Package 'grip'

August 21, 2026

Type Package

Title Graph Drawing with Intelligent Placement (GRIP)

Version 0.1.3

Description Implements GRIP multiscale graph layout with a unified choice between hop-count and geometry-aware edge-length graph metrics in 2D and 3D. Provides layout scoring, candidate comparison, multiscale trace diagnostics, synthetic graph families, and advanced experimental geodesic-KK utilities for weighted-layout evaluation and polish. Based on Gajer and Kobourov (2002) <doi:10.7155/jgaa.00052> and Gajer, Goodrich and Kobourov (2004) <doi:10.1016/j.comgeo.2004.03.014>.

License GPL (>= 3)

URL https://pgajer.github.io/grip/, https://github.com/pgajer/grip

BugReports https://github.com/pgajer/grip/issues

Encoding UTF-8

Depends R (>= 3.5.0)

Imports Rcpp

Suggests bslib, DT, FNN, geometry, igraph, htmltools, htmlwidgets, knitr, later, rmarkdown, rgl, shiny, testthat (>= 3.0.0)

LinkingTo Rcpp

SystemRequirements C++17

Config/testthat/edition 3

VignetteBuilder knitr

Config/roxygen2/version 8.0.0

NeedsCompilation yes

Author Pawel Gajer [aut, cre]

Maintainer Pawel Gajer <pgajer@gmail.com>

Repository CRAN

Date/Publication 2026-08-21 05:43:03 UTC

Contents

`grip-package` *grip: Graph dRawing with Intelligent Placement*

Description

Fast multiscale graph layouts in 2D and 3D.

Details

The recommended workflow is `grip()` with `metric = "hop"` for topology-first layouts or `metric = "edge_length"` when positive edge lengths define the graph metric. The package also provides `score.layout()` and `compare.layouts()` for real-data layout selection, `trace.grip()` for multiscale diagnostics under either metric, and advanced public experimental geodesic-KK helpers such as `prepare.edge.kk()`, `prepare.geodesic.kk()`, `score.geodesic.kk()`, `prepare.landmark.geodesic.kk()`, and `score.landmark.geodesic.kk()` for weighted-layout evaluation and refinement. Convenience graph generators such as `edges.path()`, `edges.sierpinski.triangle()`, and `edges.sierpinski.tetrahedron()`, quick plotting via `plot.layout()`, and optional Shiny explorers round out the package.

Author(s)

Maintainer: Pawel Gajer <pgajer@gmail.com>

Authors:

- Pawel Gajer <pgajer@gmail.com>

References

Gajer, P. and Kobourov, S.G. (2002). GRIP: Graph dRawing with Intelligent Placement. *Journal of Graph Algorithms and Applications*, 6(3), 203–224. doi:10.7155/jgaa.00052.

Gajer, P., Goodrich, M.T. and Kobourov, S.G. (2004). A multi-dimensional approach to force-directed layouts of large graphs. *Computational Geometry*, 29(1), 3–18. doi:10.1016/j.comgeo.2004.03.014.

See Also

Useful links:

- https://pgajer.github.io/grip/
- https://github.com/pgajer/grip
- Report bugs at https://github.com/pgajer/grip/issues

`build.misf` *Build a maximal independent set filtration (MISF) for a graph*

Description

'build.misf()' exposes the maximal independent set filtration already constructed internally by the GRIP layout engine. The result is graph-first: you can supply either an edge list plus 'n', or an adjacency/weight-list pair.

Usage

```
build.misf(
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  num_init = 24L,
  num_nbrs = 20L,
  seed = 6L
)
```

Arguments

`edges`	Two-column integer matrix of undirected edges (1-based vertex ids).
`n`	Number of vertices.
`adj_list`	Optional adjacency list (1-based integer vectors).
`weight_list`	Optional positive edge-weight list parallel to 'adj_list'.
`edge_weights`	Optional positive vector parallel to 'edges'.
`num_init`	Target top-level active-set size used by the current GRIP MISF builder. The returned highest MISF level has size at most 'min(num_init, n)'.
`num_nbrs`	Retained local neighborhood budget per MISF level, matching the current GRIP refinement schedule metadata.
`seed`	Optional integer seed passed to the current GRIP graph RNG.

Details

The current implementation returns the same MISF structure used by GRIP's multiscale layout core. Edge weights are accepted and passed through the underlying graph object, but the current MISF construction itself is driven by the graph topology and hop-distance thresholds rather than weighted shortest-path distances.

Value

An object of class ‘"grip_misf"‘ containing:

levels Named list ‘V0, V1, ...‘ of nested vertex sets (1-based).

vertex_depth Integer vector giving the highest MISF level containing each vertex.

mish_order The internal GRIP MISF order (1-based vertex ids).

misf_size Sizes of the nested levels.

num_nbrs_schedule Per-level retained neighborhood counts used by the current GRIP engine.

misf_height Highest MISF level index.

top_level_size Size of the highest MISF level.

Examples

```
edges <- edges.mesh(4, 4)
misf <- build.misf(edges = edges, n = 16, num_init = 6, seed = 1)
misf$misf_size
misf$levels[[1L]]
```

---

build.weighted.misf *Build a weighted MISF hierarchy*

Description

`build.weighted.misf()` builds the weighted max-independent-set filtration used by the weighted GRIP layout core. It is primarily a diagnostic and benchmarking helper for comparing weighted and combinatorial hierarchies on the same graph.

Usage

```
build.weighted.misf(
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  num_init = 24,
  num_nbrs = 20,
  length_normalization = c("median", "mean", "none"),
  seed = 6
)
```

Arguments

edges	Two-column integer matrix of edges (1-based vertex ids).
n	Number of vertices.
adj_list	Adjacency list (1-based) for undirected graphs.
weight_list	Parallel list of strictly positive edge lengths.
edge_weights	Optional vector of edge lengths for edges.
num_init	Number of initial vertices in the coarsest level.
num_nbrs	Maximum number of retained local neighbors per level.
length_normalization	Global edge-length normalization: "median" (default), "mean", or "none".
seed	Optional RNG seed for reproducibility. If NULL, uses current time.

Value

A list with weighted MISF levels, vertex_depth, mish_order, misf_size, num_nbrs_schedule, misf_height, top_level_size, weight_scale, and length_normalization.

---

compare.layouts *Compare multiple layout candidates across seeds*

Description

compare.layouts() computes layouts for several candidate presets or parameter lists, optionally expands a local parameter search, scores each run with score.layout(), and summarizes both quality metrics and seed-to-seed stability. This is the main real-data workflow for graphs where no canonical embedding is known.

