self-validating: needs no external replication count).
2. **Collapse signature** – every continuous estimate column has more than `collapse_threshold * n_ok` distinct values (the 0.3.1 seed-collapse produced ~17 distinct outcomes in 1000 while every chunk exited 0). An all-failed cell (`n_ok == 0`) is reported as `cell_failed`, not collapse.
3. **Seed-space collisions** – no two scenarios map onto overlapping .medsim_det_seed() sequences (the ~1e6-bucket name hash can collide).
4. **Partial chunks** – for a combined object, any chunk whose provenance is flagged `partial` (a task failed mid-run) is reported as a `partial_chunk` violation, exactly as medsim_combine_chunks() reports it. The surviving ids of a partial chunk can still be contiguous, so checks 1-3 alone would pass it. A bare data.frame carries no provenance and skips this check.

Frames without the schema-v2 stamp (produced before medsim 0.5.0, when `replication` was chunk-LOCAL and collided across chunks) skip checks 1 and 3 with a warning – they cannot be audited for id integrity.

Usage

```
medsim_audit_results(
  results,
  on_violation = c("stop", "warn", "ignore"),
  nsim = NULL,
  collapse_threshold = 0.9,
  collapse_digits = 12L,
  collapse_min_cell = 30L,
  collapse_exclude = .medsim_collapse_exclude_default
)
```

Arguments

`results`	A `medsim_results` object (e.g. from `medsim_combine_chunks()`), or a bare results data.frame.
`on_violation`	One of `"stop"` (default), `"warn"`, `"ignore"`: what to do when an integrity violation is found (missing chunks, duplicate or gapped replication ids, ragged cells, output collapse, all-failed cells, cross-scenario seed collisions, a chunk flagged partial by a failed task – including one whose every replication failed – or a nonzero count of `data_generator()` failures). `"stop"` signals a `medsim_combine_violation` condition that CARRIES the combined results – recover an hours-long run's good cells with `tryCatch(medsim_combine_chunks(...), medsim_combine_violation = function(e) e$results)`. For a deliberate partial combine (an interim look at a still-running array), pass `on_violation = "warn"`.
`nsim`	Integer or `NULL`: if supplied, additionally assert that every scenario has exactly `nsim` replications (the contiguity audit is otherwise self-validating and needs no external count).
`collapse_threshold`, `collapse_digits`, `collapse_min_cell`	Tuning for the collapse-signature audit: per scenario and per continuous estimate column, require `n_distinct(round(x, collapse_digits)) > collapse_threshold * n_ok` (defaults 0.9 and 12; calibration: the 0.3.1 seed-collapse produced ~17 distinct outcomes in 1000). Cells with fewer than `collapse_min_cell` non-NA values (default 30) are skipped as too small to diagnose.
`collapse_exclude`	Character: DISCRETE method-contract fields excluded from the collapse audit BY NAME (default: `converged`, `branch_switch`, plus the `medsim_method_mbco_mi()` branch diagnostics `branch_mix`, `stacked_branch`, `p_branch_a`, `r4`, `r4_fixed` – 0/1 flags, an `m`-valued share, and ARIVs with point mass at 0, all legitimately low-cardinality). Name-based (not distinctness-based) so a totally collapsed estimate column cannot exempt itself; add any custom discrete field your method emits.

Value

Invisibly, the list of violations found (empty if clean). Signalling follows `on_violation`, exactly as in `medsim_combine_chunks()`.

See Also

`medsim_combine_chunks()`, `medsim_run_chunk()`

`medsim_cache_clear` *Clear Cache*

Description

Deletes cache files from a directory. Can delete all files or files matching a pattern.

Usage

```
medsim_cache_clear(
  cache_dir,
  pattern = "*.rds",
  max_age = NULL,
  confirm = TRUE
)
```

Arguments

`cache_dir`	Character: Directory containing cache files
`pattern`	Character: Optional pattern to match files (e.g., "truth_*.rds")
`max_age`	Numeric: Delete only files older than this many days. If NULL, deletes all matching files.
`confirm`	Logical: Ask for confirmation before deleting (default TRUE)

Value

Invisibly returns number of files deleted

Examples

```
## Not run:
# Clear all cache
medsim_cache_clear("cache")

# Clear old cache only
medsim_cache_clear("cache", max_age = 30)

# Clear specific pattern
medsim_cache_clear("cache", pattern = "truth_*.rds")
```

```
# Without confirmation
medsim_cache_clear("cache", confirm = FALSE)

## End(Not run)
```

`medsim_cache_exists` *Check if Cache Exists*

Description

Checks if a cache file exists and is readable.

Usage

```
medsim_cache_exists(file)
```

Arguments

`file`	Character: Path to cache file

Value

Logical: TRUE if cache exists and is readable, FALSE otherwise

Examples

```
## Not run:
if (medsim_cache_exists("cache/truth.rds")) {
  truth <- medsim_cache_load("cache/truth.rds")
} else {
  truth <- list(indirect = 0.09)  # Compute your truth values
  medsim_cache_save(truth, "cache/truth.rds")
}

## End(Not run)
```

`medsim_cache_info` *Get Cache Info*

Description

Retrieves metadata about a cached object without loading the full object.

Usage

```
medsim_cache_info(file)
```

Arguments

`file`	Character: Path to cache file

Value

List with cache metadata:

- `exists`: Logical - file exists
- `size_mb`: Numeric - file size in MB
- `modified`: POSIXct - last modified time
- `age_days`: Numeric - age in days
- `timestamp`: POSIXct - creation time (if available)
- `r_version`: Character - R version used (if available)

Examples

```
# Save a cached object to a temp file, then inspect it
cache_file <- tempfile(fileext = ".rds")
medsim_cache_save(list(value = 42), cache_file)
info <- medsim_cache_info(cache_file)
print(info)
unlink(cache_file)
```

medsim_cache_init *Initialize Cache Directory*

Description

Creates a cache directory with proper structure and a README file explaining the cache system.

Usage

```
medsim_cache_init(cache_dir, create_readme = TRUE)
```

Arguments

cache_dir	Character: Path to cache directory
create_readme	Logical: Create README.md explaining cache (default TRUE)

Value

Invisibly returns cache directory path

Examples

```
cache_dir <- file.path(tempdir(), "medsim_cache_example")
medsim_cache_init(cache_dir)
unlink(cache_dir, recursive = TRUE)
```

medsim_cache_list *List Cache Files*

Description

Lists all cache files in a directory with their metadata.

Usage

```
medsim_cache_list(cache_dir, pattern = "*.rds")
```

Arguments

cache_dir	Character: Directory containing cache files
pattern	Character: Pattern to match files (default "*.rds")

Value

data.frame with columns:

- `file`: File name
- `path`: Full path
- `size_mb`: Size in MB
- `age_days`: Age in days
- `modified`: Last modified time

Examples

```
# Create a small cache in a temp directory, then list it
cache_dir <- file.path(tempdir(), "medsim_cache_list_example")
dir.create(cache_dir, showWarnings = FALSE)
medsim_cache_save(list(a = 1), file.path(cache_dir, "result1.rds"))
medsim_cache_save(list(b = 2), file.path(cache_dir, "result2.rds"))

cache_list <- medsim_cache_list(cache_dir)
print(cache_list)

unlink(cache_dir, recursive = TRUE)
```

`medsim_cache_load` *Load Object from Cache*

Description

Loads a previously cached R object. Returns NULL if cache file doesn't exist.

Usage

```
medsim_cache_load(file, check_expiry = FALSE, max_age = 30, verbose = TRUE)
```

Arguments

`file`	Character: Path to cache file
`check_expiry`	Logical: Check if cache has expired (default FALSE)
`max_age`	Numeric: Maximum age of cache in days (default 30). Only used if `check_expiry = TRUE`.
`verbose`	Logical: Print messages about cache status (default TRUE)

Details

Cache Validation:

When loading, the cache is checked for:

- File exists
- File is readable
- Age is within max_age (if check_expiry = TRUE)

Handling Missing Cache:

If cache doesn't exist:

- Returns NULL (no error)
- Calling code should compute and cache the result

Example pattern:

```
result <- medsim_cache_load("cache/truth.rds")
if (is.null(result)) {
  result <- expensive_computation()
  medsim_cache_save(result, "cache/truth.rds")
}
```

Note: ground-truth caches written by `medsim_run()` use a fingerprinted wrapper format `list(truth = , fingerprint = )` (so a scenario content change invalidates the cache) – unwrap `$truth` after loading, as in the example below.

Value

Cached object if file exists and is valid, NULL otherwise

See Also

`medsim_cache_save()`, `medsim_cache_exists()`

Examples

```
## Not run:
# Load a ground-truth cache written by medsim_run(). Since the
# fingerprinted format, the cached object is list(truth=, fingerprint=),
# not the bare truth value -- unwrap it (legacy bare caches load as-is):
cached <- medsim_cache_load("cache/truth_scenario1.rds")
truth <- if (is.list(cached) && !is.null(cached$fingerprint)) cached$truth else cached

if (is.null(truth)) {
  # Cache miss - compute and cache
  truth <- compute_expensive_truth()
  medsim_cache_save(truth, "cache/truth_scenario1.rds")
}

# Load with expiry check
result <- medsim_cache_load(
  "cache/old_result.rds",
```

```
  check_expiry = TRUE,
  max_age = 7  # 7 days
)

## End(Not run)
```

`medsim_cache_save` *Save Object to Cache*

Description

Saves an R object to a cache file for later retrieval. Uses RDS format for efficient storage and compression.

Usage

```
medsim_cache_save(object, file, compress = TRUE, overwrite = TRUE)
```

Arguments

`object`	Any R object to cache
`file`	Character: Path to cache file. Parent directory will be created if it doesn't exist.
`compress`	Logical: Compress the cached object (default TRUE). Can significantly reduce file size for large objects.
`overwrite`	Logical: Overwrite existing cache file (default TRUE)

Details

Cache File Format:

Cache files are stored in RDS format with metadata:

- Original object
- Timestamp of creation
- R version used
- Package version (if available)

File Organization:

Recommended cache directory structure:

```
cache/
|-- truth_scenario_1.rds
|-- truth_scenario_2.rds
+-- ...
```

Value

Invisibly returns the file path

See Also

medsim_cache_load(), medsim_cache_clear()

Examples

```
## Not run:
# Cache simulation truth
truth <- compute_expensive_truth()
medsim_cache_save(truth, "cache/truth_scenario1.rds")

# Cache with metadata
result <- list(value = 42, timestamp = Sys.time())
medsim_cache_save(result, "cache/my_result.rds")

## End(Not run)
```

---

medsim_check_results *Check Results for Errors*

Description

Checks a list of results from parallel execution for errors and provides a summary.

Usage

```
medsim_check_results(results, stop_on_error = FALSE)
```

Arguments

results	List of results from medsim_run_parallel()
stop_on_error	Logical: If TRUE, stops with error if any task failed

Value

Invisibly returns indices of failed tasks

Examples

```
## Not run:
results <- medsim_run_parallel(...)
medsim_check_results(results)

## End(Not run)
```

`medsim_combine_chunks` *Combine chunk RDS files produced by medsim_run_chunk()*

Description

Reads all `chunk_<k>.rds` files from `output_dir`, merges their `$results` and `$truth` data frames, and returns a single `medsim_results` object that is equivalent to running all replications in one call.

