

Package: multigrain (via r-universe)

October 8, 2026

Title Optimise Graphical Approaches to Multiple Testing Procedures

Version 1.0.0

Description Find the optimal graphical multiple testing procedure for confirmatory clinical trials that test several hypotheses across endpoints, doses, or subpopulations. Choose the graph maximising an explicit gain-function: a user-defined measure of trial success that scores each possible set of rejected hypotheses. Given simulated p-values, search over hypothesis weights and transition weights to maximise expected trial success. Conventional power criteria are recovered and the family-wise error rate is strongly controlled due to the search being confined to graphical procedures. Described in Spiers, Grayling, Wheeler, and Mander (2026) ``Gain-function optimisation of graphical multiple testing procedures for confirmatory clinical trials" <doi:10.48550/arXiv.2609.19994>.

License LGPL (>= 3)

URL <https://github.com/GSK-Biostatistics/multigrain>,
<https://gsk-biostatistics.github.io/multigrain/>

BugReports <https://github.com/GSK-Biostatistics/multigrain/issues>

Depends R (>= 4.1.0)

Imports cli, dplyr, GA, ggplot2, ggraph, glue, igraph, mvtnorm, nloptr, purrr, Rcpp, RcppParallel (>= 5.1.10), rlang (>= 1.2.0), stats, tibble, tidyr, utils, vctrs, withr

Suggests gMCPLite, spelling, svglite, testthat (>= 3.0.0), vdiff

LinkingTo Rcpp, RcppParallel

Config/Needs/website rmarkdown, gMCP, graphicalMCP, gridExtra

Config/roxygen2/markdown TRUE

Config/roxygen2/version 8.1.0

Config/testthat/edition 3

Config/testthat/parallel true

Config/testthat/start-first optimisation*

Encoding UTF-8

Language en-GB

LazyData true

SystemRequirements GNU make

Config/pak/sysreqs

cmake libfontconfig1-dev libfreetype6-dev libglpk-dev make libicu-dev libxml2-dev

Repository <https://dragosmg.r-universe.dev>

Date/Publication 2026-10-08 09:40:46 UTC

RemoteUrl <https://github.com/GSK-Biostatistics/multigrain>

RemoteRef HEAD

RemoteSha 7884824865c9324b29883320d360d71d14a8781a

Contents

autoplot.multigrain_graph_constraint	3
autoplot.multigrain_graph_optimal	4
calc_ncp	5
calc_power_pvals	6
control_global	9
control_local	10
control_nsim_local	10
graph_constraint	11
graph_constraint_free	13
graph_optimal_example	13
graph_optimal_get_control	14
graph_optimise	14
graph_random	17
is_graph_valid	18
multigrain_control	20
multigrain_verbosity	20
normalise_sum	21
simulate_pvalues	22
trial_success	24

Index

26

autoplot.multigrain_graph_constraint

Autoplot method for multigrain_graph_constraint objects

Description

Autoplot method for multigrain_graph_constraint objects

Usage

```
## S3 method for class 'multigrain_graph_constraint'
autoplot(object, ..., root = NULL, digits = NULL, title = NULL)
```

```
## S3 method for class 'multigrain_graph_constraint'
plot(x, ..., root = NULL, digits = NULL, title = NULL)
```

Arguments

object	A multigrain_graph_constraint object.
...	Additional arguments.
root	A numeric vector indicating which nodes to be regarded as root in the tree layout. Passed down to igraph::layout_as_tree() .
digits	Number of decimal places to round to (between 0 and 3). Defaults to NULL which will choose the number of digits based on how crowded the graph plot will be: • a single digit if more than 8 hypotheses • 2 digits if 5 to 8 hypotheses • 3 digits if 4 or fewer hypotheses
title	An optional plot title.
x	A multigrain_graph_constraint object.

Details

Both plot() and autoplot() methods require an optional dependency, [ggplot2](#).

Value

A [gggraph::gggraph\(\)](#) object.

Examples

```
library(ggplot2)
gc1 <- graph_constraint(
  c(NA, NA, 0, 0, 0),
  matrix(
    c(
```

```

      0, 0.8, 0.2, 0, 0,
      NA, 0, 0, NA, 0,
      NA, 0.8, 0, 0.1, NA,
      0.9, NA, NA, 0, NA,
      0, 0, 0, 1, 0
    ),
    nrow = 5,
    byrow = TRUE
  )
)

autoplot(gc1)

# If you want to control which nodes are treated as root, you can pass them
# as a numeric vector via the `root` argument. For example, we want nodes 1
# and 3 to be plotted as root:
autoplot(gc1, root = c(1, 3))

# You can control the number of decimal places to round to
autoplot(gc1, digits = 1)

# You can supply a title
autoplot(gc1, title = "My graph constraint")