Usage

```
compare.layouts(
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  dim = 2,
  candidates = c("default"),
  search = NULL,
  clusters = NULL,
  seeds = 1:3,
  sample.size.stress = 2000L,
  sample.size.nonedge = 5000L,
  edge.crossings = c("auto", "always", "never"),
  edge.crossings.max.edges = 1000L,
```

```
  score.weights = grip.default.compare.score.weights(),
  return.layouts = FALSE,
  disconnected = c("components", "error")
)
```

Arguments

edges	Two-column integer matrix of edges (1-based vertex ids).
n	Number of vertices.
adj_list	Adjacency list (1-based) for undirected graphs.
weight_list	Optional parallel list of positive edge weights.
edge_weights	Optional positive edge-weight vector parallel to edges.
dim	Layout dimension (2 or 3).
candidates	Either a character vector such as c("default", "mesh", "tree") or a named list of candidate layout specifications. Each list element may be NULL (use defaults), a single preset name, or a named list of grip() tuning arguments such as preset, placement, rounds, or repulsion_factor.
search	Optional named list describing a grid search over layout settings. Any of preset, placement, rounds, final_rounds, num_init, num_nbrs, r, s, repulsion_factor, and tinit_factor may be supplied as vectors. All combinations are expanded into candidates. Special fields candidate.prefix and include.base control candidate naming and whether the all-first-values setting is guaranteed to appear.
clusters	Optional cluster or community labels used to compute cluster.separation.
seeds	Integer seeds used for repeated runs.
sample.size.stress	Number of sampled pairs used for sampled.stress.
sample.size.nonedge	Number of sampled non-edge pairs used for sampled.nonedge.sep.ratio.
edge.crossings	How to compute edge.crossings for 2D layouts.
edge.crossings.max.edges	Edge-count threshold for edge.crossings = "auto".
score.weights	Optional named numeric vector used to compute score.composite. Set to NULL to omit the composite score.
return.layouts	If TRUE, include the realized coordinate matrices in the return value.
disconnected	Passed through to grip().

Value

A list with runs and summary data frames and, optionally, realized layouts.

Examples

```
edges <- edges.path(5)
cmp <- compare.layouts(
  edges = edges,
  n = 5,
  dim = 2,
  candidates = c("default", "tree"),
  seeds = 1L
)
cmp$summary[, c("candidate", "rounds", "final.rounds")]

search.cmp <- compare.layouts(
  edges = edges,
  n = 5,
  dim = 2,
  search = list(
    candidate.prefix = "path.search",
    rounds = c(4L, 6L),
    final_rounds = c(4L, 6L)
  ),
  seeds = 1L
)
search.cmp$summary[, c("candidate", "rounds", "final.rounds")]
```

---

cube_mask_pattern_helpers

Cube mask pattern helpers for recursive porous families

Description

Convenience constructors for cubic keep-arrays that can be passed to `edges.recursive.cube.mask()` or `recursive.cube.mask.surface.graph()` to build cubical porous benchmark families. These helpers encode three qualitatively different void patterns: repeated tunnel lattices, interior asymmetric cavities, and deterministic channel networks.

Usage

```
mask.cube.periodic.tunnels(
  side = 5,
  tunnel_width = 1,
  tunnel_period = 2,
  tunnel_offset = 2
)

mask.cube.asymmetric.cavities(
  side = 5,
  cavity_size = 2,
  pocket_size = max(1L, cavity_size - 1L)
```

```
)

mask.cube.channel.network(side = 5, channel_width = 1, branch_offset = 2)
```

Arguments

side	Side length of the cubic keep-array.
tunnel_width	Width of each removed tunnel band.
tunnel_period	Spacing between successive tunnel bands.
tunnel_offset	Starting index of the first tunnel band.
cavity_size	Side length of the larger interior cavity block.
pocket_size	Side length of the smaller secondary cavity block.
channel_width	Width of each removed channel in the channel-network family.
branch_offset	Interior offset of the extra branch channel in mask.cube.channel.network().

Details

The returned arrays use the same orientation as the recursive cube-mask helpers: the first dimension runs from top to bottom, the second from left to right, and the third from front to back.

Value

A logical cubic keep-array.

---

cylinder_surface_helpers

Weighted cylinder surface helpers

Description

Convenience helpers that embed a cylindrical grid graph into $\mathbb{R}^3$ and use the induced Euclidean edge lengths as positive graph weights. These helpers are intended for benchmark families where the graph topology is cylindrical but the intended metric comes from a curved or spatially varying 3D realization.

Usage

```
cylinder.surface.embedding(
  h,
  w = h,
  surface = c("barrel", "hourglass", "wavy"),
  radius = 1,
  height = 2,
  amplitude = 0.3,
  freq_theta = 2,
```

```
  freq_z = 1,
  twist = 0.25
)

cylinder.surface.graph(
  h,
  w = h,
  surface = c("barrel", "hourglass", "wavy"),
  radius = 1,
  height = 2,
  amplitude = 0.3,
  freq_theta = 2,
  freq_z = 1,
  twist = 0.25,
  normalize = c("median", "mean", "none")
)
```

Arguments

h	Number of rows.
w	Number of columns. Defaults to h.
surface	Cylinder surface family. One of "barrel", "hourglass", or "wavy".
radius	Positive baseline cylinder radius.
height	Positive cylinder height.
amplitude	Finite numeric deformation amplitude. The resulting radius profile must stay positive everywhere.
freq_theta	Positive angular frequency used only when surface = "wavy".
freq_z	Positive vertical frequency used only when surface = "wavy".
twist	Finite angular twist applied linearly with height.
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Details

'cylinder.surface.embedding()' returns the 3D coordinates of the embedded cylindrical grid. 'cylinder.surface.graph()' returns a reusable weighted-graph bundle containing the cylinder edges, induced edge weights, the 3D surface coordinates, and a 2D unwrapped parameterization.

Value

cylinder.surface.embedding() returns an n x 3 numeric matrix with columns x, y, and z.

cylinder.surface.graph() returns a list with components:

- edges: the undirected cylindrical-grid edges,
- n: number of vertices,
- edge_weights: induced positive edge lengths,

- `coords_surface`: the 3D surface embedding,
- `coords_param`: the 2D unwrapped parameter coordinates,
- `weight_scale`: the normalization constant applied to the raw edge lengths,
- `family`: always `"cylinder"`,
- `surface`: the chosen surface name,
- `label`: a human-readable family label.

`deprecated-grip-api` *Deprecated long-form API names*

Description

These long names are deprecated in favor of the shorter canonical API.