Usage

```
medsim_combine_chunks(
  output_dir,
  pattern = "chunk_*.rds",
  expected_chunks = NULL,
  on_violation = c("stop", "warn", "ignore"),
  nsim = NULL,
  collapse_threshold = 0.9,
  collapse_digits = 12L,
  collapse_min_cell = 30L,
  collapse_exclude = .medsim_collapse_exclude_default,
  pilot_reference = NULL,
  pilot_tol = 1e-09,
  verbose = TRUE
)
```

Arguments

`output_dir`	Character: directory containing `chunk_*.rds` files.
`pattern`	Character: glob pattern for chunk files. Default `"chunk_*.rds"`.
`expected_chunks`	Integer or `NULL`: if supplied (e.g. the SLURM array size), a shortfall in chunk files is reported as an integrity violation (see `on_violation`) – catching a failed/timed-out task instead of silently combining a partial grid. Default `NULL` (no check).
`on_violation`	One of `"stop"` (default), `"warn"`, `"ignore"`: what to do when an integrity violation is found (missing chunks, duplicate or gapped replication ids, ragged cells, output collapse, all-failed cells, cross-scenario seed collisions, a chunk flagged partial by a failed task – including one whose every replication failed – or a nonzero count of `data_generator()` failures). `"stop"` signals a `medsim_combine_violation` condition that CARRIES the combined results – recover an hours-long run's good cells with `tryCatch(medsim_combine_chunks(...), medsim_combine_violation = function(e) e$results)`. For a deliberate partial combine (an interim look at a still-running array), pass `on_violation = "warn"`.
`nsim`	Integer or `NULL`: if supplied, additionally assert that every scenario has exactly `nsim` replications (the contiguity audit is otherwise self-validating and needs no external count).

`collapse_threshold`, `collapse_digits`, `collapse_min_cell`
: Tuning for the collapse-signature audit: per scenario and per continuous estimate column, require `n_distinct(round(x, collapse_digits)) > collapse_threshold * n_ok` (defaults 0.9 and 12; calibration: the 0.3.1 seed-collapse produced ~17 distinct outcomes in 1000). Cells with fewer than `collapse_min_cell` non-NA values (default 30) are skipped as too small to diagnose.

`collapse_exclude`
: Character: DISCRETE method-contract fields excluded from the collapse audit BY NAME (default: `converged`, `branch_switch`, plus the `medsim_method_mbco_mi()` branch diagnostics `branch_mix`, `stacked_branch`, `p_branch_a`, `r4`, `r4_fixed` – 0/1 flags, an `m`-valued share, and ARIVs with point mass at 0, all legitimately low-cardinality). Name-based (not distinctness-based) so a totally collapsed estimate column cannot exempt itself; add any custom discrete field your method emits.

`pilot_reference`
: Character path (or `medsim_results` object): an archived pilot run to use as a positive control (Gate D). Because seeds depend only on `(scenario, replication)`, the full run's replications `1..B_pilot` are draw-identical to the pilot – a free regression check that harness, environment, and seeding are unchanged since the pilot passed. Identity is asserted FIRST (same sample size `n`, same scenario generator/params via fingerprint) so a stale or mis-configured pilot fails loud as `pilot_config_differs` instead of masquerading as a seeding regression; then estimate columns (only – never `elapsed` or other metadata) are compared within `pilot_tol`. Default `NULL` (no pilot check).

`pilot_tol`
: Numeric: absolute per-value tolerance for the pilot comparison (default `1e-9`). Byte-equality is deliberately NOT required: a different BLAS/R build can legitimately differ in low-order bits on correct code (FORK-reproducibility scope). For a rebuilt environment, loosen the tolerance or use `on_violation = "warn"`.

`verbose`
: Logical: print file counts.

Value

A `medsim_results` object with combined `$results` and `$truth`.

See Also

`medsim_run_chunk()`, `medsim_audit_results()`, `medsim_analyze_coverage()`

Examples

```
# After all SLURM chunks complete:
# combined <- medsim_combine_chunks("simulation_results/")
# coverage <- medsim_analyze_coverage(combined)
```

`medsim_compare_configs` *Compare Multiple Configurations*

Description

Creates a comparison table showing differences between test, local, and cluster configurations.

Usage

```
medsim_compare_configs()
```

Value

A data.frame with configuration comparison

Examples

```
comparison <- medsim_compare_configs()
print(comparison)
```

`medsim_compare_methods` *Compare Multiple Methods*

Description

Compares performance of multiple estimation methods across scenarios. Combines results from multiple simulation runs and computes relative performance metrics.

Usage

```
medsim_compare_methods(..., metrics = "all")
```

Arguments

`...`	Named medsim_results objects (e.g., method1 = results1, method2 = results2)
`metrics`	Character vector: which comparison metrics to compute. Options: "accuracy", "timing", "coverage", "all" (default)

Details

Comparison Metrics:

- **Accuracy**: MAE, RMSE, bias across methods
- **Timing**: Mean/median execution time
- **Coverage**: CI coverage rates (if CIs available)
- **Relative Efficiency**: Ratio of MSE (lower = better)

Notes:

All results must:

- Use the same scenarios (matched by name)
- Use the same configuration (n_replications, seed)
- Have ground truth available (for accuracy comparison)

Value

A list with class "medsim_comparison" containing:

- `accuracy_comparison`: data.frame comparing accuracy metrics
- `timing_comparison`: data.frame comparing timing
- `coverage_comparison`: data.frame comparing coverage (if applicable)
- `summary`: Overall comparison summary

See Also

`medsim_analyze()`, `medsim_analyze_coverage()`

Examples

```
## Not run:
# Run multiple methods
results_proposed <- medsim_run(method_proposed, scenarios, config)
results_delta <- medsim_run(method_delta, scenarios, config)
results_boot <- medsim_run(method_boot, scenarios, config)

# Compare
comparison <- medsim_compare_methods(
  proposed = results_proposed,
  delta = results_delta,
  bootstrap = results_boot
)

# View accuracy comparison
print(comparison$accuracy_comparison)

# View timing comparison
print(comparison$timing_comparison)

## End(Not run)
```

`medsim_config` *Create Simulation Configuration*

Description

Creates a configuration object for simulation studies. Automatically detects whether running on local machine or HPC cluster and adjusts parameters accordingly.

Usage

```
medsim_config(
  mode = "auto",
  n_replications = NULL,
  n_cores = NULL,
  scenarios = NULL,
  output_dir = NULL,
  seed = 12345,
  chunk_id = NULL,
  n_chunks = NULL,
  array_size = NULL,
  seed_stream = NULL,
  partition = NULL,
  walltime = NULL,
  mem_per_cpu = NULL,
  r_module = NULL,
  ...,
  n = NULL,
  n_truth = NULL
)
```

Arguments

`mode`	Character: "auto", "test", "local", or "cluster" • "auto": Detect based on environment variables (SLURM, PBS, LSF) • "test": Quick validation (~30 seconds) • "local": Development on local machine (~15 minutes) • "cluster": Production on HPC cluster (hours)
`n_replications`	Integer: Number of Monte Carlo replications. If NULL, uses mode defaults (test=20, local=100, cluster=1000)
`n_cores`	Integer: Number of CPU cores for parallel processing. If NULL, auto-detects (all cores - 2 on local, SLURM_CPUS_PER_TASK on cluster)
`scenarios`	Character: "all" or "test". Use "test" for single challenging scenario during development
`output_dir`	Character: Directory for saving results

seed	Integer: Base seed, stored on the config and printed for provenance. Consulted directly by medsim_run() to seed the compute_truth pass (set.seed(config$seed) before computing ground truth), so every chunk of an array computes the same truth. **Does not** determine individual replication draws – since the .medsim_det_seed() fix, those are seeded deterministically from (scenario_name, global_rep_id), independent of seed/chunk/worker/ cluster-type. See seed_stream below for its deprecation.
chunk_id	Integer: SLURM array task index (1-based) for this chunk. Auto-detected from SLURM_ARRAY_TASK_ID when running inside a SLURM array job and not supplied explicitly. NULL = no chunking.
n_chunks	Integer: Total number of chunks (SLURM array size). Used by medsim_run_chunk() to slice the replication index.
array_size	Integer: Alias for n_chunks (matches SLURM terminology). When both are supplied, n_chunks wins.
seed_stream	Integer: **Deprecated.** This knob is stored on the config but has never been consumed by any medsim function – passing a non-NULL value now emits a deprecation warning and has no effect on any RNG stream. For reproducible per-worker L'Ecuyer-CMRG streams, pass seed= directly to medsim_run_parallel(); medsim_run()/ medsim_run_chunk() seed each replication deterministically (see the seed param above) and need no stream seed.
partition	Character: SLURM partition (queue) name. Defaults to "general" in cluster mode; NULL otherwise.
walltime	Character: SLURM wall-time limit (HH:MM:SS). Defaults to "08:00:00" in cluster mode.
mem_per_cpu	Character: SLURM memory per CPU. Defaults to "4G" in cluster mode.
r_module	Character: Environment module string for module load. Defaults to "r/4.4.0-ytj2" (UNM CARC Hopper) in cluster mode.
...	Additional custom parameters
n	Positive whole number or NULL: the sample size of each simulated data set. medsim_run() and medsim_run_chunk() call every scenario's data_generator(n = n). NULL (the default) leaves it unset, and the runner uses 200. Because it follows ..., it must be named in full: medsim_config("local", n = 100).
n_truth	Positive whole number or NULL: the sample size of the single large data set drawn to compute ground truth (see compute_truth in medsim_run()). NULL (the default) uses 10000.

Details

Execution Modes:

Mode	Replications	Cores	Runtime	Use Case
test	20	4	~30s	Quick validation
local	100	auto	~15m	Development
cluster	1000	SLURM	hours	Production

Environment Detection:

When mode = "auto", checks for:

- SLURM_JOB_ID (SLURM scheduler)
- PBS_JOBID (PBS/Torque scheduler)
- LSB_JOBID (LSF scheduler)

If any are found, uses "cluster" mode. Otherwise, uses "local" mode.

Custom Parameters:

You can add custom parameters via ...:

```
config <- medsim_config(
  mode = "local",
  n_bootstrap = 5000,
  alpha = 0.05,
  custom_param = "value"
)
```

Value

A list with simulation configuration parameters

See Also

medsim_detect_environment(), print.medsim_config()

Examples

```
# Auto-detect environment
config <- medsim_config(mode = "auto")

# Explicit test mode for quick validation
config_test <- medsim_config(mode = "test")

# Local mode with custom replications
config_local <- medsim_config(
  mode = "local",
  n_replications = 500
)

# Cluster mode (auto-detects SLURM cores)
config_cluster <- medsim_config(mode = "cluster")

# Sample size of each simulated data set (default 200)
config_n100 <- medsim_config(mode = "local", n = 100)

# Custom parameters
config_custom <- medsim_config(
  mode = "local",
  n_bootstrap = 1000,
  ci_level = 0.95
)
```

```
# Print configuration
print(config)
```

`medsim_detect_cores` *Detect Number of Available Cores*

Description

Detects the number of CPU cores available for parallel processing. On HPC clusters, uses scheduler-allocated cores. On local machines, uses all cores minus 2 to avoid overwhelming the system.

Usage

```
medsim_detect_cores()
```

Details

On clusters, checks for:

- SLURM_CPUS_PER_TASK (SLURM)
- PBS_NUM_PPN (PBS/Torque)
- LSB_DJOB_NUMPROC (LSF)

On local machines, uses `parallel::detectCores() - 2`.

Value

Integer: Number of cores to use

Examples

```
n_cores <- medsim_detect_cores()
message(sprintf("Using %d cores", n_cores))
```

`medsim_detect_environment` *Detect Computing Environment*

Description

Detects whether code is running on local machine or HPC cluster by checking for scheduler environment variables.

Usage

```
medsim_detect_environment()
```

Details

Checks for the following environment variables:

- SLURM_JOB_ID (SLURM scheduler)
- PBS_JOBID (PBS/Torque scheduler)
- LSB_JOBID (LSF scheduler)

If any are found, returns "cluster". Otherwise, returns "local".

Value

Character: "local" or "cluster"

Examples

```
env <- medsim_detect_environment()
if (env == "cluster") {
  message("Running on HPC cluster")
} else {
  message("Running on local machine")
}
```

`medsim_estimand` *Declare the estimand kind for a simulation scenario*

Description

Attach a first-class estimand descriptor to a `medsim_scenario()`. Every downstream command (`medsim_run`, `medsim_analyze_coverage`, `medsim_tables`, `medsim_figures`, `medsim_workflow`) dispatches on `estimand$kind` rather than assuming a scalar point-estimand.