```

autoplot.multigrain_graph_optimal

Autoplot method for multigrain_graph_optimal objects

Description

Autoplot method for multigrain_graph_optimal objects

Usage

```
## S3 method for class 'multigrain_graph_optimal'
autoplot(object, ..., root = NULL, digits = NULL, title = NULL)
```

```
## S3 method for class 'multigrain_graph_optimal'
plot(x, ..., root = NULL, digits = NULL, title = NULL)
```

Arguments

object	A multigrain_graph_optimal object.
...	Additional arguments.
root	A numeric vector indicating which nodes to be regarded as root in the tree layout. Passed down to igraph::layout_as_tree() .
digits	Number of decimal places to round to (between 0 and 3). Defaults to NULL which will choose the number of digits based on how crowded the graph plot will be:

- a single digit if more than 8 hypotheses
 - 2 digits if 5 to 8 hypotheses
 - 3 digits if 4 or fewer hypotheses
- title An optional plot title.
- x A multigrain_graph_optimal object.

Details

Both plot() and autoplot() methods require an optional dependency, [ggplot2](#).

Value

A [ggraph::ggraph\(\)](#) object.

Examples

```
library(ggplot2)

# plotting the example graph optimal
autoplot(graph_optimal_example)

# If you want to control which nodes are treated as root, you can pass them
# as a numeric vector via the `root` argument. For example, we do not like
# the default layout and we would like nodes 1 and 2 to be plotted as root:
autoplot(graph_optimal_example, root = c(1, 2))

# control the rounding with the `digits` argument.
autoplot(graph_optimal_example, root = c(1, 2), digits = 2)

# title
autoplot(graph_optimal_example, root = c(1, 2), title = "My graph optimal")
```

calc_ncp

Calculate non-centrality parameter

Description

Calculate the non-centrality parameter of the test statistic that corresponds to the statistical power for a given number of hypotheses in a trial.

Usage

```
calc_ncp(power, alpha = 0.025)
```

Arguments

- power A numeric vector indicating each hypothesis' statistical power (probability of rejecting the null hypothesis if it false).
- alpha A single numeric value representing the overall one-sided significance level. Default is 0.025.

Details

calc_ncp() uses the quantiles of the standard normal distribution to compute the non-centrality parameter. It is given by the difference between the critical value for a standard normal distribution and the quantile corresponding to the specified power.

Value

A numeric vector of non-centrality parameters corresponding to each element in power.

Examples

```
# Basic usage
calc_ncp(power = c(0.8, 0.9))

# Custom significance level
calc_ncp(power = c(0.8, 0.9), alpha = 0.01)
```

calc_power_pvals	<i>Calculate power for a graph-based multiple test procedure using p-values</i>
------------------	---

Description

Calculate multiplicity-adjusted (marginal) power values for a graph-based multiple test procedure applied to a matrix of p-values. Provide both the hyp_weight vector and transition matrix trans_matrix for the graph you wish to evaluate.

Usage

```
calc_power_pvals(
  pvals,
  hyp_weight,
  trans_matrix,
  ...,
  alpha = 0.025,
  custom_power = NULL,
  sum_to_one_constraint = TRUE,
  call = rlang::caller_env()
)
```

Arguments

pvals	A numeric matrix of p-values. Each row represents a simulated trial, and each column corresponds to a hypothesis.
hyp_weight	A numeric vector representing the weights of the nodes (hypotheses) in the graph. All elements must be in the interval [0, 1] and the vector must sum to 1.

trans_matrix	A numeric matrix representing the transition matrix between nodes. All elements must be in the interval [0, 1], with diagonal elements equal to 0, and each row must sum to 1.
...	These dots are for future extensions and must be empty.
alpha	A single numeric value representing the overall one-sided significance level. Default is 0.025.
custom_power	A list of user-defined power functions. Alternatively, a single function or <code>multigrain_trial_success</code> object can be provided. There are two ways to specify this: • Anonymous functions can be provided to specify the success criteria. Functions must take one simulation's logical vector of results as an input, and return a scalar. For example, if one is interested in the power to reject hypotheses 1 and 3 one could specify: <code>f = function(x) {x[1] && x[3]}</code>. If the power of rejecting hypotheses 1 and 2 is also of interest one would use an (optionally named) list: <code>f = list(power1and3 = function(x) {x[1] && x[3]}, power1and2 = function(x) {x[1] && x[2]})</code>. If the list has no names, the functions will be referenced as "func1", "func2", and so on, in the output. The user can also provide a <code>multigrain_trial_success</code> object instead (resulting in a faster calculation). • Instead of anonymous functions, one can pass a <code>multigrain_trial_success</code> object (or within a list as with the anonymous functions). See trial_success() and the examples.
sum_to_one_constraint	A logical value controlling whether to allow graphs where transition matrix rows are not constrained to sum to one, for example in a fixed sequence. Defaults to TRUE.
call	The execution environment of a currently running function, e.g. <code>caller_env()</code> . The function will be mentioned in error messages as the source of the error. See the <code>call</code> argument of abort() for more information.