Usage

```
grip.layout(...)

grip.layout.globalrep(...)

grip.layout.globalrep.weighted(...)

grip.layout.legacy(...)

grip.layout.trace(...)

grip.layout.trace.legacy(...)

grip.layout.trace.weighted(...)

grip.layout.weighted(...)

grip.layout.weighted.nd(...)

grip.plot(...)

grip.project.3d(...)

grip.metric.mds.layout(...)

grip.optimize.kernel.gram.gkk.layout(...)

grip.score.gmds.layout(...)

grip.gmds.layout.result(...)
```

`grip.edge.length.density.stiffness(...)`

`grip.prepare.landmark.geodesic.kk(...)`

`grip.prepare.geodesic.kk(...)`

`grip.prepare.graph.geodesic.mds(...)`

`grip.prepare.misf.geodesic.kk(...)`

`grip.score.landmark.geodesic.kk(...)`

`grip.score.geodesic.kk(...)`

`grip.score.misf.geodesic.kk(...)`

`grip.optimize.landmark.geodesic.kk(...)`

`grip.optimize.geodesic.kk(...)`

`grip.optimize.misf.geodesic.kk(...)`

`grip.optimize.edge.repulsive.stage(...)`

`grip.optimize.repulsive.stage(...)`

`grip.edge.repulsive.state(...)`

`grip.repulsive.state(...)`

`grip.build.misf.weighted(...)`

`grip.build.misf(...)`

`grip.geometry.diagnostics(...)`

`grip.params.from.summary(...)`

Arguments

`...`	Arguments passed to the replacement function.

Value

The replacement function's return value.

edge.kk *Optimize an edge-KK local repair layout*

Description

'edge.kk()' is the edge-restricted Kamada–Kawai local repair operator for the experimental GMDS layout program. Earlier notes called this operator edge-gKK or edge-isometric GMDS; edge-KK is the preferred name because the objective is restricted to graph edges rather than all graph-geodesic pairs. It minimizes weighted edge-length stress

$$\frac{1}{2}\sum_{(i,j)\in E} k_{ij}\left(\|z_i - z_j\|_2 - sw_{ij}\right)^2$$

using deterministic gradient descent with Armijo backtracking. The default 'density_mix_schedule' runs a continuation from density-weighted stiffnesses toward uniform stiffnesses.

Usage

```
edge.kk(
  coords = NULL,
  prepared = NULL,
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  dim = 2L,
  init = c("metric_mds", "weighted_grip", "random"),
  weighted.grip.args = list(),
  stiffness_method = c("density", "uniform", "distance_power"),
  stiffness_transform = c("identity", "sqrt", "log"),
  density_mix_schedule = c(0, 0.25, 0.5, 0.75, 1),
  bandwidth = NULL,
  density_n = 512L,
  distance_power = 0,
  stiffness_floor = 0,
  stiffness_ceiling = Inf,
  scale_mode = c("profiled", "identity", "fixed_initial", "user"),
  scale = NULL,
  max_iter = 50L,
  initial_step = 1,
  step_shrink = 0.5,
  armijo_factor = 1e-04,
  grad_tol = 1e-08,
  min_step = 1e-08,
  edge_length_epsilon = 1e-08,
  distance_floor = 1e-08,
```

```
  recenter = TRUE,
  return_trace = TRUE,
  diagnostics = TRUE,
  seed = 1L,
  engine = c("cpp", "R")
)
```

Arguments

coords	Optional starting coordinates. If omitted, ‘init‘ is used.
prepared	Optional object returned by [prepare.edge.kk()], [prepare.graph.geodesic.mds()] or [prepare.geodesic.kk()]. Edge-only objects from [prepare.edge.kk()] report only edge diagnostics; all-pairs GMDS path and chord diagnostics are unavailable.
edges	Two-column edge matrix used when ‘prepared‘ is omitted.
n	Number of vertices used when ‘prepared‘ is omitted.
adj_list	Optional adjacency list used when ‘prepared‘ is omitted.
weight_list	Optional edge-weight list parallel to ‘adj_list‘.
edge_weights	Optional positive edge weights parallel to ‘edges‘.
dim	Target embedding dimension.
init	Starting layout used when ‘coords‘ is omitted. ‘"metric_mds"‘ uses ordinary metric MDS from an all-pairs prepared object, ‘"weighted_grip"‘ runs [grip()] with ‘metric = "edge_length"‘ on the graph edges first, and ‘"random"‘ uses centered Gaussian coordinates.
weighted.grip.args	Named list of additional tuning arguments passed to [grip()] when ‘init = "weighted_grip"‘. Graph inputs, ‘dim‘, ‘seed‘, and ‘metric‘ are supplied by ‘edge.kk()‘ and may not be repeated here.
stiffness_method, stiffness_transform, density_mix_schedule, bandwidth, density_n	Parameters passed to [edge.length.density.stiffness()].
distance_power, stiffness_floor, stiffness_ceiling	Additional stiffness constructor parameters.
scale_mode	Scale policy for edge targets. ‘"profiled"‘ analytically refits ‘s‘ at every state evaluation, ‘"identity"‘ fixes ‘s = 1‘, ‘"fixed_initial"‘ fits ‘s‘ once at the first continuation stage, and ‘"user"‘ uses ‘scale‘.
scale	User scale for ‘scale_mode = "user"‘.
max_iter	Maximum iterations per continuation stage.
initial_step, step_shrink, armijo_factor, grad_tol, min_step	Line-search controls.
edge_length_epsilon	Small stabilizer for fixed-path embedded lengths.
distance_floor	Positive floor for relative residuals.
recenter	If ‘TRUE‘, recenter the layout after accepted steps.

return_trace	If ‘TRUE‘, keep per-iteration trace rows and coordinate frames. If ‘FALSE‘, omit those payloads while retaining compact stage summaries in ‘metadata$stage_summaries‘.
diagnostics	If ‘TRUE‘, attach the common GMDS diagnostic panel.
seed	Random seed used for ‘init = "weighted_grip"‘ and ‘init = "random"‘.
engine	Optimizer engine. ‘"cpp"‘ uses the Rcpp backend for the edge-stress loop; ‘"R"‘ uses the reference implementation.

Details

For scalable repair from an existing layout, pass ‘coords‘ or an edge-only object from [prepare.edge.kk()]. When raw graph inputs are supplied, ‘edge.kk()‘ uses edge-only preparation whenever ‘coords‘ are supplied or ‘init != "metric_mds"‘. Use ‘init = "weighted_grip"‘ for a scalable weighted-GRIP warm start followed by edge-KK polish. If ‘coords‘ is omitted and ‘init = "metric_mds"‘, use an all-pairs prepared object from [prepare.graph.geodesic.mds()] or [prepare.geodesic.kk()] instead.