Five kinds are recognised:

kind	what `method()` returns	coverage notion
`"point"`	`{p}`, `{p}_ci_lower/_upper`, `{p}_p`	truth in CI
`"interval"`	`{p}_lower/_upper`, `{p}_im_lower/_im_upper`, `feasible`, `falsified`	truth in [lower, upper]; IM-C
`"probabilistic"`	`pmed`, `pmed_ci_lower/_upper`, `pmed_p`, `branch_switch`	truth in CI (truth from poter
`"variance_share"`	`{p}`, `{p}_ci_lower/_upper`	truth in Wald CI (bounded s
`"numeric"`	`est_error`, `abs_error`, `elapsed_sec`	none

`"variance_share"` is a bounded scalar point estimand with a standard Wald CI — the Sobol / functional-ANOVA proportion mediated $P^{\sigma^2}_{med} = V_{med}/V_T \in [0,1]$ (see `medsim_scenario_sobol()`, `medsim_method_sobol()`). Coverage uses the generic `truth in CI` path; the dedicated kind exists for clear labeling and so `medsim_validate_scenario()` checks the causal-notation `A/M/Y/C` columns the Sobol estimator requires (rather than the legacy `X/M/Y` of `"point"`).

Scenarios with `estimand = NULL` (the default in `medsim_scenario()`) are treated as `kind = "point"` throughout the package – full back-compatibility with all existing code.

Usage

```
medsim_estimand(
  kind = c("point", "interval", "probabilistic", "variance_share", "numeric"),
  params = character(),
  truth = NULL,
  ci = c("standard", "imbens_manski", "mbco", "none"),
  extra = character()
)
```

Arguments

`kind`	Estimand kind: `"point"` (default), `"interval"`, `"probabilistic"`, `"variance_share"`, or `"numeric"`.
`params`	Character vector of estimand parameter names – e.g. `c("indirect")` for point, `c("NDE", "NIE")` for interval, `c("pmed")` for probabilistic. Defaults to `character()` (auto-inferred downstream).
`truth`	Optional truth-extractor `function(scenario)` returning a **named** numeric vector keyed by the names in `params`. Required for `kind = "interval"` (bounds truth differs from the estimand itself).

ci	CI method: "standard" (Wald/bootstrap), "imbens_manski" (partial-ID dual CI), "mbco" (constrained optimisation), or "none".
extra	Character vector of additional result columns beyond params that method() returns – e.g. c("feasible", "branch_switch", "timing").

Value

An object of class c("medsim_estimand", "list").

See Also

medsim_scenario() for attaching an estimand to a scenario; medsim_analyze_coverage() for the keystone dispatch on kind.

Examples

```
# Point estimand (the historic default -- back-compat)
medsim_estimand("point")

# Interval (partial-ID bounds)
medsim_estimand("interval",
  params = c("NDE", "NIE"),
  ci     = "imbens_manski",
  extra  = c("feasible", "falsified"))

# Probabilistic (P_med)
medsim_estimand("probabilistic",
  params = "pmed",
  ci     = "mbco",
  extra  = "branch_switch")

# Numerical accuracy (no coverage/power)
medsim_estimand("numeric", params = "est_error", ci = "none",
  extra = "elapsed_sec")
```

---

medsim_estimate_speedup

Estimate Parallel Speedup

Description

Estimates the expected speedup from parallel processing based on number of cores and task characteristics.

Usage

```
medsim_estimate_speedup(n_tasks, n_cores, overhead = 0.1)
```

Arguments

n_tasks	Integer: Number of tasks
n_cores	Integer: Number of cores
overhead	Numeric: Overhead per core (default 0.1 = 10%)

Details

Speedup is estimated using Amdahl's law with overhead:

Speedup = n_cores / (1 + overhead * n_cores)

Efficiency = Speedup / n_cores

Value

Named list with:

- speedup: Expected speedup factor
- efficiency: Parallel efficiency (0-1)
- recommendation: Character recommendation

Examples

```
# For 100 tasks on 10 cores
est <- medsim_estimate_speedup(100, 10)
cat(sprintf("Expected speedup: %.1fx\n", est$speedup))
cat(sprintf("Efficiency: %.0f%%\n", est$efficiency * 100))
```

---

medsim_figures *Generate all standard figures from simulation results*

Description

Convenience wrapper that produces the standard set of medsim plots (coverage, error boxplot, timing) and writes them to output_dir.

Usage

```
medsim_figures(
  results,
  output_dir = "figures",
  format = "pdf",
  width = 8,
  height = 6,
  ...
)
```

Arguments

`results`	A `medsim_results` object from `medsim_run()`.
`output_dir`	Directory to write figures to. Created if it doesn't exist.
`format`	File extension passed to `ggplot2::ggsave()` (e.g. `"pdf"`, `"png"`).
`width, height`	Figure dimensions in inches.
`...`	Additional arguments passed to `ggplot2::ggsave()`.

Value

Invisibly, a named character vector of file paths written.

`medsim_get_optimal_cores` *Get Optimal Number of Cores*

Description

Determines the optimal number of cores to use for parallel processing based on system capabilities and task requirements.

Usage

```
medsim_get_optimal_cores(n_tasks, reserve = 2, max_cores = NULL)
```

Arguments

`n_tasks`	Integer: Number of tasks to process. If fewer tasks than cores, returns `n_tasks`.
`reserve`	Integer: Number of cores to reserve for system (default 2). Only applied on local machines, not clusters.
`max_cores`	Integer: Maximum number of cores to use. If NULL, uses all available (minus reserve).

Value

Integer: Optimal number of cores to use

Examples

```
# For 100 tasks
medsim_get_optimal_cores(n_tasks = 100)

# For few tasks (returns n_tasks)
medsim_get_optimal_cores(n_tasks = 3)

# With maximum limit
medsim_get_optimal_cores(n_tasks = 1000, max_cores = 8)
```

`medsim_method_bounds` *Bounds estimator adapter for differential-misclassification (DM) studies*

Description

Returns partial-identification bounds (NDE/NIE) following the `interval`-kind result contract expected by `medsim_run()`, so that `medsim_analyze_coverage()` dispatches the Imbens-Manski / partial-ID interval coverage branch.

Current implementation: OLS decomposition + inflated interval (synthetic bounds for unit-testing the harness). A future version will call `medrobust::bound_ne()` once its accessor API stabilises on CRAN.

Result columns returned (flat named list):

- `NDE_lower`, `NDE_upper` – bounds on natural direct effect
- `NDE_im_lower`, `NDE_im_upper` – Imbens-Manski CI for NDE
- `NIE_lower`, `NIE_upper` – bounds on natural indirect effect
- `NIE_im_lower`, `NIE_im_upper` – Imbens-Manski CI for NIE
- `feasible` – logical: bounds are non-empty / consistent
- `falsified` – logical: data rejects identifying assumptions

Usage

```
medsim_method_bounds(
  data,
  params,
  misclass_type = c("mediator", "exposure"),
  alpha = 0.05
)
```

Arguments

`data`	A data.frame produced by the scenario's `data_generator`. Expected columns: `A` (binary treatment), `M` or `M_star` (mediator, possibly misclassified), `Y` (outcome).
`params`	Named list of scenario parameters. Must include `NDE` and `NIE` (ground-truth natural effects) so truth can be tracked.
`misclass_type`	`"mediator"` (default) or `"exposure"`.
`alpha`	Nominal level; `1 - alpha` CI (default `0.05`).

Value

Named list with elements `NDE_lower`, `NDE_upper`, `NDE_im_lower`, `NDE_im_upper`, `NIE_lower`, `NIE_upper`, `NIE_im_lower`, `NIE_im_upper`, `feasible`, `falsified`.

See Also

medsim_scenario_dm() for the companion scenario factory; medsim_analyze_coverage() for interval-kind coverage analysis.

Examples

```
set.seed(42)
d <- data.frame(A = rbinom(200, 1, 0.5), M = rnorm(200), Y = rnorm(200))
result <- medsim_method_bounds(d, list(NDE = 0.2, NIE = 0.3))
str(result)
```

---

medsim_method_gauge *Gauge-residual (P_med + W) estimator wrapper*

Description

Adapter that calls an external ward_residual() estimator and returns the flat field contract for two parameters:

Field	Description
pmed, pmed_ci_lower/_upper	interventional proportion mediated IIE/OE
w, w_ci_lower/_upper	gauge residual R/OE

The estimator is **not vendored** (medsim keeps estimators external). Pass it via estimator, or default to an object named ward_residual on the search path / probmed::ward_residual. The estimator must return an S7 GaugePmedResult (read via @p_med/@p_med_ci/@W/@W_ci) or a plain list with fields p_med, p_med_ci (length 2), W, W_ci (length 2).

Usage

```
medsim_method_gauge(
  data,
  params = list(),
  estimator = get0("ward_residual"),
  covars = "C",
  K = 5L,
  ci_level = 0.95,
  seed = 1L,
  reps = 1L,
  se_method = "analytic",
  B = 200L,
  fieller = FALSE
)
```

Arguments

data	data.frame with columns A, M, Y, C.
params	Named list from medsim_scenario_gauge() (accepted for the method(data, params) signature; unused by the estimator).
estimator	Function; default get0("ward_residual").
covars	Character covariate names (default "C").
K	Integer cross-fitting folds (default 5).
ci_level	Numeric CI level (default 0.95).
seed	Integer seed (default 1).
reps	Integer repeated cross-fitting fold draws (default 1).
se_method	"analytic" (default) or "bootstrap" (percentile).
B	Integer bootstrap resamples when se_method = "bootstrap" (default 200).
fieller	Logical: compute the estimator's Fieller bounds (default FALSE). medsim reports only the Wald/percentile pmed/w CIs, so the Fieller set is discarded — disabling it avoids wasted compute per replication across a large grid.

Value

Named list pmed, pmed_ci_lower, pmed_ci_upper, w, w_ci_lower, w_ci_upper.

See Also

medsim_scenario_gauge(), medsim_analyze_coverage()

---

medsim_method_ipw *Thin IPW estimator adapter (robustness appendix)*

Description

Inverse-probability-weighting comparator. Positioned as a robustness appendix, not a co-equal main-results arm (decided 2026-06-11).

Usage

```
medsim_method_ipw(model, ...)
```

Arguments

model	Mediation model spec.
...	Passed to the underlying weighting / fitting routines.

Value

A function(data, params) returning the shared method contract (branch_switch = NA).

Examples

```
set.seed(1)
d <- data.frame(X = rnorm(200))
d$M <- 0.39 * d$X + rnorm(200)
d$Y <- 0.39 * d$M + 0.14 * d$X + rnorm(200)
d <- medsim_amputate(d, target = "M", mechanism = "MAR", prop = 0.25)
out <- medsim_method_ipw(model = NULL)(d, list())
out[c("indirect", "indirect_ci_lower", "indirect_ci_upper")]
```

---

medsim_method_mbco_mi *MBCO-MI estimator adapter*

Description

Multiple imputation (via `mice`) + **D4-stacked MBCO** likelihood-ratio inference for the union null `H0: ab = 0`. Ports the validated `Missing Effect/code/prototype-d4-mbco.R` (phase-2; the D4 statistic matches `mitml::testModels(method = "D4")`, and so does the p-value with `df = "chan_meng"`): per-imputation MBCO LRTs are pooled with the Reiter/Chan-Meng D4 rule (Grund et al. 2021) into an F reference.