Details

`calc_power_pvals()` calculates several power metrics:

- *local power* (probability to reject each individual hypothesis),
- *disjunctive power* (the probability to reject at least one hypothesis),
- *conjunctive power* (the probability to reject all hypotheses), and
- the expected number of rejections. Optionally, you can specify user-defined trial success criteria via `custom_power`.

Value

A list containing:

- `local_power`: The local power for each hypothesis: the proportion of simulations in which each hypothesis is rejected.
- `exp_rejections`: The expected number of rejections across all simulations.

- `disj_power`: Disjunctive power: the probability of rejecting at least one hypothesis.
- `conj_power`: Conjunctive power: the probability of rejecting all hypotheses.
- `"..."`: Additional results as specified by user-defined success functions in `custom_power`.

Examples

```
# First we simulate our p-value distribution

power_nominal <- c(0.90, 0.87, 0.73)
corr_matrix <- matrix(
  c(
    1, 0.2, 0.2,
    0.2, 1, 0.2,
    0.2, 0.2, 1
  ),
  nrow = 3,
  byrow = TRUE
)

pvals <- simulate_pvalues(
  power_nominal = power_nominal,
  corr_matrix = corr_matrix
)

# Second we construct our graph (in this case a fixed sequence)

hyp_weights <- c(1, 0, 0)
trans_matrix <- matrix(
  c(
    0, 1, 0,
    0, 0, 1,
    0, 0, 0
  ),
  nrow = 3,
  byrow = TRUE
)

# Third, we construct a list of metrics we wish to evaluate our graph with

trial_success_measure <- trial_success((r1 && r2) || r3)
power_metrics <- list(
  trial_success = trial_success_measure,
  average_power = function(x) {
    x[1] + x[2] + x[3]
  }
)

# Finally we calculate the power of graph conditional on p-value distribution

result <- calc_power_pvals(
  pvals = pvals,
  hyp_weight = hyp_weights,
```

```

    trans_matrix = trans_matrix,
    custom_power = power_metrics,
    sum_to_one_constraint = FALSE # As third row of graph does not sum to 1
  )

  result

```

control_global	<i>Modify global optimisation options</i>
----------------	---

Description

control_global() is for expert use only; it allows you to directly set `GA::ga()` options to access features that are otherwise not available in multigrain.

Usage

```
control_global(.ctrl, ...)
```

Arguments

.ctrl	A multigrain_control object.
...	<dynamic-dots> Name-value pairs of <code>GA::ga()</code> options and their values.

Value

A modified [multigrain_control](#).

Examples

```

# `control_global()` allows you to access `GA::ga()` options that are not
# otherwise exposed by multigrain. For example, in special cases you may
# need to control the population size (the number of candidate graphs
# evaluated at each generation).
# multigrain makes some informed choices, but if you're convinced you want to
# try other values, you can access this (and other) [GA::ga()] options:
multigrain_control() |>
  control_global(pcrossover = 0.2)

```

control_local	<i>Modify local optimisation options</i>
---------------	--

Description

control_local() is for expert use only; it allows you to directly set `nloptr::nloptr()` options to access features that are otherwise not available in multigrain.

Usage

```
control_local(.ctrl, ...)
```

Arguments

.ctrl	A <code>multigrain_control</code> object.
...	<dynamic-dots> Name-value pairs of <code>nloptr::nloptr()</code> options and their values.

Value

A modified `multigrain_control`.

Examples

```
# `control_local()` allows you to access `nloptr::nloptr()` options that are
# not otherwise exposed by multigrain. For example, in special cases you
# may want to try a different local optimisation algorithm. multigrain uses
# "NLOPT_LN_COBYLA", but if you're convinced you want to try other
# algorithms, you can access this (and other) `nloptr::nloptr()` options:
multigrain_control() |>
  control_local(algorithm = "NLOPT_LN_NEWUOA")
```

control_nsim_local	<i>Modify the number of simulations</i>
--------------------	---

Description

control_nsim_local() allows you to set the number of local simulations.
 control_nsim_global() allows you to set the number of global simulations.

Usage

```
control_nsim_local(ctrl, nsim_local)

control_nsim_global(ctrl, nsim_global)
```

Arguments

ctrl	A multigrain_control object.
nsim_local	The number of simulations to use when evaluating the trial success function in local optimisation. • If set and lower than the number of rows in the pvals matrix, <code>graph_optimise()</code> will use a random sample of <code>nsim_local</code> rows from pvals for local optimisation. • If unset or greater than or equal to the number of rows in the pvals matrix, <i>all</i> rows from pvals will be used.
nsim_global	The number of simulations to use when evaluating the trial success function in global optimisation. If unset, the minimum between 50000 and the number of sets of p-values will be used.

Value

A modified [multigrain_control](#).