Value

A ‘"grip_gmds_layout"‘ object with method ‘"edge_kk"‘.

---

edge.length.density.stiffness

Construct edge-length stiffnesses for edge-isometric GMDS layouts

---

Description

‘edge.length.density.stiffness()‘ constructs stiffnesses for edge-only geodesic-MDS layouts. It turns positive edge lengths into spring stiffnesses normalized to mean one. The ‘"density"‘ method emphasizes edge lengths near the empirical edge-length density mode, and ‘mix‘ provides a continuation path from that density-weighted signal to uniform stiffness.

Usage

```
edge.length.density.stiffness(
  edge_weights,
  method = c("density", "uniform", "distance_power"),
  mix = 0,
  bandwidth = NULL,
  density_n = 512L,
  transform = c("identity", "sqrt", "log"),
  distance_power = 0,
  stiffness_floor = 0,
  stiffness_ceiling = Inf
)
```

Arguments

edge_weights	Positive numeric edge lengths.
method	Stiffness rule. ‘"density"‘ estimates an empirical density, ‘"uniform"‘ returns equal stiffnesses, and ‘"distance_power"‘ uses ‘(w / median(w))^distance_power‘.
mix	Continuation parameter in ‘[0, 1]‘. ‘0‘ uses the selected method; ‘1‘ returns uniform stiffness.
bandwidth	Optional bandwidth passed to ‘stats::density()‘.
density_n	Number of evaluation points for ‘stats::density()‘.
transform	Optional transformation of the raw density/power signal before mixing and normalization.
distance_power	Exponent for ‘method = "distance_power"‘.
stiffness_floor, stiffness_ceiling	Optional clipping bounds applied before the final mean-one normalization.

Value

A list with ‘stiffness‘, raw signal diagnostics, estimated mode, and clipping/normalization metadata.

---

edge.repulsive.stage *Optimize one edge-isometric repulsive unfolding stage*

Description

‘edge.repulsive.stage()‘ performs one gradient-descent stage for the objective evaluated by ‘edge.repulsive.state()‘. It uses Armijo backtracking, optionally recenters coordinates after each proposal, and returns the final coordinates plus a per-iteration trace.

Usage

```
edge.repulsive.stage(
  coords,
  edges,
  edge.lengths,
  edge.weights = NULL,
  lambda = 0,
  edge.family = c("quadratic", "upper_barrier"),
  eps.plus = 0.35,
  beta = 0,
  pair.index = NULL,
  pair.weights = NULL,
  repulsion.family = c("log", "inverse_power"),
  repulsion.delta = 0.001,
  repulsion.power = 1,
  max.iter = 80L,
```

```
  initial.step = 0.02,
  step.shrink = 0.5,
  armijo = 1e-04,
  min.step = 1e-08,
  grad.tol = 1e-07,
  recenter = TRUE,
  distance.eps = 1e-10,
  return.frames = FALSE,
  engine = c("cpp", "R")
)
```

Arguments

coords	Numeric ‘n‘ by ‘dim‘ coordinate matrix.
edges	Integer or numeric matrix with two columns containing 1-based graph edge endpoints.
edge.lengths	Numeric vector of target edge lengths, parallel to ‘edges‘.
edge.weights	Optional non-negative edge weights. Defaults to one.
lambda	Repulsion strength.
edge.family	Edge potential family, either ‘"quadratic"‘ or ‘"upper_barrier"‘.
eps.plus	Upper-barrier slack parameter.
beta	Upper-barrier strength. When ‘beta <= 0‘, the edge potential is quadratic.
pair.index	Optional two-column matrix of 1-based vertex pairs for the repulsion term. If ‘NULL‘ and ‘lambda > 0‘, all unordered pairs are used.
pair.weights	Optional pair weights, parallel to ‘pair.index‘.
repulsion.family	Repulsion potential family, either ‘"log"‘ or ‘"inverse_power"‘.
repulsion.delta	Small positive softening parameter for pair distances.
repulsion.power	Power used by the ‘"inverse_power"‘ repulsion.
max.iter	Maximum number of gradient-descent iterations.
initial.step	Initial gradient step size.
step.shrink	Multiplicative shrink factor used by backtracking.
armijo	Armijo sufficient-decrease coefficient.
min.step	Minimum allowable step before the stage stops.
grad.tol	Gradient-norm stopping tolerance.
recenter	Logical; whether to recenter coordinates after each proposal.
distance.eps	Small positive distance floor used in derivatives.
return.frames	Logical; whether to return accepted coordinate frames. Frame 0 is the starting coordinate matrix and later frames are accepted updates.
engine	Backend engine. ‘"cpp"‘ is the default; ‘"R"‘ uses the reference implementation.

Value

A list with ‘coords‘, final ‘state‘, a data-frame ‘trace‘, and, when requested, a list of coordinate ‘frames‘.

`edge.repulsive.state` *Evaluate an edge-isometric repulsive unfolding objective*

Description

‘edge.repulsive.state()‘ evaluates an experimental GMDS objective that combines an edge-isometric term with a configurable repulsion term. The edge term penalizes deviation of embedded edge lengths from graph edge lengths. The repulsion term can be evaluated on all vertex pairs or on a selected pair set.

Usage

```
edge.repulsive.state(
  coords,
  edges,
  edge.lengths,
  edge.weights = NULL,
  edge.family = c("quadratic", "upper_barrier"),
  eps.plus = 0.35,
  beta = 0,
  lambda = 0,
  pair.index = NULL,
  pair.weights = NULL,
  repulsion.family = c("log", "inverse_power"),
  repulsion.delta = 0.001,
  repulsion.power = 1,
  distance.eps = 1e-10,
  engine = c("cpp", "R")
)
```

Arguments

`coords`	Numeric ‘n‘ by ‘dim‘ coordinate matrix.
`edges`	Integer or numeric matrix with two columns containing 1-based graph edge endpoints.
`edge.lengths`	Numeric vector of target edge lengths, parallel to ‘edges‘.
`edge.weights`	Optional non-negative edge weights. Defaults to one.
`edge.family`	Edge potential family, either ‘"quadratic"‘ or ‘"upper_barrier"‘.
`eps.plus`	Upper-barrier slack parameter.
`beta`	Upper-barrier strength. When ‘beta <= 0‘, the edge potential is quadratic.