Usage

```
medsim_method_mbco_mi(
  model,
  m = 20L,
  ariv = c("fixed", "own"),
  df = c("chan_meng", "li"),
  ...
)
```

Arguments

`model`	Mediation model spec (accepted for API symmetry; the estimator reads the `X`, `M`, `Y` (+ optional `C*`) columns of `data`).
`m`	Integer number of imputations.
`ariv`	Which relative-increase-in-variance (ARIV) estimate feeds the `indirect_p` column: `"fixed"` (default, as in missingmed) recomputes every per-imputation statistic on the branch the stacked constrained fit selected, so branch disagreement across imputations cannot pull the ARIV down; `"own"` (the previous default) pools per-imputation MBCO statistics each computed on its own winning branch (standard Chan-Meng `r4`) and can be liberal at high missingness. Both p-values are always emitted (`indirect_p` and `indirect_p_fixed`), together with the branch-mixing diagnostics `branch_mix`, `p_branch_a`, `stacked_branch`, `r4`, `r4_fixed` (with a single imputation there is no stacked fit: `stacked_branch` is that fit's branch and `r4 = r4_fixed = 0`; on the error path all six are `NA`).

`df` Denominator degrees of freedom for the D4 `F` reference, used for both `indirect_p` and `indirect_p_fixed`: `"chan_meng"` (default, as in missingmed) is Chan and Meng (2022, eq. 2.15), $\nu = k(K-1)(1+1/r_4)^2$, the reference used by `mitml::testModels(method = "D4")`; `"li"` (the previous default) is the Li et al. (1991) / Meng-Rubin approximation, $\nu = 4 + (k(K-1)-4)\{1+(1-2/(k(K-1)))/r_4\}^2$ when $k(K-1) > 4$. Both give $\nu = \infty$ (a chi-square reference) when $r_4 = 0$. Chan and Meng derive eq. 2.15 for the own-branch r_4 at a regular null; for `ariv = "fixed"`, and at the nonregular null $a = b = 0$, either choice is a working reference whose calibration rests on simulation. The choice changes only the p-values: the statistic, r_4 and the random-number stream are identical.

`...` Reserved for future MI / MBCO options.

Details

The point estimate is the Rubin-pooled `a*b`; the CI is a Monte-Carlo product interval on the pooled paths; the p-value is the D4-pooled MBCO test. With no missingness (or without `mice`) it degrades to the complete-case MBCO LRT with a chi-square reference – never to a Sobel normal approximation for the test.

Value

A `function(data, params)` returning the shared method contract (see file header).

Examples

```
if (requireNamespace("mice", quietly = TRUE)) {
  set.seed(1)
  d <- data.frame(X = rnorm(200))
  d$M <- 0.39 * d$X + rnorm(200)
  d$Y <- 0.39 * d$M + 0.14 * d$X + rnorm(200)
  d <- medsim_amputate(d, target = "M", mechanism = "MAR", prop = 0.25)
  fit <- medsim_method_mbco_mi(model = NULL, m = 10, ariv = "fixed", df = "chan_meng")
  out <- fit(d, list())
  out[c("indirect", "indirect_p", "indirect_p_fixed", "r4_fixed")]
}
```

`medsim_method_mc_ci` *Monte-Carlo CI estimator adapter*

Description

Multiple imputation (via `mice`) + Monte-Carlo confidence interval for the indirect effect (via `RMediation::medci()` when available, else a base-R product-of-normals draw on the Rubin-pooled paths).

Usage

```
medsim_method_mc_ci(model, m = 20L, ...)
```

Arguments

model	Mediation model spec (accepted for API symmetry; the estimator reads the X, M, Y (+ optional C*) columns of data).
m	Integer number of imputations.
...	Reserved for future MI / MBCO options.

Value

A function(data, params) returning the shared method contract (branch_switch = NA).

Examples

```
if (requireNamespace("mice", quietly = TRUE)) {
  set.seed(1)
  d <- data.frame(X = rnorm(200))
  d$M <- 0.39 * d$X + rnorm(200)
  d$Y <- 0.39 * d$M + 0.14 * d$X + rnorm(200)
  d <- medsim_amputate(d, target = "M", mechanism = "MAR", prop = 0.25)
  out <- medsim_method_mc_ci(model = NULL, m = 10)(d, list())
  out[c("indirect", "indirect_ci_lower", "indirect_ci_upper")]
}
```

---

medsim_method_pmed_mbco

MBCO confidence interval for P_med (two-branch method)

Description

Estimates P_med from observed data using OLS point estimation and a **two-branch MBCO** (maximally balanced constrained optimisation) confidence interval. Returns the 6-field contract expected by medsim_run():

Field	Description
pmed	Point estimate of P_med
pmed_ci_lower	Lower bound of MBCO CI
pmed_ci_upper	Upper bound of MBCO CI
pmed_p	Two-sided p-value (H0: P_med = 0.5)
branch_switch	1 if the MBCO union-null LRT switched branches
converged	1 if optimisation converged

Algorithm:

1. Fit linear SEM: M ~ A and Y ~ A + M.
2. Estimate path coefficients alpha (A->M) and beta (M->Y).

3. Estimate P_med via a parametric bootstrap PO draw under the estimated SEM.
4. Build a delta-method normal CI and clamp to [0, 1].
5. The "branch_switch" flag records whether the union-null test selected the constrained branch (alpha*beta* <= *0) over the unconstrained branch (alpha*beta > 0).

Usage

```
medsim_method_pmed_mbco(data, params, alpha = 0.05, n_boot = 2000L)
```

Arguments

`data`	A `data.frame` with columns `A`, `M`, `Y`.
`params`	Named list from `medsim_scenario_pmed()`; must contain at minimum `alpha_ax`, `beta_my`, `beta_ay`, `sigma_m`, `sigma_y`.
`alpha`	Significance level for CI (default 0.05).
`n_boot`	Integer: parametric bootstrap size for P_med point estimate (default 2000).

Value

A named list with fields `pmed`, `pmed_ci_lower`, `pmed_ci_upper`, `pmed_p`, `branch_switch`, `converged`.

See Also

`medsim_scenario_pmed()`, `medsim_analyze_coverage()`

`medsim_method_sobol` *Sobol / functional-ANOVA variance-share estimator wrapper*

Description

Adapter that calls an external `sobol_pmed()` estimator and returns the flat field contract keyed by the `pmed_sobol` token:

Field	Description
`pmed_sobol`	Point estimate of the Sobol share V_{med}/V_T
`pmed_sobol_ci_lower`	Lower Wald CI bound
`pmed_sobol_ci_upper`	Upper Wald CI bound

Following medsim convention, the actual estimator is **not vendored** into the package (medsim keeps estimators external — it `Suggests` companion packages, never hard-depends on them). The prototype lives at `pmed-modern-sobol/03-sobol-pmed/sims/sobol_pmed.R`; `source()` it (or load the future `probmed` export) before running, or pass it via `estimator`.

The external `sobol_pmed(d, ...)` is expected to return a list with `P_med_sobol` (scalar) and `ci` (length-2 numeric `c(lower, upper)`).

Usage

```
medsim_method_sobol(
  data,
  params = list(),
  estimator = get0("sobol_pmed"),
  covars = "C",
  K = 5L,
  level = 0.95,
  seed = 1L
)
```

Arguments

data	A data.frame with columns A, M, Y, C.
params	Named list from medsim_scenario_sobol() (unused by the estimator itself, but accepted for the standard method(data, params) signature; may carry covars, K, level).
estimator	Function implementing the Sobol estimator. Defaults to get0("sobol_pmed"), i.e. an object named sobol_pmed visible on the search path (e.g. after source()-ing the prototype).
covars	Character vector of covariate column names passed to the estimator (default "C").
K	Integer cross-fitting folds passed to the estimator (default 5).
level	Numeric CI level passed to the estimator (default 0.95).
seed	Integer seed passed to the estimator (default 1).

Value

A named list with fields pmed_sobol, pmed_sobol_ci_lower, pmed_sobol_ci_upper.

See Also

medsim_scenario_sobol(), medsim_analyze_coverage()

---

medsim_nsim_for_mcse *Replications needed for a target coverage Monte Carlo SE*

Description

Inverts MCSE = sqrt(p(1-p)/nsim) to give the nsim such that a coverage cell near p has Monte Carlo SE no larger than target_mcse.

Usage

```
medsim_nsim_for_mcse(target_mcse, p = 0.95)
```

Arguments

target_mcse	Positive numeric target Monte Carlo SE.
p	Assumed coverage proportion (default 0.95, the nominal level).

Value

Integer number of replications (ceiling).

Examples

```
medsim_nsim_for_mcse(0.005, 0.95)  # 1900
```

---

medsim_plot_combined_panel

Create Combined Multi-Panel Figure

Description

Creates a combined figure with multiple panels for manuscript submission. Useful for showing multiple aspects of simulation results in one figure.

Usage

```
medsim_plot_combined_panel(
  results,
  panels = c("error", "timing"),
  layout = NULL,
  ...
)
```

Arguments

results	A medsim_results object or named list of results objects
panels	Character vector: which panels to include. Options: "error", "timing", "coverage" (default: c("error", "timing"))
layout	Character: layout specification like "2x1", "1x2", "2x2" (default: auto-determined from number of panels)
...	Additional arguments passed to individual plotting functions

Details

Requires either patchwork or gridExtra package for combining plots.

Value

A combined plot object (patchwork or gridExtra)

Examples

```
## Not run:
# Two-panel figure
p <- medsim_plot_combined_panel(
  results,
  panels = c("error", "timing"),
  layout = "1x2"
)

# Four-panel figure for manuscript
p <- medsim_plot_combined_panel(
  list(Proposed = res1, Delta = res2),
  panels = c("error", "timing", "coverage"),
  layout = "2x2"
)

## End(Not run)
```

`medsim_plot_coverage` *Plot Coverage Rates*

Description

Creates a plot showing confidence interval coverage rates across scenarios and methods.

Usage

```
medsim_plot_coverage(
  coverage,
  expected_coverage = 0.95,
  tolerance = 0.02,
  title = NULL,
  ...
)
```

Arguments

`coverage`	A medsim_coverage object from `medsim_analyze_coverage()`, or a named list of coverage objects for method comparison
`expected_coverage`	Numeric: expected coverage rate to show as reference line (default: 0.95)
`tolerance`	Numeric: tolerance band around expected coverage (default: 0.02)
`title`	Character: plot title (default: auto-generated)
`...`	Additional arguments passed to ggplot2::theme()

Value

A ggplot2 object

Examples

```
## Not run:
coverage <- medsim_analyze_coverage(results)
p <- medsim_plot_coverage(coverage, expected_coverage = 0.95)

## End(Not run)
```

`medsim_plot_error_boxplot`
Plot Error Distribution Boxplots

Description

Creates boxplots showing the distribution of estimation errors across scenarios. Useful for comparing accuracy of different methods.

Usage

```
medsim_plot_error_boxplot(
  results,
  parameter = "indirect",
  log_scale = FALSE,
  color_palette = NULL,
  title = NULL,
  ...
)
```

Arguments

`results`	A medsim_results object from `medsim_run()`, or a list of named results objects for method comparison
`parameter`	Character: which parameter to plot (default: "indirect"). If NULL, plots all parameters with available ground truth.
`log_scale`	Logical: use log10 scale for y-axis (default: FALSE)
`color_palette`	Character: color palette name from RColorBrewer or "viridis" (default: NULL uses ggplot2 defaults)
`title`	Character: plot title (default: auto-generated)
`...`	Additional arguments passed to ggplot2::theme()

Details

Interpreting Boxplots:

- Box: IQR (25th to 75th percentile)
- Line in box: Median error
- Whiskers: 1.5 x IQR
- Points: Outliers beyond whiskers

Tighter boxes = more consistent estimates

Usage Patterns:

```
# Single method
p <- medsim_plot_error_boxplot(results)

# Compare methods
p <- medsim_plot_error_boxplot(list(
  Proposed = results1,
  Delta = results2,
  Bootstrap = results3
))

# Customize
p <- medsim_plot_error_boxplot(
  results,
  log_scale = TRUE,
  color_palette = "Set2"
)
p + ggplot2::theme_minimal()
```

Value

A ggplot2 object

Examples

```
## Not run:
results <- medsim_run(method, scenarios, config)
p <- medsim_plot_error_boxplot(results)
print(p)

# Save to file
ggplot2::ggsave("error_boxplot.pdf", p, width = 10, height = 6)

## End(Not run)
```

medsim_plot_se_vs_estimate *Diagnostic scatter of model SE against the point estimate*

Description

Plots each replication's model standard error against its point estimate, faceted by scenario, to expose numerator-denominator coupling and outliers (White et al. 2023, *How to check a simulation study*, IJE).