Examples

```
multigrain_control() |>
  control_nsim_local(10000)

multigrain_control() |>
  control_nsim_global(10000)

multigrain_control() |>
  control_nsim_local(10000) |>
  control_nsim_global(5000)
```

graph_constraint	<i>Create a graph constraint for optimisation procedures</i>
------------------	--

Description

A *graph_constraint* object defines constraints on the hypothesis weight vector and transition matrix for optimisation of graph-based multiple testing procedures.

Usage

```
graph_constraint(
  hyp_constraint = NULL,
  trans_constraint = NULL,
  ...,
  names = "auto",
  diagnose = FALSE,
  tolerance = sqrt(.Machine$double.eps)
)
```

Arguments

hyp_constraint	A numeric vector defining the constraints on the hypothesis weight vector. If NULL, all hypothesis weights are free parameters to be optimised.
trans_constraint	A numeric matrix defining the constraints on the transition matrix between hypotheses. If NULL, all transition matrix weights are free parameters to be optimised, except for diagonal elements which remain set to 0.
...	These dots are for future extensions and must be empty.
names	An optional character vector containing hypotheses' names. If not provided it defaults to "auto" meaning the hypotheses will be automatically named "H1", "H2", and so on.
diagnose	A logical value enabling detailed diagnosis. Default is FALSE.
tolerance	numeric ≥ 0 . Differences smaller than tolerance will not be reported. The default value is close to $1.5e-8$ - i.e. <code>sqrt(.Machine\$double.eps)</code> (the standard R definition of "practically equal", as used by <code>base::all.equal()</code>).

Details

The object contains a constraint on the weights (`hyp_constraint`) and a constraint on the transition matrix (`trans_constraint`).

The *graph_constraint* object is used to define constraints on both the hypothesis weight vector and the transition matrix in graph-based optimisation procedures. The `graph_optimise()` function will read the graph constraints and only optimise free parameters (specified by NA in `hyp_constraint` and `trans_constraint`).

Either `hyp_constraint` or `trans_constraint` must be provided (they can't both be NULL at the same time). If only one is provided, the *graph_constraint* object will allow `graph_optimise()` to optimise any parameter (i.e., no constraints will be specified) in the other one.

`hyp_constraint` and `trans_constraint` must refer to the same number of hypotheses.

Value

A `multigrain_graph_constraint` object containing:

- `hyp_constraint`: a numeric vector representing the constraints on the hypothesis weight vector.
- `trans_constraint`: a numeric matrix representing the constraints on the transition matrix. If an element is NA, it is a free parameter to be optimised by `graph_optimise()`.

References

Xi, D. and Chen, Y. (2024). Optimal weighted Bonferroni tests and their graphical extensions. *Statistics in Medicine*, 43(3), 475–500. doi:10.1002/sim.9958

Examples

```
# Create a graph_constraint object with predefined weight constraints
graph_constraint(hyp_constraint = c(NA, 0.4, NA))
```

graph_constraint_free *Create an unconstrained graph constraint*

Description

Create a graph constraint that allows for all hypotheses transitions to be optimised.

Usage

```
graph_constraint_free(num_hyp, names = "auto")
```

Arguments

num_hyp	An integer denoting the number of hypotheses. Must be greater than or equal to 2.
names	An optional character vector containing hypotheses' names. If not provided it defaults to "auto" meaning the hypotheses will be automatically named "H1", "H2", and so on.

Value

An unconstrained multigrain_graph_constraint object. In hyp_constraint all values are NA and similarly in the trans_constraint, except for the diagonal which is set to 0. If we have only 2 hypotheses, then the matrix will have 0 on the diagonal and the other 2 elements are set to 1.

Examples

```
# Create a graph constraint object with 3 hypotheses and no constraints
graph_constraint_free(3)

# Create a graph constraint object with 2 hypotheses results in set values in
# the transition matrix
graph_constraint_free(2)
```

graph_optimal_example *Optimised graph example*

Description

An example of an optimised graph with 4 hypotheses and a custom power trial success utility.

Usage

```
graph_optimal_example
```

Format

graph_optimal_example:

A multigrain_graph_optimal object. For more details see [graph_optimise\(\)](#)

graph_optimal_get_control

Get the optimisation control

Description

Retrieve the settings used for the graph optimisation.

Usage

```
graph_optimal_get_control(graph_optimal)
```

Arguments

graph_optimal A multigrain_graph_optimal object.

Value

The multigrain_control object holding the optimisation settings.

Examples

```
# `graph_optimal_example` is an example optimised graph
graph_optimal_get_control(graph_optimal_example)
```

graph_optimise

Optimise graph-based multiple testing procedures

Description

Optimise the hypothesis weights and transition matrix of a graph-based multiple testing procedure using [nloptr::nloptr\(\)](#) local optimisation and, optionally, a genetic algorithm using [GA::ga\(\)](#) for global optimisation.