`lambda`	Repulsion strength.
`pair.index`	Optional two-column matrix of 1-based vertex pairs for the repulsion term. If ‘NULL‘ and ‘lambda > 0‘, all unordered pairs are used.
`pair.weights`	Optional pair weights, parallel to ‘pair.index‘.
`repulsion.family`	Repulsion potential family, either ‘"log"‘ or ‘"inverse_power"‘.
`repulsion.delta`	Small positive softening parameter for pair distances.
`repulsion.power`	Power used by the ‘"inverse_power"‘ repulsion.
`distance.eps`	Small positive distance floor used in derivatives.
`engine`	Backend engine. ‘"cpp"‘ is the default; ‘"R"‘ uses the reference implementation.

Details

The default ‘engine = "cpp"‘ uses the compiled backend. ‘engine = "R"‘ keeps a reference implementation available for diagnostics and regression tests.

Value

A list containing total energy, edge and repulsion energies, gradient, gradient norm, feasibility flag, embedded edge lengths, relative edge lengths, edge residuals, and number of upper-barrier wall violations.

`geodesic.kk` *Optimize a layout under the full geodesic KK energy*

Description

`geodesic.kk()` applies a deterministic gradient-descent polish under the full all-pairs geodesic Kamada–Kawai objective.

Usage

```
geodesic.kk(
  coords,
  prepared = NULL,
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  max_iter = 16L,
  stiffness = 1,
  distance_floor = 1e-08,
```

```
  edge_length_epsilon = 1e-08,
  initial_step = 1,
  step_shrink = 0.5,
  armijo_factor = 1e-04,
  grad_tol = 1e-08,
  min_step = 1e-08,
  recenter = TRUE,
  return_trace = FALSE,
  scale_mode = c("fixed_initial", "profiled", "user"),
  scale.L0 = NULL
)
```

Arguments

coords	Numeric coordinate matrix with 2 or 3 columns.
prepared	Optional object returned by prepare.geodesic.kk().
edges	Two-column integer matrix of edges (1-based vertex ids).
n	Number of vertices.
adj_list	Adjacency list (1-based) for an undirected graph.
weight_list	Optional parallel list of positive edge weights.
edge_weights	Optional positive edge-weight vector parallel to edges.
max_iter	Maximum number of gradient-descent iterations.
stiffness	Global stiffness constant \(K\).
distance_floor	Small positive floor used in k_ij = K / max(g_ij, distance_floor)^2.
edge_length_epsilon	Small positive stabilizer added inside each embedded edge length.
initial_step	Initial line-search step size.
step_shrink	Multiplicative shrink factor in ‘(0, 1)‘ for backtracking.
armijo_factor	Non-negative Armijo decrease constant.
grad_tol	Non-negative stopping tolerance on the gradient norm.
min_step	Positive minimum accepted line-search step before giving up.
recenter	If TRUE, recenter the layout to zero mean after each accepted step.
return_trace	If TRUE, include per-iteration diagnostics and the accepted intermediate coordinate frames.
scale_mode	One of "fixed_initial", "profiled", or "user".
scale.L0	Optional user-supplied fixed scale, required when scale_mode = "user".

Details

The optimizer supports three scale policies. With scale_mode = "fixed_initial" the target scale is fit once from the starting layout and then held fixed during optimization, matching the landmark geodesic KK behavior. With scale_mode = "profiled" the scale is re-fit analytically at each evaluation. With scale_mode = "user", a fixed user-supplied scale.L0 is used throughout.

Value

A list with coords, trace, frames, prepared, and score.

---

geometry.diagnostics *Geometry-aware diagnostics against a canonical target*

Description

geometry.diagnostics() augments the graph-aware quality measures in score.layout() with target-aware geometric diagnostics. The function aligns coords to target.coords using an orthogonal Procrustes fit, then reports global symmetry, local angle preservation, edge-axis concentration, and, for Sierpinski carpet layouts, boundary, corridor, and hole-center diagnostics.

Usage

```
geometry.diagnostics(
  coords,
  target.coords,
  edges,
  family = NULL,
  sample.size.symmetry = 512L,
  sample.size.wedges = 4000L,
  rng.seed = 1L
)
```

Arguments

coords	Numeric layout matrix with 2 or 3 columns.
target.coords	Canonical target coordinates with the same shape as coords.
edges	Two-column integer edge matrix.
family	Optional graph-family label. Use "sierpinski.carpet" to enable carpet-specific diagnostics.
sample.size.symmetry	Number of vertices sampled when evaluating global symmetry.
sample.size.wedges	Number of wedges sampled when evaluating local angle deviation.
rng.seed	Integer seed used for the symmetry and wedge sampling.

Details

Metrics are reported so that larger global.symmetry.score and edge.axis.concentration are better, while smaller procrustes.rmse, local.angle.deviation, boundary.waviness, corridor.waviness, hole.center.error, central.hole.skew, central.hole.aspect.error, and central.hole.center.error are better.

Value

A one-row data frame of geometric diagnostics.

`globalrep.grip` *Compute a GRIP layout with coarse global repulsion*

Description

This compatibility entry point is equivalent to `grip(..., metric = "hop")`. It keeps the old global-repulsion name available while using the same quality-first multiscale GRIP engine and adaptive `final_rounds` schedule as the primary layout API.

Usage

```
globalrep.grip(
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  dim = 3,
  placement = c("barycenter", "circle"),
  preset = NULL,
  rounds = 160,
  final_rounds = 384,
  num_init = 24,
  num_nbrs = 20,
  r = 0.03,
  s = 7.5,
  repulsion_factor = 2.5,
  coarse_repulsion_factor = 1.5,
  coarse_repulsion_sample = 16,
  coarse_repulsion_exact_below = 64,
  final_anchor_factor = 0,
  final_move_scale_after_first = 1,
  final_mode = c("fr", "kk_repulse"),
  insertion_anchor_count = 3,
  insertion_anchor_scope = c("any_higher", "prev_misf"),
 insertion_anchor_strategy = c("first", "distance_band", "balanced_band", "spread_prev"),
  level0_insertion_mode = c("inherit", "barycenter", "least_squares"),
  level0_anchor_count = insertion_anchor_count,
  level0_local_kk_steps = 3,
  lgkk_polish_rounds = 0L,
  lgkk_multiscale_rounds = 0L,
  lgkk_rounds_coarse = NULL,
  lgkk_rounds_pre_final = NULL,
```

```
  lgkk_rounds_final = NULL,
  lgkk_local_nbrs = 20L,
  lgkk_landmark_count = 8L,
  lgkk_multiscale_scope = c("all", "coarse"),
  lgkk_active_limit = 4096L,
  tinit_factor = 6,
  seed = 6,
  disconnected = c("components", "error")
)
```