Usage

```
medsim_plot_se_vs_estimate(results, param = "indirect", by = "scenario")
```

Arguments

results	A results data.frame from medsim_run().
param	Parameter name; expects <param> and <param>_se columns. If <param>_se is absent, derived from <param>_ci_lower/_upper.
by	Grouping/faceting column (default "scenario").

Value

A ggplot object.

medsim_plot_timing *Plot Timing Comparison*

Description

Creates bar plots comparing computational time across methods and scenarios.

Usage

```
medsim_plot_timing(
  results,
  metric = "mean",
  log_scale = FALSE,
  title = NULL,
  ...
)
```

Arguments

results	A named list of medsim_results objects (required for comparison)
metric	Character: which timing metric to plot. Options: "mean", "median", "total" (default: "mean")
log_scale	Logical: use log10 scale for y-axis (default: FALSE)
title	Character: plot title (default: auto-generated)
...	Additional arguments passed to ggplot2::theme()

Value

A ggplot2 object

Examples

```
## Not run:
p <- medsim_plot_timing(list(
  Proposed = results1,
  Delta = results2,
  Bootstrap = results3
))

## End(Not run)
```

---

medsim_rnonnormal *Draw nonnormal values with a target marginal skew/kurtosis*

Description

Fleishman / Vale?Maurelli power-method draws. Pure base R; no hard dependency. Warns/errors on (skew, kurtosis) outside the Fleishman-feasible region.

Usage

```
medsim_rnonnormal(n, mean = 0, sd = 1, skew = 0, kurtosis = 0)
```

Arguments

n	Integer sample size.
mean, sd	Target marginal mean and SD (applied after standardizing).
skew	Target marginal skewness (third standardized moment).
kurtosis	Target marginal excess kurtosis (fourth standardized moment minus 3).

Value

Numeric vector of length n with the requested marginal moments.

Examples

```
x <- medsim_rnonnormal(500, skew = 1.5, kurtosis = 4)
c(mean = mean(x), sd = stats::sd(x))
```

`medsim_run` *Run Simulation Study*

Description

Executes a complete simulation study by running a user-defined method across multiple scenarios and replications. Supports parallel processing and automatic result aggregation.

Usage

```
medsim_run(
  method,
  scenarios,
  config,
  compute_truth = NULL,
  parallel = TRUE,
  verbose = TRUE,
  truth_on_failure = c("stop", "warn")
)
```

Arguments

`method`	Function: User-defined simulation method. Must accept two arguments: `data` (data.frame) and `params` (list). Should return a named list of results.
`scenarios`	List of scenario objects from `medsim_scenario()` or `medsim_scenarios_mediation()`
`config`	Configuration object from `medsim_config()`
`compute_truth`	Function: Optional function to compute ground truth. Takes same arguments as `method`. If NULL, no ground truth computed.
`parallel`	Logical: Use parallel processing (default TRUE)
`verbose`	Logical: Print progress messages (default TRUE)
`truth_on_failure`	`"stop"` (default) or `"warn"`: what to do when `compute_truth` errors, or succeeds but returns no usable parameter, for a scenario. `"stop"` aborts before any replication runs; `"warn"` continues with the scenario omitted from `$truth`. Nothing is cached either way. See *Ground Truth*.

Details

Method Function Requirements:

The `method` function must:

- Accept `data` (data.frame) as first argument
- Accept `params` (list) as second argument
- Return a named list with at least one numeric element
- NOT use the reserved field name `error` – it marks failed replications in the results schema (a non-NA `error` forces `converged = 0`); a method returning it stops with an informative error

Example:

```
my_method <- function(data, params) {
  fit_m <- lm(M ~ X, data = data)
  fit_y <- lm(Y ~ X + M, data = data)

  a <- coef(fit_m)["X"]
  b <- coef(fit_y)["M"]

  list(
    indirect = a * b,
    a_path = a,
    b_path = b,
    se_indirect = ...,  # Optional
    ci_lower = ...,     # Optional
    ci_upper = ...      # Optional
  )
}
```

Parallel Processing:

When `parallel = TRUE`:

- Uses `medsim_run_parallel()` internally
- Number of cores from `config$n_cores`
- Progress bars shown (local) or suppressed (cluster)
- All necessary objects exported to workers

Ground Truth:

If `compute_truth` is provided:

- Computed once per scenario (cached automatically)
- Used to calculate errors/bias in results
- Can use Monte Carlo or analytical methods

`compute_truth` must return a **named list of length-1 numerics**, e.g. `list(indirect = 0.09)`. A bare named vector (`c(indirect = 0.09)`) or a list element of length > 1 contributes no parameter and is treated as a failure.

A failed or unusable truth is never cached, so the next run retries it, and the scenario is left out of `$truth`. What happens to the run is set by `truth_on_failure`: `"stop"` (the default) raises a `medsim_truth_failure_error` (subclass of `medsim_error`) carrying `$scenario` and `$parent`; `"warn"` warns and continues. Truth is computed in Step 1, before any replication runs, so stopping wastes no replication compute.

Task-level failures:

An error that escapes a replication task – a contract violation (e.g. the reserved-`error`-field stop) or an infrastructure error, as opposed to a `method()` or `data_generator()` failure (both of which become failure rows and let the run continue) – stops `medsim_run()` with a `medsim_run_failure` condition (subclass of `medsim_error`) rather than silently dropping the failed replications. The condition carries whatever completed before the failure: `$results` is a partial `medsim_results` object (prior scenarios complete, the failing scenario's successful reps included, later scenarios never started), and `$errors` is a list of `list(scenario, global_rep_id, message)`. Recover the partial run instead of losing it:

```
partial <- tryCatch(
  medsim_run(method, scenarios, config),
  medsim_run_failure = function(e) e$results
)
```

Value

A medsim_results object (list) containing:

- `results`: data.frame with all simulation results
- `summary`: data.frame with summary statistics
- `truth`: data.frame with ground truth values (if compute_truth provided)
- `config`: configuration used
- `scenarios`: scenarios used
- `method_name`: name of method function
- `timestamp`: when simulation was run

See Also

`medsim_config()`, `medsim_scenario()`, `medsim_run_parallel()`

Examples

```
## Not run:
# Define method
my_method <- function(data, params) {
  fit_m <- lm(M ~ X, data = data)
  fit_y <- lm(Y ~ X + M, data = data)
  list(indirect = coef(fit_m)["X"] * coef(fit_y)["M"])
}

# Configure and run
config <- medsim_config("local")
scenarios <- medsim_scenarios_mediation()

results <- medsim_run(
  method = my_method,
  scenarios = scenarios,
  config = config
```

```
)

# View results
print(results)
summary(results)

## End(Not run)
```

`medsim_run_chunk` *Run one chunk of a chunked SLURM array simulation*

Description

Runs replications `[(chunk_id-1)*chunk_size+1 .. chunk_id*chunk_size]` and saves a single RDS file named `chunk_<chunk_id>.rds` in `output_dir`. Designed to be called from inside a SLURM array task script; `chunk_id` and `n_chunks` are auto-detected from `config` (which auto-reads `SLURM_ARRAY_TASK_ID`).

The chunk `.rds` is the sole artifact: in chunk mode the intermediate per-scenario/summary CSVs that a standalone `medsim_run()` writes are skipped (concurrent array tasks sharing an `output_dir` would clobber the fixed-name files, leaving partial data that looks complete). Result rows record the **global** replication id (schema v2): chunk 2 of a 4-chunk/nsim-20 run writes `replication` 6..10, not 1..5, so ids never collide across chunks and `medsim_combine_chunks()` can audit the combined grid.

Each chunk file also carries a provenance attribute (Gate C): R version, medsim + key dependency versions, hostname, a code SHA, and seconds-per-replication timing. The SHA is auto-detected from the run script's git tree (falling back to a `pkg:medsim-<version>` tag outside git); pass `code_sha` to stamp explicitly. `medsim_combine_chunks()` asserts a single SHA across all chunks – catching a mid-run code edit plus partial resubmit, which would silently mix results from two different code states.

Usage

```
medsim_run_chunk(
  scenarios,
  method,
  config,
  verbose = TRUE,
  code_sha = NULL,
  compute_truth = NULL,
  truth_on_failure = c("stop", "warn")
)
```

Arguments

`scenarios`	A list of `medsim_scenario()` objects.
`method`	A function with signature `method(data, params)`.

`config` A `medsim_config` object with `chunk_id` and `n_chunks` set.

`verbose` Logical: print progress messages.

`code_sha` Character or `NULL`: code-state identifier stamped into the chunk's provenance. `NULL` (default) auto-detects via `git rev-parse HEAD` in the running script's directory, degrading to `pkg:medsim-<version>` outside a git tree.

`compute_truth` Optional function with signature `(data, params)`, passed to `medsim_run()`. Without it the combined result carries no `$truth`, so it supports power analysis but not coverage or bias. Only **chunk 1** computes truth; every other chunk ignores `compute_truth`, so truth is computed once, not once per chunk. Set `config$compute_truth_every_chunk <- TRUE` to compute it in every chunk (useful only when truth is cheap and chunk 1 may not survive). Truth is seeded from `config$seed`, so chunks that compute it agree, and `medsim_combine_chunks()` keeps one row per scenario from whichever chunks carry it. The chunk's provenance records `truth_requested`, so a combine that should have truth but does not raises a `truth_coverage` violation.

Chunk 1 then runs longer than the others (truth plus its replications); size `--time` for it, not from a pilot of another chunk. To keep chunk 1 balanced, compute truth once before submitting: a short `medsim_run()` over the same scenarios, `compute_truth` and `output_dir` warms the truth cache, and chunk 1 then loads the truth instead of computing it. Keep passing `compute_truth` to the chunks: it is what records `truth_requested`, so a lost truth is still reported at combine.

`truth_on_failure` `"stop"` (default) or `"warn"`, passed to `medsim_run()`. Under `"stop"` a failed truth stops chunk 1 before any replication runs, so no chunk file is written and the array task exits nonzero.

Value

Invisibly, the path of the RDS file written. If a task-level error (a `medsim_run_failure` from `medsim_run()`) interrupts the chunk, whatever replications completed are still written – atomically, with provenance `partial = TRUE` – before this function re-`stop()`s so the array task still exits nonzero (`sacct` visibility, not a `--requeue` signal). **This holds even when NO replication completed**: a chunk whose every replication failed writes a flagged, zero-row file rather than no file at all. That file is the only carrier of the `partial = TRUE` marker, and without it a trailing all-failed chunk would combine with zero violations (the contiguity audit cannot see it – the surviving ids are a clean `1..k` block). `medsim_combine_chunks()`/`medsim_audit_results()` flag any such chunk via the `partial_chunk` violation.

See Also

`medsim_combine_chunks()`, `medsim_audit_results()`, `medsim_write_submit_script()`

`medsim_run_parallel` *Run Tasks in Parallel*

Description

Executes a list of tasks in parallel using multiple CPU cores. Provides progress bars, proper error handling, and automatic cluster management.