Usage

```

graph_optimise(
  pvals,
  graph_constraint,
  trial_success,
  ...,
  alpha = 0.025,
  start_graph = list(list(hyp_weight = NULL, trans_matrix = NULL)),
  global_search = TRUE,
  num_threads = 1L,
  control = multigrain_control(),
  verbose = multigrain_verbosity()
)

graph_optimize(
  pvals,
  graph_constraint,
  trial_success,
  ...,
  alpha = 0.025,
  start_graph = list(list(hyp_weight = NULL, trans_matrix = NULL)),
  global_search = TRUE,
  num_threads = 1L,
  control = multigrain_control(),
  verbose = multigrain_verbosity()
)

```

Arguments

<code>pvals</code>	A numeric matrix of p-values. Each row represents a simulated trial, and each column corresponds to a hypothesis.
<code>graph_constraint</code>	A <code>multigrain_graph_constraint</code> object containing constraints on the graph's weights and transition matrix. Created with graph_constraint() .
<code>trial_success</code>	A <code>multigrain_trial_success</code> object defining the trial success measure (power objective) function to maximize. Created with trial_success() .
<code>...</code>	These dots are for future extensions and must be empty.
<code>alpha</code>	A single numeric value representing the overall one-sided significance level. Default is 0.025.
<code>start_graph</code>	Optional. Initial list of graphs suggested. Each graph is defined as a list containing starting weight vector <code>hyp_weight</code> and transition matrix <code>trans_matrix</code> . If <code>NULL</code> , default starting values are generated (currently with a Bonferroni-Holm graph).
<code>global_search</code>	A logical indicating whether to perform a global optimisation before the local optimisation. Defaults to <code>TRUE</code> .

num_threads	Number of threads to use for parallel execution of the shortcut algorithm. On shared systems (HPC clusters, login nodes), always explicitly set num_threads based on your resource allocation. Default is 1L (serial execution).
control	An optional multigrain_control object can be used to set various graph optimisation parameters. Created with <code>multigrain_control()</code> .
verbose	An optional string controlling verbosity ("detail" > "info" > "silent"). Verbosity can also be set at package level with the multigrain_verbosity option (see <code>multigrain_verbosity()</code>): • "info" (default): will only show milestones / informational messages highlighting the progress of the optimisation at coarse-grained level. • "detail": will show milestones and information about fine-grained optimisation events. • "silent": no information about the progress of the optimisation is printed to the console. Errors and warnings are still thrown normally.

Details

The output is a graph where a specified objective function - the *trial success measure* - is maximised under given constraints on the graph structure, conditional on a p-value distribution supplied.

Value

A multigrain_graph_optimal object containing:

- hyp_weight: Optimised hypothesis weights (numeric vector).
- trans_matrix: Optimised transition matrix (numeric matrix).
- constraints: List of constraints used in the optimisation for weights and transition matrix.
- trial_success: The trial success function used in the optimisation.
- power: Power metrics for the optimised graph.
- solution: A list containing:
 - opt_source: Source of the optimal solution (local or global).
 - graph_valid: Named logical vector indicating validity of the local and global solutions (c("local" = TRUE/FALSE, "global" = TRUE/FALSE)).
- global_search: TRUE or FALSE indicating whether a global optimisation was performed.
- control: The modified `multigrain_control` object used. Values passed on by the user are complemented with contextual defaults.
- global_output: Output from the genetic algorithm if global optimisation was performed.
- local_output: Output from the NLOPT optimisation.
- start_graph: Initial starting values used in the optimisation.

References

Spiers, A. D. V., Grayling, M. J., Wheeler, G. M., and Mander, A. P. (2026). Gain-function optimisation of graphical multiple testing procedures for confirmatory clinical trials. *arXiv:2609.19994v1*. <https://arxiv.org/abs/2609.19994>

Examples

```
# Generate test data
pvals <- simulate_pvalues(
  power_nominal = c(0.9, 0.85, 0.8, 0.75),
  corr_matrix = diag(4),
  nsim = 5000
)

# Create trial success function
ts <- trial_success(r1 + r2 + r3 + r4)

# Optimise graph
result <- graph_optimise(
  pvals = pvals,
  graph_constraint = graph_constraint_free(4),
  trial_success = ts,
  num_threads = 2
)
```

graph_random*Generate random graph*

Description

Generate a random graph, consisting of a vector of hypothesis weights and a transition matrix. The weights and transition matrix are randomly generated respecting any constraints provided, ensuring that the sum of the weights equals 1 and each row of the transition matrix sums to 1.