Arguments

`edges`	Two-column integer matrix of edges (1-based vertex ids).
`n`	Number of vertices.
`adj_list`	Adjacency list (1-based) for undirected graphs.
`weight_list`	Optional parallel list of positive edge lengths for `adj_list`. `NULL` treats every edge as length 1.
`edge_weights`	Optional positive edge lengths for `edges`, in row order. `NULL` treats every edge as length 1.
`dim`	Layout dimension (2 or 3). Default is 3.
`placement`	Initial placement strategy. "circle" is only used for 2D.
`preset`	Optional tuning preset. `NULL` uses the quality-first defaults. `"carpet"` applies a preset tuned for Sierpinski-carpet-like graphs and validated on carpet levels 3 and 4. `"mesh"` applies a preset tuned for rectangular lattice graphs and validated on 8x8 and 12x12 mesh layouts. `"torus"` applies a preset tuned for 3D torus layouts and validated on torus sizes from 8x8 through 20x20. `"tree"` applies a preset tuned for symmetric force-directed layouts of tree-like graphs and validated on binary trees of depths 5 and 6. Presets only fill in tuning arguments that you did not supply explicitly.
`rounds`	Initial rounds for refinement.
`final_rounds`	Final rounds for refinement.
`num_init`	Number of initial vertices in the coarsest level.
`num_nbrs`	Maximum number of graph-distance neighbors retained for local refinement at each filtration level.
`r`	Main local temperature adaptation rate in `[0, 1]`.
`s`	Non-negative boost factor applied when successive displacements have a consistent direction.
`repulsion_factor`	Non-negative multiplier applied to GRIP's finest-level repulsive force scale.
`coarse_repulsion_factor`	Non-negative multiplier applied to the extra coarse-level active-set repulsion term. `0` disables that extra term; when the remaining tuning arguments also match `legacy.grip()`, the result matches the legacy layout behavior.

`coarse_repulsion_sample`
: Positive integer sample size used to approximate active-set-wide repulsion on larger coarse levels.

`coarse_repulsion_exact_below`
: Positive integer threshold. When the active set size is at most this value, the coarse repulsion is computed exactly against all currently active vertices instead of being sampled.

`final_anchor_factor`
: Non-negative multiplier for an anchor term that pulls the final FR stage back toward the pre-final full-graph layout. ‘0‘ disables the anchor and preserves the current behavior.

`final_move_scale_after_first`
: Scalar in ‘[0, 1]‘ applied to the final FR displacement after the first finest-level round. Values below ‘1‘ damp later full-graph movement while keeping the first FR round unchanged.

`final_mode`
: Final full-graph refinement mode. `"fr"` keeps the current Fruchterman-Reingold-style final stage. `"kk_repulse"` uses a KK-style local distance-matching update with explicit active-set repulsion instead of the final FR phase.

`insertion_anchor_count`
: Positive integer number of anchor vertices used during multiscale insertion on non-initial MISF refinement levels. This is the closest current implementation to a global `K_mish` parameter.

`insertion_anchor_scope`
: Anchor-eligibility rule used during multiscale insertion. `"any_higher"` matches the historical GRIP behavior and allows anchors from any already placed higher MISF level. `"prev_misf"` restricts anchors to the immediately previous MISF level only.

`insertion_anchor_strategy`
: Anchor-selection rule used during multiscale insertion. `"first"` keeps the historical first-anchors-found BFS behavior. `"distance_band"` keeps exploring until the `K_mish`-th anchor distance band is exhausted, then places the new vertex from that less order-sensitive anchor pool. `"balanced_band"` uses the same band expansion, then explicitly selects a subset whose centroid stays centered in the candidate cloud while remaining geometrically spread out. `"spread_prev"` is a symmetry-oriented band strategy intended to be paired with `insertion_anchor_scope = "prev_misf"`; it selects anchors with broad angular and geometric coverage before placement.

`level0_insertion_mode`
: Level-0 insertion placement override used only when the finest filtration level is first populated. `"inherit"` keeps the current GRIP behavior. `"barycenter"` disables the 2D circle heuristic at level 0 and uses barycentric anchor placement. `"least_squares"` uses a multi-anchor least-squares distance fit at level 0 before any local micro-polish.

`level0_anchor_count`
: Positive integer number of already placed anchors to collect for level-0 insertion experiments. By default this inherits `insertion_anchor_count`. The legacy behavior uses 3.

`level0_local_kk_steps`
: Non-negative integer number of tiny local KK micro-polish steps applied immediately after each level-0 insertion. The legacy behavior uses 3.

`lgkk_polish_rounds`
: Non-negative integer number of experimental landmark-geodesic KK polish iterations applied after the main GRIP solve. `0` disables the polish.

`lgkk_multiscale_rounds`
: Non-negative integer number of compiled landmark-geodesic KK refinement rounds applied inside the multiscale solver after each eligible MISF level completes its standard GRIP rounds. This legacy shared budget is used as a fallback when any of the more specific per-stage budgets below are left `NULL`.

`lgkk_rounds_coarse`
: Optional non-negative integer number of compiled LGKK rounds applied on coarse MISF levels with `misf_level > 1`. When `NULL`, this falls back to `lgkk_multiscale_rounds`.

`lgkk_rounds_pre_final`
: Optional non-negative integer number of compiled LGKK rounds applied on the last coarse level just before the full graph is opened (`misf_level == 1`). When `NULL`, this falls back to `lgkk_multiscale_rounds`.

`lgkk_rounds_final`
: Optional non-negative integer number of compiled LGKK rounds applied after the full graph level completes its standard GRIP rounds (`misf_level == 0`). When `NULL`, this falls back to `lgkk_multiscale_rounds`.

`lgkk_local_nbrs`
: Number of nearest graph-metric neighbors retained per vertex in the LGKK sparse local set when either LGKK stage is enabled.

`lgkk_landmark_count`
: Number of farthest-point landmarks retained per vertex in the LGKK sparse long-range set when either LGKK stage is enabled.

`lgkk_multiscale_scope`
: Scope for the compiled multiscale LGKK stage. `"all"` applies it after every eligible MISF level, including the final full-graph level. `"coarse"` applies it only on coarse levels.

`lgkk_active_limit`
: Positive integer upper bound on the active-set size for compiled multiscale LGKK cache construction. Levels larger than this skip the multiscale LGKK stage.

`tinit_factor`
: Initial temperature factor.

`seed`
: Optional RNG seed for reproducibility. If NULL, uses current time.