Usage

```
medsim_run_parallel(
  tasks,
  fun,
  n_cores = NULL,
  progress = TRUE,
  export = NULL,
  packages = NULL,
  cluster_type = NULL,
  seed = NULL
)
```

Arguments

`tasks`	Vector or list: Tasks to process (e.g., 1:100, list of params)
`fun`	Function: Function to apply to each task. Should accept one argument (the task) and return a result.
`n_cores`	Integer: Number of CPU cores to use. If NULL, uses all available cores minus 2.
`progress`	Logical: Show progress bar (default TRUE). Automatically suppressed when running on cluster (batch jobs).
`export`	Character vector: Names of objects to export to workers. These objects must be available in the calling environment.
`packages`	Character vector: Names of packages to load on each worker. Use this if `fun` requires specific packages.
`cluster_type`	Character: Type of cluster ("PSOCK" or "FORK"). FORK is more efficient but only works on Unix. Auto-detected by default.
`seed`	Integer or `NULL`: Master seed for L'Ecuyer-CMRG reproducible per-worker RNG streams. When supplied, calls `RNGkind("L'Ecuyer-CMRG")` and `parallel::clusterSetRNGStream(cl seed)` immediately after cluster creation so each worker gets a distinct, deterministic sub-stream. Default `NULL` leaves each worker's RNG seeded by the OS (non-reproducible). Note: `medsim_run()`/`medsim_run_chunk()` do **not** pass a `seed` here – they get chunk/worker-independent reproducibility from `.medsim_det_seed()` instead (see `?medsim_run`). This parameter matters only if you call `medsim_run_parallel()` directly.

Details

Cluster Types:

- **PSOCK** (default on Windows): Creates separate R sessions. Requires explicit export of objects and packages. Works on all platforms.
- **FORK** (default on Unix): Copies current R session. More efficient, automatic object sharing, but Unix-only.

Progress Bars:

Progress bars are shown in interactive sessions and local execution, but suppressed on HPC clusters to avoid cluttering log files.

Error Handling:

If a task fails:

- Error is caught and returned as an error object
- Other tasks continue processing
- Check results with `sapply(results, inherits, "error")`

Memory Considerations:

Each worker process requires memory. For large tasks:

- Reduce `n_cores` if running out of memory
- Process tasks in chunks
- Use FORK cluster (Unix) which shares memory

Value

List: Results from applying `fun` to each task (same length as tasks)

See Also

`parallel::makeCluster()`, `parallel::parLapply()`, `pbapply::pblapply()`

Examples

```
## Not run:
# Simple parallel computation
results <- medsim_run_parallel(
  tasks = 1:100,
  fun = function(i) {
    Sys.sleep(0.1)  # Simulate work
    i^2
  },
  n_cores = 4
)

# With exports
my_data <- data.frame(x = 1:10, y = 11:20)

results <- medsim_run_parallel(
```

```
  tasks = 1:10,
  fun = function(i) {
    mean(my_data$x[1:i])
  },
  export = "my_data"
)

# With packages
results <- medsim_run_parallel(
  tasks = 1:10,
  fun = function(i) {
    dplyr::tibble(x = i, y = i^2)
  },
  packages = "dplyr"
)

# Check for errors
errors <- sapply(results, inherits, "error")
if (any(errors)) {
  cat("Errors occurred in:", which(errors), "\n")
}

## End(Not run)
```

`medsim_scenario` *Create Custom Simulation Scenario*

Description

Define a custom scenario for simulation studies. A scenario consists of a name, description, data generation function, and population parameters.

Usage

```
medsim_scenario(
  name,
  description = "",
  data_generator,
  params = list(),
  estimand = NULL
)
```

Arguments

`name`	Character: Descriptive name for the scenario
`description`	Character: Detailed description (optional)
`data_generator`	Function: Takes n (sample size) and returns data.frame

params	List: Known population parameters for validation
estimand	Optional medsim_estimand() object declaring the estimand kind ("point", "interval", "probabilistic", "numeric"). NULL (the default) is treated as kind = "point" throughout the package – full back-compatibility with existing scenarios.

Details

Data Generator Function:

The data_generator function must:

- Accept n (sample size) as first argument
- Return a data.frame with at least: X (treatment), M (mediator), Y (outcome)
- Can include additional variables or covariates

Parameters List:

The params list should include known population values:

- a: X -> M path coefficient
- b: M -> Y path coefficient
- c_prime: X -> Y direct effect
- indirect: True indirect effect (a * b)
- Additional parameters as needed

Value

A scenario object (list with class "medsim_scenario")

See Also

medsim_scenarios_mediation() for standard scenarios

Examples

```
# Simple custom scenario
my_scenario <- medsim_scenario(
  name = "Large Effects",
  description = "Both paths have large effects",
  data_generator = function(n = 200) {
    X <- rnorm(n)
    M <- 0.7 * X + rnorm(n)
    Y <- 0.7 * M + rnorm(n)
    data.frame(X = X, M = M, Y = Y)
  },
  params = list(
    a = 0.7,
    b = 0.7,
    indirect = 0.49
  )
)
```

```
# Generate data
data <- my_scenario$data_generator(n = 100)

# Scenario with covariates
covariate_scenario <- medsim_scenario(
  name = "With Covariates",
  data_generator = function(n = 200) {
    C1 <- rnorm(n)
    C2 <- rbinom(n, 1, 0.5)
    X <- rnorm(n)
    M <- 0.3 * X + 0.2 * C1 + rnorm(n)
    Y <- 0.3 * M + 0.2 * C2 + rnorm(n)
    data.frame(X = X, M = M, Y = Y, C1 = C1, C2 = C2)
  },
  params = list(
    a = 0.3,
    b = 0.3,
    indirect = 0.09,
    gamma_m = 0.2,
    gamma_y = 0.2
  )
)
```

`medsim_scenario_dm` *Construct a differential-misclassification (DM) mediation scenario*

Description

Builds a `medsim_scenario()` for the bounds-pair simulation studies (me-mediator-bounds / me-exposure-recall). The `data_generator` wraps `medrobust::simulate_dm_data(...)@observed`; if **medrobust** is not installed a lightweight synthetic fallback is used so that the harness and `interval`-kind analysis machinery can be unit-tested without the package.

The scenario is tagged with `estimand = medsim_estimand("interval", ...)` so that `medsim_analyze_coverage()` automatically dispatches the Imbens-Manski / partial-ID interval branch.

Usage

```
medsim_scenario_dm(
  name,
  true_params,
  dm_params = list(),
  misclass_type = c("mediator", "exposure")
)
```

Arguments

`name`	Scenario name (character).

true_params	List of true generating parameters passed through to medrobust::simulate_dm_data() and stored in scenario$params. Must include at minimum NDE and NIE (the true natural direct/indirect effects) so that coverage against ground truth can be computed.
dm_params	List of differential-misclassification parameters passed through to medrobust::simulate_dm_data().
misclass_type	"mediator" (default, me-mediator-bounds study) or "exposure" (me-exposure-recall study).

Value

A medsim_scenario() object with estimand = medsim_estimand("interval", params = c("NDE", "NIE"), ...).

See Also

medsim_method_bounds() for the corresponding estimator adapter; medsim_analyze_coverage() for interval-kind coverage analysis.

Examples

```
# Synthetic scenario (no medrobust required):
sc <- medsim_scenario_dm(
  name        = "small_misclass",
  true_params = list(NDE = 0.2, NIE = 0.3, n = 200),
  dm_params   = list(delta = 0.1),
  misclass_type = "mediator"
)
# Data generator uses synthetic fallback when medrobust is absent:
d <- sc$data_generator(n = 50)
head(d)
```

---

medsim_scenario_gauge *Create a gauge-residual (P_med + W) simulation scenario*

Description

Thin wrapper over medsim_scenario() mirroring medsim_scenario_sobol(): an A -> M -> Y linear-Gaussian DGP with covariate C and A*M interaction (kappa), a variance_share estimand, and **closed-form** ground truth for the interventional proportion mediated P_med = IIE/OE and the gauge residual W = R/OE, where R = OE - IDE - IIE.

Corner means theta_{a,a'} = tau_a*a + (tau_m + kappa*a)*beta_a*a' (C centered) give OE = theta11-theta00, IDE = theta10-theta00, IIE = theta01-theta00. At kappa = 0, R = 0 and W = 0.

Usage

```
medsim_scenario_gauge(name, true_params = list())
```

Arguments

name	Character scenario name.
true_params	Named list; defaults beta_a=0.6, tau_a=0.5, tau_m=0.7, kappa=0, gamma_mc=0.5, gamma_yc=0.4

Value

A medsim_scenario with estimand$kind = "variance_share" and params$pmed, params$w closed-form truths.

See Also

medsim_method_gauge(), medsim_scenario_sobol()

Examples

```
sc <- medsim_scenario_gauge("interaction", list(kappa = 0.4))
c(sc$params$pmed, sc$params$w)
```

---

medsim_scenario_missing

Construct a missing-data mediation scenario

Description

Wraps medsim_scenario() with a data_generator that (1) draws complete X -> M -> Y with optional nonnormal residuals (medsim_rnonnormal()), then (2) amputes target under mechanism (medsim_amputate()).

Usage

```
medsim_scenario_missing(
  name,
  true_params,
  mechanism,
  prop = 0.2,
  target = "M",
  nonnormal = NULL
)
```

Arguments

name	Scenario name.
true_params	List of true generating parameters: a, b, cp, residual SDs.
mechanism	One of "MCAR", "MAR", "MNAR".
prop	Target missingness proportion.
target	Column(s) to make missing (default "M").
nonnormal	NULL for normal residuals, or list(skew=, kurtosis=).

Value

A `medsim_scenario` object.

Examples

```
scn <- medsim_scenario_missing(
  name = "MAR_M_30",
  true_params = list(a = 0.39, b = 0.39, cp = 0.14),
  mechanism = "MAR", prop = 0.3
)
scn$name
```

---

medsim_scenario_missing_grid

Build the full factorial of missing-data scenarios

Description

Convenience: expand `SPEC-simulation-design` cells (mechanism x prop x n x effect x nonnormality) into a list of `medsim_scenario_missing()` objects in one call.

Usage

```
medsim_scenario_missing_grid(
  true_params_list,
  mechanisms,
  props,
  nonnormal_list = list(NULL)
)
```

Arguments

`true_params_list`	List of `true_params` lists (one per effect-size cell).
`mechanisms, props`	Character/numeric vectors crossed factorially.
`nonnormal_list`	List of `nonnormal` specs (incl. `NULL` for normal).

Value

A list of `medsim_scenario` objects.

Examples

```
grid <- medsim_scenario_missing_grid(
  true_params_list = list(small = list(a = 0.14, b = 0.14, cp = 0.14)),
  mechanisms = c("MCAR", "MAR"),
  props = c(0.1, 0.3)
)
length(grid)
```

`medsim_scenario_numeric` *Create a numeric accuracy simulation scenario*

Description

Thin wrapper over `medsim_scenario()` for studies that evaluate numerical **accuracy** or **timing** of a computation – not coverage or power. The estimand kind is `"numeric"`, which signals downstream commands to skip coverage/power analysis and produce accuracy/timing tables instead.

Typical use-cases:

- Product-of-three CI: compare DOP, MBCO, Monte-Carlo CI point accuracy and computation time across parameter cells.
- Approximation quality: absolute error and relative error vs. analytic truth.

The `data_generator` for numeric scenarios is optional (default: returns `data.frame()` with `n` rows of `NA`s as a placeholder). Methods that only need `params` can ignore the `data` argument.

Usage

```
medsim_scenario_numeric(
  name,
  true_params = list(),
  data_generator = NULL,
  extra = character()
)
```

Arguments

`name`	Character: scenario name.
`true_params`	Named list of ground-truth parameter values. These are passed to `method(data, params)` as `params`; what counts as "truth" is method-defined. For accuracy studies this typically includes the analytic reference value (e.g. `list(ci_true = 0.95)`).
`data_generator`	Function `function(n)` returning a data frame. Defaults to a no-op that returns an empty data frame (suitable for methods that derive everything from `params` rather than a random sample).

extra — Character vector of additional result columns beyond the mandatory est_error/abs_error/elapsed_sec columns. These are passed to medsim_estimand() and stored on the scenario. (est_error, not error: error is reserved for medsim's failure-row schema and a successful method() returning it is rejected by medsim_run().)

Value

A medsim_scenario object with estimand$kind = "numeric".