Usage

```
graph_random(m = NULL, graph_constraint = NULL, names = "auto")
```

Arguments

- | | |
|------------------|--|
| m | An integer representing the number of hypotheses. It defines both the length of the weight vector and the dimensions ($m \times m$) of the transition matrix. Optional when <code>graph_constraint</code> is supplied (inferred from constraint dimensions). If both <code>m</code> and <code>graph_constraint</code> are supplied, they must agree. |
| graph_constraint | An optional graph constraint object created by graph_constraint() . When supplied, fixed elements are honoured and only free (NA) positions are randomised. |
| names | An optional character vector containing hypotheses' names. If not provided it defaults to "auto" meaning the hypotheses will be automatically named "H1", "H2", and so on. |

Value

A list containing:

- `hyp_weight`: A numeric vector of length `m` representing the generated hypothesis weights.
- `trans_matrix`: A numeric matrix of dimension `m` x `m` representing the generated transition matrix.

Examples

```
# Generate a random graph for 5 hypotheses
random_graph <- graph_random(5)

# print the weight vector
random_graph$hyp_weight

# print the transition matrix
random_graph$trans_matrix

# Generate a random graph respecting constraints
gc <- graph_constraint(
  hyp_constraint = c(0.5, NA, NA),
  trans_constraint = matrix(c(0, NA, NA, NA, 0, NA, NA, NA, 0), 3, 3)
)

random_graph <- graph_random(graph_constraint = gc)
random_graph
```

`is_graph_valid`

Check the validity of a graph-based MTP

Description

Check that a given weight vector and transition matrix satisfy several conditions to determine if they form a valid graph. It verifies that the matrix is square, the dimensions match the length of the weight vector, the weight vector elements and matrix elements lie within the interval $[0, 1]$, and that the weight vector sums to 1. Additionally, it checks the matrix is properly normalised, with all diagonal elements equal to 0 and each row summing to 1.

Usage

```
is_graph_valid(
  hyp_weight,
  trans_matrix,
  sum_to_one_constraint = TRUE,
  tolerance = sqrt(.Machine$double.eps)
)
```

Arguments

hyp_weight	A numeric vector representing the weights of the nodes (hypotheses) in the graph. All elements must be in the interval [0, 1] and the vector must sum to 1.
trans_matrix	A numeric matrix representing the transition matrix between nodes. All elements must be in the interval [0, 1], with diagonal elements equal to 0, and each row must sum to 1.
sum_to_one_constraint	A logical indicating whether to allow graphs where both the transition matrix rows are not constrained to sum-to-one, for example in a fixed sequence. Defaults to TRUE.
tolerance	numeric ≥ 0 . The tolerance when evaluating the sum-to-one constraints of the hypothesis weights and transition matrix rows. The default value is close to $1.5e-8$ - i.e. <code>sqrt(.Machine\$double.eps)</code> (the standard R definition of "practically equal", as used by <code>base::all.equal()</code>).

Details

The function performs the following checks to determine graph validity:

- `trans_matrix` is a square matrix.
- the length of `hyp_weight` matches the number of rows and columns in `trans_matrix`.
- all diagonal elements of `trans_matrix` are 0.
- all elements of `hyp_weight` and `trans_matrix` lie in the interval [0, 1].
- the sum of `hyp_weight` equals 1.
- each row of `trans_matrix` sums to 1.

If any of these conditions are not satisfied, the function returns FALSE and issues the corresponding warning message.

Value

A logical value: TRUE if the graph is valid, otherwise FALSE. The function issues warnings if any of the validity checks fail.

Examples

```
hyp_weight <- c(0.4, 0.3, 0.3)
trans_matrix <- matrix(
  c(0, 0.5, 0.5, 0.3, 0, 0.7, 0.6, 0.4, 0),
  nrow = 3,
  byrow = TRUE
)
is_graph_valid(hyp_weight, trans_matrix)
```

multigrain_control	<i>Set parameters for graph optimisation</i>
--------------------	--

Description

There are two steps needed to fine tune the graph optimisation with multigrain:

1. Create a control object with `multigrain_control()`.
2. Define its behaviour with `control_` functions:
 - `control_nsim_local()` to set the number of local simulations.
 - `control_nsim_global()` to set the number of global simulations.
 - `control_local()` to modify the local optimisation options.
 - `control_global()` to modify the global optimisation options.

Any unset parameters will automatically be set before running the optimisation. There are predefined defaults, but they are calibrated based on the `pvals` dimensions.

Usage

```
multigrain_control()
```

Value

A `multigrain_control` object.

Examples

```
multigrain_control()
```

multigrain_verbosity	<i>Multigrain verbosity</i>
----------------------	-----------------------------

Description

`multigrain_verbosity()` returns the option named `multigrain_verbosity`, which controls multigrain's verbosity. There are 3 possible levels ("`detail`" > "`info`" > "`silent`");

- "`info`" (default): will only show milestones / informational messages highlighting the progress of the optimisation at coarse-grained level.
- "`detail`": will show milestones and information about fine-grained optimisation events.
- "`silent`": no information about the progress of the optimisation is printed to the console. Errors and warnings are still thrown normally.

Usage

```
multigrain_verbosity()
```

Details

If the `multigrain_verbosity` option is unset, then the "info" level will be used.