`disconnected`
: How to handle disconnected graphs: `"components"` (default) lays out each connected component separately and packs them into one coordinate matrix; `"error"` stops with an error.

Value

A numeric matrix with ‘n‘ rows and ‘dim‘ columns.

Examples

```
edges <- edges.mesh(4, 4)
coords <- globalrep.grip(edges, n = max(edges), dim = 2,
                                rounds = 8, final_rounds = 8,
                                num_init = 6, num_nbrs = 8,
                                coarse_repulsion_factor = 0.2,
                                coarse_repulsion_sample = 8,
                                coarse_repulsion_exact_below = 32,
                                seed = 1)
round(coords, 3)
```

`globalrep.weighted.grip`

Compute a weighted geometry-aware GRIP layout

Description

`globalrep.weighted.grip()` is the compatibility backend equivalent to `grip(..., metric = "edge_length")`. It runs the edge-length multiscale core, which uses edge lengths in the filtration, local neighborhoods, insertion, refinement forces, and optional in-core multiscale LGKK refinement.

Usage

```
globalrep.weighted.grip(
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  dim = 3,
  placement = c("barycenter", "circle"),
  preset = NULL,
  rounds = 160,
  final_rounds = 384,
  num_init = 24,
  num_nbrs = 20,
  r = 0.03,
  s = 7.5,
  repulsion_factor = 2.5,
  coarse_repulsion_factor = 1.5,
  coarse_repulsion_sample = 16,
  coarse_repulsion_exact_below = 64,
  final_anchor_factor = 0,
  final_move_scale_after_first = 1,
  final_mode = c("fr", "kk_repulse"),
  insertion_anchor_count = 3,
  insertion_anchor_scope = c("any_higher", "prev_misf"),
```

```
  insertion_anchor_strategy = c("first", "distance_band", "balanced_band", "spread_prev"),
  level0_insertion_mode = c("inherit", "barycenter", "least_squares"),
  level0_anchor_count = insertion_anchor_count,
  level0_local_kk_steps = 3,
  lgkk_polish_rounds = 0L,
  lgkk_multiscale_rounds = 0L,
  lgkk_rounds_coarse = NULL,
  lgkk_rounds_pre_final = NULL,
  lgkk_rounds_final = NULL,
  lgkk_local_nbrs = 20L,
  lgkk_landmark_count = 8L,
  lgkk_multiscale_scope = c("all", "coarse"),
  lgkk_active_limit = 4096L,
  metric_neighbor_cap = NULL,
  length_normalization = c("median", "mean", "none"),
  tinit_factor = 6,
  seed = 6,
  disconnected = c("components", "error")
)
```

Arguments

edges	Two-column integer matrix of edges (1-based vertex ids).
n	Number of vertices.
adj_list	Adjacency list (1-based) for undirected graphs.
weight_list	Parallel list of positive edge lengths for adj_list; required when adjacency-list input is used.
edge_weights	Positive edge lengths for edges, in row order; required when edge-list input is used.
dim	Layout dimension (2 or 3). Default is 3.
placement	Initial placement strategy. "circle" is only used for 2D.
preset	Optional weighted tuning preset. NULL uses the quality-first defaults for the weighted core. "mesh" targets rectangular and near-mesh weighted surfaces, "cylinder" targets cylindrical grids, "torus" targets wrapped surface grids, "sphere" targets closed near-spherical meshes, "irregular" targets irregular manifold-like weighted families, "tree" targets intrinsic weighted trees, and "carpet" keeps a high-neighborhood profile for carpet-like recursive lattices. Explicit tuning arguments override the preset field by field.
rounds	Initial rounds for refinement.
final_rounds	Final rounds for refinement.
num_init	Number of initial vertices in the coarsest level.
num_nbrs	Maximum number of graph-distance neighbors retained for local refinement at each filtration level.
r	Main local temperature adaptation rate in [0, 1].

`s` Non-negative boost factor applied when successive displacements have a consistent direction.

`repulsion_factor`
Non-negative multiplier applied to GRIP's finest-level repulsive force scale.

`coarse_repulsion_factor`
Non-negative multiplier applied to the extra coarse-level active-set repulsion term. `0` disables that extra term; when the remaining tuning arguments also match `legacy.grip()`, the result matches the legacy layout behavior.

`coarse_repulsion_sample`
Positive integer sample size used to approximate active-set-wide repulsion on larger coarse levels.

`coarse_repulsion_exact_below`
Positive integer threshold. When the active set size is at most this value, the coarse repulsion is computed exactly against all currently active vertices instead of being sampled.

`final_anchor_factor`
Non-negative multiplier for an anchor term that pulls the final FR stage back toward the pre-final full-graph layout. '0' disables the anchor and preserves the current behavior.

`final_move_scale_after_first`
Scalar in '[0, 1]' applied to the final FR displacement after the first finest-level round. Values below '1' damp later full-graph movement while keeping the first FR round unchanged.

`final_mode` Final full-graph refinement mode. `"fr"` keeps the current Fruchterman-Reingold-style final stage. `"kk_repulse"` uses a KK-style local distance-matching update with explicit active-set repulsion instead of the final FR phase.

`insertion_anchor_count`
Positive integer number of anchor vertices used during multiscale insertion on non-initial MISF refinement levels. This is the closest current implementation to a global `K_mish` parameter.

`insertion_anchor_scope`
Anchor-eligibility rule used during multiscale insertion. `"any_higher"` matches the historical GRIP behavior and allows anchors from any already placed higher MISF level. `"prev_misf"` restricts anchors to the immediately previous MISF level only.

`insertion_anchor_strategy`
Anchor-selection rule used during multiscale insertion. `"first"` keeps the historical first-anchors-found BFS behavior. `"distance_band"` keeps exploring until the `K_mish`-th anchor distance band is exhausted, then places the new vertex from that less order-sensitive anchor pool. `"balanced_band"` uses the same band expansion, then explicitly selects a subset whose centroid stays centered in the candidate cloud while remaining geometrically spread out. `"spread_prev"` is a symmetry-oriented band strategy intended to be paired with `insertion_anchor_scope = "prev_misf"`; it selects anchors with broad angular and geometric coverage before placement.

`level0_insertion_mode`
Level-0 insertion placement override used only when the finest filtration level is first populated. `"inherit"` keeps the current GRIP behavior. `"barycenter"`

disables the 2D circle heuristic at level 0 and uses barycentric anchor placement. `"least_squares"` uses a multi-anchor least-squares distance fit at level 0 before any local micro-polish.

`level0_anchor_count`
: Positive integer number of already placed anchors to collect for level-0 insertion experiments. By default this inherits `insertion_anchor_count`. The legacy behavior uses 3.