See Also

medsim_estimand(), medsim_scenario(), medsim_analyze()

Examples

```
# Product-of-three scenario cell
sc <- medsim_scenario_numeric(
  name        = "dop_small_n",
  true_params = list(a  = 0.5, b = 0.3, se_a = 0.1, se_b = 0.1,
                     ci_true = 0.01),
  extra       = c("elapsed_sec")
)
str(sc$estimand)
```

---

medsim_scenario_pmed *Create a P_med simulation scenario*

Description

Thin wrapper over medsim_scenario() that bakes in:

- A linear structural equation model (SEM) data generator for the A -> M -> Y causal chain.
- An estimand = medsim_estimand("probabilistic", params = "pmed", ci = "mbco", extra = "branch_switch") descriptor so that medsim_analyze_coverage() dispatches the MBCO-CI coverage branch.
- An exact closed-form ground-truth P_med: under the all-Gaussian linear SEM the cross-world difference Y1 - Y0 is Normal, so P_med = Phi((beta_ay + alpha_ax * beta_my) / sqrt(2 * (beta_my^2 * sigma_m^2 + sigma_y^2))).

The estimand P_med = P(Y_a=1(M_a=1) > Y_a=0(M_a=1)) + 0.5 * P(Y_a=1(M_a=1) == Y_a=0(M_a=1)) uses the cross-world assumption – it is a *probabilistic* mediation effect, not a difference in expectations.

Usage

```
medsim_scenario_pmed(name, true_params = list())
```

Arguments

name	Character: scenario name passed to medsim_scenario().
true_params	Named list with entries: • alpha_ax: path coefficient A -> M (default 0.5) • beta_my: path coefficient M -> Y (default 0.5) • beta_ay: direct path A -> Y (default 0.0; set 0 for perfect mediation) • sigma_m: residual SD for M (default 1.0) • sigma_y: residual SD for Y (default 1.0)

Value

A medsim_scenario object with estimand$kind = "probabilistic".

See Also

medsim_method_pmed_mbco(), medsim_analyze_coverage()

Examples

```
sc <- medsim_scenario_pmed(
  name        = "full_mediation",
  true_params = list(alpha_ax = 0.6, beta_my = 0.5, beta_ay = 0.0)
)
str(sc$estimand)
```

---

medsim_scenario_sobol *Create a Sobol / functional-ANOVA variance-share simulation scenario*

Description

Thin wrapper over medsim_scenario() that bakes in:

- A linear-Gaussian-with-interaction data generator for the A -> M -> Y causal chain with a single covariate C and an A*M interaction term (kappa).
- An estimand = medsim_estimand("variance_share", params = "pmed_sobol", ci = "standard") descriptor, so medsim_analyze_coverage() uses the generic Wald truth %in% CI path and medsim_validate_scenario() checks for the A/M/Y/C columns the Sobol estimator requires.
- A **closed-form** truth for the Sobol share, exact for the linear-Gaussian-with-interaction DGP (no Monte-Carlo PO draw needed).

The estimand is the variance-scale proportion mediated $P^{\sigma^2}_{med} = V_{med}/V_T$, where the functional-ANOVA variance components are built from the four corner means $\theta_{a,a'} = E[Y(a, M(a'))]$. For the linear-Gaussian DGP with centered C,

$$\theta_{a,a'} = \tau_a a + (\tau_m + \kappa a)\,\beta_a a'.$$

At kappa = 0 the share reduces to $NIE^2/(NIE^2 + NDE^2)$ with $NIE = \tau_m \beta_a$ and $NDE = \tau_a$.

Usage

```
medsim_scenario_sobol(name, true_params = list(), pd = 0.5, pm = 0.5)
```

Arguments

name	Character: scenario name passed to medsim_scenario().
true_params	Named list with entries (defaults in parentheses): • beta_a: path coefficient A -> M (0.6) • tau_a: direct path A -> Y (0.5) • tau_m: path coefficient M -> Y (0.7) • kappa: A*M interaction coefficient in the Y model (0.0) • gamma_mc: covariate effect C -> M (0.5) • gamma_yc: covariate effect C -> Y (0.4) • p_a: treatment-assignment probability used for the variance weights pd = pm = p_a (0.5)
pd, pm	Bernoulli weights for the direct / mediator design variances in the functional-ANOVA decomposition (default 0.5; override only for non-balanced designs).

Value

A medsim_scenario object with estimand$kind = "variance_share" and params$pmed_sobol holding the closed-form ground truth.

See Also

medsim_method_sobol(), medsim_analyze_coverage(), medsim_estimand()

Examples

```
sc <- medsim_scenario_sobol(
  name        = "interaction",
  true_params = list(beta_a = 0.6, tau_a = 0.5, tau_m = 0.7, kappa = 0.4)
)
sc$params$pmed_sobol   # closed-form Sobol share
```

---

medsim_scenarios_mediation

Create Standard Mediation Scenarios

Description

Creates a list of standard mediation scenarios for simulation studies. These scenarios cover common patterns in mediation analysis including independent paths, various correlation structures, suppression effects, and non-standard conditions.

Usage

```
medsim_scenarios_mediation()
```

Details

Standard Scenarios:

1. **Independent Paths**: No correlation between X, M, Y
2. **Moderate Correlation**: rho = 0.3 between all pairs
3. **High Correlation**: rho = 0.7 between all pairs
4. **Suppression**: Mixed positive and negative correlations
5. **Non-zero Effects**: Small to moderate true effects
6. **Unequal Variances**: Different residual variances

Each scenario generates data with:

- Sample size n (default: 200)
- Treatment X, Mediator M, Outcome Y
- Known population parameters for validation

Value

A list of scenario objects, each with name, description, and data_generator function

See Also

`medsim_scenario()` for creating custom scenarios

Examples

```
# Get all standard scenarios
scenarios <- medsim_scenarios_mediation()

# See scenario names
sapply(scenarios, function(s) s$name)

# Generate data from first scenario
data <- scenarios[[1]]$data_generator(n = 100)
head(data)

# Access scenario parameters
scenarios[[1]]$params
```

medsim_summarize_branch_switch *Summarize the MBCO majority-branch rate per scenario*

Description

Reads the `branch_switch` column emitted by `medsim_method_mbco_mi()`: for each replicate, the union-null (`ab = 0`) branch chosen by the majority of imputations (1 = the `a = 0` branch, 0 = the `b = 0` branch). The returned `branch_switch_rate` is therefore the share of replicates whose majority branch is `a = 0`, not the share in which imputations disagree; that is the `branch_mix` column. The result is suitable for joining as a column onto a `medsim_table_coverage()` table. Non-converged rows (`converged == 0`) are dropped before summarizing.

Usage

```
medsim_summarize_branch_switch(results, by = "scenario")
```

Arguments

`results`	A `medsim_results` object from `medsim_run()`.
`by`	Grouping columns (default `"scenario"`).

Value

A data.frame: one row per group with `branch_switch_rate` and `n_valid`.

Examples

```
res <- data.frame(
  scenario = rep(c("s1", "s2"), each = 4),
  branch_switch = c(1, 1, 0, 1, 0, 0, 1, 0),
  converged = 1
)
medsim_summarize_branch_switch(res)
```

medsim_table_accuracy *Generate Accuracy Table*

Description

Creates a publication-ready LaTeX table showing accuracy metrics (bias, MAE, RMSE) from simulation analysis results.

Usage

```
medsim_table_accuracy(
  analysis,
  digits = 3,
  metrics = c("bias", "mae", "rmse"),
  by_scenario = TRUE,
  caption = NULL,
  label = "tab:accuracy"
)
```

Arguments

analysis	A medsim_analysis object from `medsim_analyze()`
digits	Integer: number of decimal places (default: 3)
metrics	Character vector: which metrics to include. Options: "bias", "mae", "rmse", "median_ae", "max_ae", "relative_bias" (default: c("bias", "mae", "rmse"))
by_scenario	Logical: include scenario breakdown (default: TRUE)
caption	Character: table caption (default: auto-generated)
label	Character: LaTeX label for cross-referencing (default: "tab:accuracy")

Value

A medsim_table object (character vector with LaTeX code)

Examples

```
## Not run:
results <- medsim_run(method, scenarios, config)
analysis <- medsim_analyze(results)

# Generate table
table <- medsim_table_accuracy(analysis)
print(table)  # Preview

# Write to file
medsim_write_table(table, "tables/accuracy.tex")

## End(Not run)
```

`medsim_table_comparison`

Generate Method Comparison Table

Description

Creates a publication-ready LaTeX table comparing multiple methods across accuracy, timing, and coverage metrics.

Usage

```
medsim_table_comparison(
  comparison,
  metrics = c("accuracy", "timing"),
  caption = NULL,
  label = "tab:comparison"
)
```

Arguments

comparison	A medsim_comparison object from medsim_compare_methods()
metrics	Character vector: which metrics to include ("accuracy", "timing", "coverage")
caption	Character: table caption
label	Character: LaTeX label

Value

A medsim_table object

---

medsim_table_coverage *Generate Coverage Table*

Description

Creates a publication-ready LaTeX table showing confidence interval coverage rates.

Usage

```
medsim_table_coverage(
  coverage,
  expected = 0.95,
  by_scenario = TRUE,
  caption = NULL,
  label = "tab:coverage"
)
```

Arguments

coverage	A medsim_coverage object from medsim_analyze_coverage()
expected	Numeric: expected/nominal coverage rate (default: 0.95)
by_scenario	Logical: show coverage by scenario (default: TRUE)
caption	Character: table caption
label	Character: LaTeX label

Value

A medsim_table object

---

medsim_table_performance *Render a performance summary as a LaTeX table*

Description

Render a performance summary as a LaTeX table

Usage

```
medsim_table_performance(perf)
```

Arguments

perf	Output of medsim_analyze_performance().

Value

list(latex=, type="performance").

---

medsim_table_power *Generate Power Table*

Description

Creates a publication-ready LaTeX table showing empirical power (rejection rates) from simulation analysis.

Usage

```
medsim_table_power(
  power,
  by_scenario = TRUE,
  caption = NULL,
  label = "tab:power"
)
```

Arguments

power	A medsim_power object from medsim_analyze_power()
by_scenario	Logical: show power by scenario (default: TRUE)
caption	Character: table caption
label	Character: LaTeX label

Value

A medsim_table object

`medsim_table_timing` *Generate Timing Comparison Table*

Description

Creates a publication-ready LaTeX table comparing computational time across methods.

Usage

```
medsim_table_timing(
  results_list,
  metric = "mean",
  include_speedup = TRUE,
  reference_method = NULL,
  by_scenario = FALSE,
  caption = NULL,
  label = "tab:timing"
)
```

Arguments

`results_list`	Named list of medsim_results objects, one per method
`metric`	Character: timing metric to use ("mean", "median", "total")
`include_speedup`	Logical: include speedup column relative to slowest method
`reference_method`	Character: method name to use as reference for speedup (default: NULL uses slowest method)
`by_scenario`	Logical: show timing by scenario (default: FALSE)
`caption`	Character: table caption
`label`	Character: LaTeX label

Value

A medsim_table object

Examples

```
## Not run:
table <- medsim_table_timing(list(
  Proposed = results1,
  Delta = results2,
  Bootstrap = results3
))

## End(Not run)
```

`medsim_tables` *Generate all standard LaTeX tables from simulation results*

Description

Convenience wrapper around `medsim_tables_workflow()` with sensible defaults.

Usage

```
medsim_tables(results, output_dir = "tables", format = "latex", ...)
```

Arguments

`results`	A `medsim_results` object from `medsim_run()`.
`output_dir`	Directory to write tables to. Created if it doesn't exist.
`format`	Output format passed to `medsim_tables_workflow()` (e.g. `"latex"`, `"markdown"`).
`...`	Additional arguments passed to `medsim_tables_workflow()`.

Value

Invisibly, the result of `medsim_tables_workflow()`.

`medsim_tables_workflow`
Generate All Tables

Description

Convenience function to generate all tables from simulation results and save them to an output directory.