Value

A string indicating the verbosity level as set with options or "info" if `multigrain_verbosity` is unset.

Examples

```
multigrain_verbosity()
```

<code>normalise_sum</code>	<i>Normalise graph weights to sum to a target value</i>
----------------------------	---

Description

Ensures that a numeric vector of graph weights is rescaled so its elements sum to a target value (default 1). This is useful when exporting optimised graphs (e.g. to gMCP), where weights must form a valid probability vector.

Usage

```
normalise_sum(
  x,
  ...,
  fixed_idx = integer(0),
  target = 1,
  tolerance = sqrt(.Machine$double.eps)
)
```

Arguments

<code>x</code>	A numeric vector of weights to be normalised.
<code>...</code>	These dots are for future extensions and must be empty.
<code>fixed_idx</code>	An integer vector representing the indices of elements that must not be modified (e.g. diagonal entries in a transition matrix, or positions locked by <code>graph_constraint()</code>). Defaults to <code>integer(0)</code> (no fixed elements).
<code>target</code>	A number representing the desired sum for the output vector. Default is 1.
<code>tolerance</code>	numeric ≥ 0 . The tolerance to be used when checking whether the sum has converged to target. The default value is close to $1.5e-8$ - i.e. <code>sqrt(.Machine\$double.eps)</code> (the standard R definition of "practically equal", as used by <code>base::all.equal()</code>).

Details

Many downstream tools (e.g. gMCP) require hypothesis weights to sum to 1. Direct division by `sum(x)` may leave tiny discrepancies due to floating-point rounding. This helper fixes such issues automatically, ensuring exported graphs are valid.

The algorithm first scales all free (non-fixed) elements proportionally so they sum to `target - sum(x[fixed_idx])`. It then computes the largest free element (the "anchor") as the exact complement of all other elements (`target - sum(x[-anchor])`), absorbing any remaining rounding residual. A single additive fallback pass handles the rare case where the prior scaling introduced enough rounding for the complement to land 1 ULP off.

Value

A numeric vector of the same length as `x`, adjusted so that `sum(x)` is within tolerance of `target`.

Note

Two edge cases return early without enforcing `target`:

- If all elements of `x` are zero, the zero vector is returned unchanged (proportional scaling is undefined for an all-zero input).
- If every index is in `fixed_idx` (no free elements), the input is returned as-is — the caller is responsible for ensuring `sum(x) == target` when no elements may be modified.

Examples

```
x <- c(0.2, 0.3, 0.500000001)
print(sum(x) == 1)

x_sum_to_1 <- normalise_sum(x)
print(all.equal(sum(x_sum_to_1), 1))

# With fixed elements: indices 1 and 3 are held constant
x2 <- c(0.25, 0.45, 0.30)
normalise_sum(x2, fixed_idx = c(1L, 3L))
```

simulate_pvalues

Simulate raw p-values

Description

`simulate_pvalues()` simulates raw p-values from under the alternative hypotheses and the assumption that the distribution of test statistics is a multivariate normal distribution. The most important input parameters are the nominal power of each hypothesis, the significance level, and a correlation matrix.

Usage

```
simulate_pvalues(
  power_nominal,
  ...,
  alpha = 0.025,
  corr_matrix = diag(length(power_nominal)),
  nsim = 1e+05
)
```

Arguments

<code>power_nominal</code>	A numeric vector of nominal power values for each hypothesis.
<code>...</code>	These dots are for future extensions and must be empty.
<code>alpha</code>	A single numeric value representing the overall one-sided significance level. Default is 0.025.
<code>corr_matrix</code>	A numeric matrix representing the correlation matrix Σ of the test statistics.
<code>nsim</code>	An integer indicating the number of simulations to run. Default is 1e5.

Details

It starts by calculating the non-centrality parameter Δ , of the a test statistic z - where $z \sim N(\Delta, 1)$ - based on the nominal power (probability of rejection under the alternative, unadjusted for multiplicity) and significance level. It then generates raw p-values by simulating the test statistics using a multivariate normal distribution with the given correlation matrix.

`simulate_pvalues()` assumes a point global alternative, using a vector of transformed p-values $(\Phi^{-1}(1-p_1), \dots, \Phi^{-1}(1-p_m))$ which follows a multivariate normal distribution with a correlation matrix Σ . Here, Φ^{-1} signifies the inverse function of the standard normal distribution.

This assumption holds, for example, when $p_1, \dots, p_m$ are the raw p-values derived from one-sided z-tests for distinct hypotheses.

Note: applying the transformation $\Phi^{-1}(1-p_i)$ to p-values from two-sided tests does not generally result in a multivariate normal distribution.

Value

A matrix where each row represents a set of simulated raw p-values for each hypothesis.

Examples

```
# Define parameters for simulation
nominal_power <- c(0.8, 0.85, 0.9)
corr <- matrix(
  c(1, 0.5, 0.5,
    0.5, 1, 0.5,
    0.5, 0.5, 1),
  nrow = 3
)
num_simulations <- 1000
```

```
# Simulate raw p-values
pvals <- simulate_pvalues(
  nominal_power,
  corr_matrix = corr,
  nsim = num_simulations
)
```

trial_success	Create a trial success function
---------------	---------------------------------

Description

Create a user-defined **trial-success utility** ψ that assigns value to each rejection pattern from a graphical multiple testing procedure. The function compiles ψ to fast C++ for simulation/optimisation.

Usage

```
trial_success(objective, verbose = multigrain_verbosity())
```

Arguments

- | | |
|-----------|---|
| objective | An expression or string encoding the trial-success utility ψ . The symbols $r_1, r_2, \dots$ refer to rejection indicators for the corresponding hypotheses. To inject values from your R session, use <code>rlang</code> 's injection operator - <code>!!</code> - (see Examples). Arithmetic and logical operators are allowed. |
| verbose | An optional string controlling verbosity ("detail" > "info" > "silent"). Verbosity can also be set at package level with the <code>multigrain_verbosity</code> option (see multigrain_verbosity()): <ul style="list-style-type: none"> • "info" (default): will inform about the successful compilation of the trial success function. • "detail": no additional information available, will have the same effect as "info". • "silent": silent, no information about the trial success function compilation. Errors and warnings are still thrown. |

Details

In code, ψ is written using symbols $r_1, r_2, \dots, r_m$, where each r_i is the binary indicator that hypothesis H_i is rejected by the chosen graph. You can combine these indicators with arithmetic (+ - * /) and logical operators (&&, ||, or the words and, or) to reflect your design priorities (e.g., "any success", "all successes", weighted composites).

The utility ψ is evaluated on each simulated rejection pattern and averaged, yielding the **expected trial success** for a given graph and data-generating scenario. This lets you optimise graphs against the utility that captures your clinical/regulatory goals, rather than a single power summary.

Value

A multigrain_trial_success object made up of:

- func: compiled function that evaluates ψ row-wise on a matrix of rejection indicators and returns the mean utility (i.e., expected trial success under the simulated scenario).
- m: number of hypotheses implied by $r_1, \dots, r_m$.
- objective: the original utility expression (as a string).

References

Spiers, A. D. V., Grayling, M. J., Wheeler, G. M., and Mander, A. P. (2026). Gain-function optimisation of graphical multiple testing procedures for confirmatory clinical trials. *arXiv:2609.19994v1*. <https://arxiv.org/abs/2609.19994>

Examples

```
# Expected number of rejections (equals m × average power)
exp_rejs <- trial_success(r1 + r2 + r3 + r4)

# Disjunctive success: any rejection among four
disj_power <- trial_success(r1 || r2 || r3 || r4)

# Conjunctive success: all four must be rejected
conj_power <- trial_success(r1 && r2 && r3 && r4)

# Composite utility: require H1 AND H2 rejection to get H3 and H4 value
composite <- trial_success((r1 && r2) * (r3 + r4))

# Weighted priorities (e.g., H1 gets weight 2 times H2 or H3)
weighted <- trial_success(2*r1 + r2 + r3)

# Inject values from your environment with !!
w <- 2
trial_success(!!w * r1 + r2 + r3)

# Programmatic string input
expr_str <- sprintf("%s * r1 + r2", w)
trial_success(!!expr_str)
```

Index

- * **datasets**
 - graph_optimal_example, 13
- abort(), 7
- autoplot.multigrain_graph_constraint, 3
- autoplot.multigrain_graph_optimal, 4
- base::all.equal(), 12, 19, 21
- calc_ncp, 5
- calc_power_pvals, 6
- control_global, 9
- control_global(), 20
- control_local, 10
- control_local(), 20
- control_nsim_global
 - (control_nsim_local), 10
- control_nsim_global(), 20
- control_nsim_local, 10
- control_nsim_local(), 20
- GA::ga(), 9, 14
- ggplot2, 3, 5
- ggraph::ggraph(), 3, 5
- graph_constraint, 11
- graph_constraint(), 15, 17, 21
- graph_constraint_free, 13
- graph_optimal_example, 13
- graph_optimal_get_control, 14
- graph_optimise, 14
- graph_optimise(), 12, 14
- graph_optimize(graph_optimise), 14
- graph_random, 17
- igraph::layout_as_tree(), 3, 4
- is_graph_valid, 18
- multigrain_control, 9–11, 16, 20
- multigrain_control(), 16
- multigrain_verbosity, 20
- multigrain_verbosity(), 16, 24
- nloptr::nloptr(), 10, 14
- normalise_sum, 21
- plot.multigrain_graph_constraint
 - (autoplot.multigrain_graph_constraint), 3
- plot.multigrain_graph_optimal
 - (autoplot.multigrain_graph_optimal), 4
- simulate_pvalues, 22
- trial_success, 24
- trial_success(), 7, 15