Usage

```
medsim_tables_workflow(
  results,
  output_dir,
  tables = "all",
  format = "latex",
  prefix = "table_"
)
```

Arguments

results	A medsim_results object from medsim_run()
output_dir	Character: output directory for tables
tables	Character vector: which tables to generate ("accuracy", "coverage", "power", "all")
format	Character: output format ("latex", "csv", "markdown")
prefix	Character: prefix for file names (default: "table_")

Value

Invisibly returns a list of generated file paths

Examples

```
## Not run:
results <- medsim_run(method, scenarios, config)

# Generate all tables
files <- medsim_tables_workflow(
  results,
  output_dir = "manuscript/tables",
  format = "latex"
)

## End(Not run)
```

---

medsim_validate_scenario

Validate Scenario

Description

Checks that a scenario object is valid and can generate data correctly.

Usage

```
medsim_validate_scenario(scenario, n = 10)
```

Arguments

scenario	A medsim_scenario object
n	Sample size to test (default: 10)

Value

TRUE if valid, throws error otherwise

Examples

```
scenarios <- medsim_scenarios_mediation()
medsim_validate_scenario(scenarios[[1]])
```

---

medsim_workflow *Generate analysis, figures, and tables from simulation results*

Description

End-to-end output pipeline: runs medsim_analyze(), medsim_figures(), and medsim_tables() and writes everything to output_dir.

Usage

```
medsim_workflow(
  results,
  output_dir = "results",
  figures_format = "pdf",
  tables_format = "latex",
  ...
)
```

Arguments

results	A medsim_results object from medsim_run().
output_dir	Top-level directory for all outputs. Subdirectories figures/ and tables/ are created underneath. Created if it doesn't exist.
figures_format	File format for figures (e.g. "pdf", "png").
tables_format	Output format for tables (e.g. "latex", "markdown").
...	Additional arguments passed to medsim_figures().

Value

Invisibly, a list with elements analysis, figures (file paths), and tables (return value of medsim_tables_workflow()).

Examples

```
## Not run:
config <- medsim_config("test")
scenarios <- medsim_scenarios_mediation()
results <- medsim_run(my_method, scenarios, config)
medsim_workflow(results, output_dir = "results")

## End(Not run)
```

`medsim_write_submit_script` *Write a SLURM array submit script for chunked simulation jobs*

Description

Generates a `submit_array.sh` bash script targeting the UNM CARC Hopper cluster (and compatible SLURM systems). The script submits one SLURM array job per scenario chunk; each task runs an R script that calls `medsim_run_chunk()`.

The emitted script is fail-loud (Gate B of the chunked-run integrity layer): a login shell (`#!/bin/bash -l` – on Hopper `module` is only defined in login shells), `set -eo pipefail`, a hard-failing `module load` (never `|| true`), a `command -v Rscript` pre-check, `#SBATCH --requeue`, and the `Rscript` call as the final command so its exit code is the task's exit code. There is deliberately NO output-file gate in the script: the writer's config and the run script's runtime config are independent, so a baked-in path could fail successful tasks; completeness is audited at combine time by `medsim_combine_chunks()` instead. Set `config$array_throttle = K` to cap concurrently-running array tasks (`--array=1-N%K`).

Usage

```
medsim_write_submit_script(
  config,
  run_script = "run_simulation_chunk.R",
  output_file = "submit_array.sh",
  account = NULL
)
```

Arguments

`config`	A `medsim_config` object (from `medsim_config()`). Must have `n_chunks` > 0 and `mode == "cluster"`. Optional: `array_throttle`.
`run_script`	Character: path (on the cluster) to the per-chunk R script (the one that calls `medsim_run_chunk()`). Default `"run_simulation_chunk.R"`.
`output_file`	Character: where to write the bash script. Default `"submit_array.sh"` in the current directory.
`account`	Character: SLURM account/PI account string (e.g. `"pi-dtofighi"`). If `NULL`, no `--account` line is emitted.

Value

Invisibly, the path to the written script.

See Also

medsim_run_chunk(), medsim_combine_chunks()

Examples

```
cfg <- medsim_config(
  mode = "cluster", n_replications = 1000, n_chunks = 10,
  walltime = "04:00:00", mem_per_cpu = "8G"
)
tmp <- tempfile(fileext = ".sh")
medsim_write_submit_script(cfg, output_file = tmp)
# cat(readLines(tmp), sep = "\n")
```

---

medsim_write_table *Write Table to File*

Description

Writes a medsim_table object to a file in LaTeX, CSV, or Markdown format.

Usage

```
medsim_write_table(table, file, format = NULL)
```

Arguments

table	A medsim_table object
file	Character: output file path
format	Character: output format ("latex", "csv", "markdown") If NULL, inferred from file extension.

Value

Invisibly returns the file path

Examples

```
## Not run:
table <- medsim_table_accuracy(analysis)

# Write LaTeX
medsim_write_table(table, "tables/accuracy.tex")
```

```
# Write CSV
medsim_write_table(table, "tables/accuracy.csv", format = "csv")

## End(Not run)
```

---

print.medsim_analysis *Print Analysis Results*

Description

Print Analysis Results

Usage

```
## S3 method for class 'medsim_analysis'
print(x, ...)
```

Arguments

x	A medsim_analysis object
...	Additional arguments (ignored)

Value

Invisibly returns x

---

print.medsim_comparison *Print Method Comparison*

Description

Print Method Comparison

Usage

```
## S3 method for class 'medsim_comparison'
print(x, ...)
```

Arguments

x	A medsim_comparison object
...	Additional arguments (ignored)

Value

Invisibly returns x

print.medsim_config *Print Configuration Summary*

Description

Print Configuration Summary

Usage

```
## S3 method for class 'medsim_config'
print(x, ...)
```

Arguments

x	A medsim_config object
...	Additional arguments (ignored)

Value

Invisibly returns x

print.medsim_coverage *Print Coverage Results*

Description

Print Coverage Results

Usage

```
## S3 method for class 'medsim_coverage'
print(x, ...)
```

Arguments

x	A medsim_coverage object
...	Additional arguments (ignored)

Value

Invisibly returns x

print.medsim_power *Print Power Results*

Description

Print Power Results

Usage

```
## S3 method for class 'medsim_power'
print(x, ...)
```

Arguments

x	A medsim_power object
...	Additional arguments (ignored)

Value

Invisibly returns x

print.medsim_results *Print Simulation Results*

Description

Print Simulation Results

Usage

```
## S3 method for class 'medsim_results'
print(x, ...)
```

Arguments

x	A medsim_results object
...	Additional arguments (ignored)

Value

Invisibly returns x

print.medsim_scenario *Print Scenario Summary*

Description

Print Scenario Summary

Usage

```
## S3 method for class 'medsim_scenario'
print(x, ...)
```

Arguments

x	A medsim_scenario object
...	Additional arguments (ignored)

Value

Invisibly returns x

print.medsim_table *Print medsim_table*

Description

Print medsim_table

Usage

```
## S3 method for class 'medsim_table'
print(x, ...)
```

Arguments

x	A medsim_table object
...	Additional arguments (ignored)

Value

Invisibly returns x

summary.medsim_results *Summarize Simulation Results*

Description

Summarize Simulation Results

Usage

```
## S3 method for class 'medsim_results'
summary(object, ...)
```

Arguments

object	A medsim_results object
...	Additional arguments (ignored)

Value

Summary data.frame

Index

.medsim_det_seed(), *10*
.medsim_failure_split(), *7*

ggplot2::ggsave(), *32*

medsim_amputate, 3
medsim_amputate(), *58*
medsim_analyze, 4
medsim_analyze(), *7*, *10*, *23*, *61*, *66*, *72*
medsim_analyze_coverage, 6
medsim_analyze_coverage(), *5*, *10*, *21*, *23*, *30*, *33–35*, *39*, *40*, *42*, *56*, *57*, *61–63*, *67*
medsim_analyze_performance, 8
medsim_analyze_performance(), *68*
medsim_analyze_power, 9
medsim_analyze_power(), *7*, *68*
medsim_audit_results, 10
medsim_audit_results(), *21*, *51*
medsim_cache_clear, 12
medsim_cache_clear(), *19*
medsim_cache_exists, 13
medsim_cache_exists(), *17*
medsim_cache_info, 14
medsim_cache_init, 15
medsim_cache_list, 15
medsim_cache_load, 16
medsim_cache_load(), *19*
medsim_cache_save, 18
medsim_cache_save(), *17*
medsim_check_results, 19
medsim_combine_chunks, 20
medsim_combine_chunks(), *8*, *10–12*, *50*, *51*, *73*, *74*
medsim_compare_configs, 22
medsim_compare_methods, 22
medsim_compare_methods(), *5*, *67*
medsim_config, 24
medsim_config(), *47*, *49*, *73*
medsim_detect_cores, 27
medsim_detect_environment, 28
medsim_detect_environment(), *26*
medsim_estimand, 29
medsim_estimand(), *7*, *55*, *61*, *63*
medsim_estimate_speedup, 30
medsim_figures, 31
medsim_figures(), *72*
medsim_get_optimal_cores, 32
medsim_method_bounds, 33
medsim_method_bounds(), *57*
medsim_method_gauge, 34
medsim_method_gauge(), *58*
medsim_method_ipw, 35
medsim_method_mbco_mi, 36
medsim_method_mbco_mi(), *65*
medsim_method_mc_ci, 37
medsim_method_pmed_mbco, 38
medsim_method_pmed_mbco(), *62*
medsim_method_sobol, 39
medsim_method_sobol(), *29*, *63*
medsim_nsim_for_mcse, 40
medsim_plot_combined_panel, 41
medsim_plot_coverage, 42
medsim_plot_error_boxplot, 43
medsim_plot_se_vs_estimate, 45
medsim_plot_timing, 45
medsim_rnonnormal, 46
medsim_rnonnormal(), *58*
medsim_run, 47
medsim_run(), *5*, *6*, *8*, *9*, *17*, *25*, *32*, *33*, *38*, *43*, *45*, *49–52*, *61*, *65*, *70–72*
medsim_run_chunk, 50
medsim_run_chunk(), *12*, *21*, *25*, *52*, *73*, *74*
medsim_run_parallel, 52
medsim_run_parallel(), *19*, *25*, *48*, *49*
medsim_scenario, 54
medsim_scenario(), *29*, *30*, *47*, *49*, *50*, *56–58*, *60–64*
medsim_scenario_dm, 56

medsim_scenario_dm(), *34*
medsim_scenario_gauge, 57
medsim_scenario_gauge(), *35*
medsim_scenario_missing, 58
medsim_scenario_missing(), *59*
medsim_scenario_missing_grid, 59
medsim_scenario_numeric, 60
medsim_scenario_pmed, 61
medsim_scenario_pmed(), *39*
medsim_scenario_sobol, 62
medsim_scenario_sobol(), *29*, *40*, *57, 58*
medsim_scenarios_mediation, 63
medsim_scenarios_mediation(), *47*, *55*
medsim_summarize_branch_switch, 65
medsim_table_accuracy, 65
medsim_table_comparison, 66
medsim_table_coverage, 67
medsim_table_performance, 68
medsim_table_power, 68
medsim_table_timing, 69
medsim_tables, 70
medsim_tables(), *72*
medsim_tables_workflow, 70
medsim_tables_workflow(), *70*, *72*
medsim_validate_scenario, 71
medsim_validate_scenario(), *29*, *62*
medsim_workflow, 72
medsim_write_submit_script, 73
medsim_write_submit_script(), *51*
medsim_write_table, 74

parallel::makeCluster(), *53*
parallel::parLapply(), *53*
pbapply::pblapply(), *53*
print.medsim_analysis, 75
print.medsim_comparison, 75
print.medsim_config, 76
print.medsim_config(), *26*
print.medsim_coverage, 76
print.medsim_power, 77
print.medsim_results, 77
print.medsim_scenario, 78
print.medsim_table, 78

summary.medsim_results, 79