`level0_local_kk_steps`
: Non-negative integer number of tiny local KK micro-polish steps applied immediately after each level-0 insertion. The legacy behavior uses 3.

`lgkk_polish_rounds`
: Non-negative integer number of experimental landmark-geodesic KK polish iterations applied after the main GRIP solve. `0` disables the polish.

`lgkk_multiscale_rounds`
: Non-negative integer number of compiled landmark-geodesic KK refinement rounds applied inside the multiscale solver after each eligible MISF level completes its standard GRIP rounds. This legacy shared budget is used as a fallback when any of the more specific per-stage budgets below are left `NULL`.

`lgkk_rounds_coarse`
: Optional non-negative integer number of compiled LGKK rounds applied on coarse MISF levels with `misf_level > 1`. When `NULL`, this falls back to `lgkk_multiscale_rounds`.

`lgkk_rounds_pre_final`
: Optional non-negative integer number of compiled LGKK rounds applied on the last coarse level just before the full graph is opened (`misf_level == 1`). When `NULL`, this falls back to `lgkk_multiscale_rounds`.

`lgkk_rounds_final`
: Optional non-negative integer number of compiled LGKK rounds applied after the full graph level completes its standard GRIP rounds (`misf_level == 0`). When `NULL`, this falls back to `lgkk_multiscale_rounds`.

`lgkk_local_nbrs`
: Number of nearest graph-metric neighbors retained per vertex in the LGKK sparse local set when either LGKK stage is enabled.

`lgkk_landmark_count`
: Number of farthest-point landmarks retained per vertex in the LGKK sparse long-range set when either LGKK stage is enabled.

`lgkk_multiscale_scope`
: Scope for the compiled multiscale LGKK stage. `"all"` applies it after every eligible MISF level, including the final full-graph level. `"coarse"` applies it only on coarse levels.

`lgkk_active_limit`
: Positive integer upper bound on the active-set size for compiled multiscale LGKK cache construction. Levels larger than this skip the multiscale LGKK stage.

`metric_neighbor_cap`
: Optional cap on the number of settled Dijkstra vertices used when building weighted neighborhood caches for inserted vertices. `NULL` (default) keeps the exact weighted neighborhood search, but now stops as soon as the required

	weighted neighbors and anchors are filled. Supplying a positive integer enables an approximate weighted neighborhood mode for larger graphs.
length_normalization	Global edge-length normalization: "median" (default), "mean", or "none".
tinit_factor	Initial temperature factor.
seed	Optional RNG seed for reproducibility. If NULL, uses current time.
disconnected	How to handle disconnected graphs: "components" (default) lays out each connected component separately and packs them into one coordinate matrix; "error" stops with an error.

Value

A numeric matrix with n rows and dim columns.

---

gmds.result *Construct a common GMDS layout result*

Description

‘gmds.result()‘ wraps coordinates, method metadata, optional traces, and optional diagnostics in a common result shape used by the experimental GMDS layout program.

Usage

```
gmds.result(
  coords,
  method,
  prepared = NULL,
  trace = NULL,
  diagnostics = NULL,
  metadata = list()
)
```

Arguments

coords	Numeric coordinate matrix.
method	Short method identifier.
prepared	Optional prepared graph object.
trace	Optional trace table or list.
diagnostics	Optional one-row diagnostic data frame.
metadata	Optional named list with method-specific metadata.

Value

A list of class ‘"grip_gmds_layout"‘ with fields ‘coords‘, ‘method‘, ‘prepared‘, ‘trace‘, ‘diagnostics‘, and ‘metadata‘.

`graph.riemannian.star.structure` *Build a local Riemannian star structure for Gram-gKK layouts*

Description

'graph.riemannian.star.structure()' builds the local star-pair table used by kernel Gram-gKK. For each center vertex 'u', it considers unordered neighbor pairs '(v, v')' in the graph star and stores the target edge lengths, target cosine angle, and kernel weight

$$r_u(v, v') \left(\frac{1 - \cos \alpha_u(v, v')}{2} \right)^q .$$

In the first implementation, target angles are estimated from the ambient coordinates 'X'. The optimizer is intentionally agnostic to where those cosines came from, so later graph constructors can attach a different Riemannian source without changing the layout backend.

Usage

```
graph.riemannian.star.structure(
  graph = NULL,
  X,
  prepared = NULL,
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  angle.power = 4,
  reliability = c("length.balance", "none"),
  min.angle.weight = 0,
  star.quantile = 0
)
```

Arguments

`graph`	Optional graph/prepared object containing 'adj_list' and 'weight_list', or 'edges'/'edge_targets' when it is a prepared GMDS graph.
`X`	Numeric matrix used to estimate local target angles.
`prepared`	Optional prepared GMDS object. Used when 'graph' is omitted.
`edges`, `n`, `edge_weights`	Optional edge representation used when 'graph' and 'prepared' are omitted.
`adj_list`, `weight_list`	Optional adjacency-list representation.
`angle.power`	Non-negative exponent 'q' in the antipodal kernel.

`reliability` Reliability weighting rule. ‘"length.balance"‘ multiplies by ‘min(w_1, w_2) / max(w_1, w_2)‘; ‘"none"‘ uses one.

`min.angle.weight` Pairs with final angle weight at or below this value are omitted from the returned table.

`star.quantile` Optional quantile in ‘[0, 1)‘. When positive, retain only star pairs whose angle weight is at least this empirical quantile after applying ‘min.angle.weight‘.

Value

A list of class ‘"grip_riemannian_star"‘ with the star-pair table and construction metadata.

---

graph_generators *Sample graph generators*

Description

Convenience helpers that build small undirected graph families as two-column integer edge matrices suitable for `grip()`. These helpers are meant for examples, experiments, and reproducible tests.

Usage

```
edges.path(n)

edges.cycle(n)

edges.mesh(h, w = h, connectivity = c("orthogonal", "diagonal"))

edges.occupied.mesh(keep, connectivity = c("orthogonal", "diagonal"))

edges.cylinder(h, w = h)

edges.torus(h, w = h)

edges.irregular.ball(
  base = c("tetrahedron", "octahedron", "icosahedron"),
  level = 1,
  layers = 3,
  outer_radius = 1,
  radial_irregularity = 0.25,
  layer_twist = 0.35
)

edges.irregular.shell(
  base = c("tetrahedron", "octahedron", "icosahedron"),
  level = 1,
  layers = 3,
```

```
  inner_radius = 0.45,
  outer_radius = 1,
  radial_irregularity = 0.25,
  layer_twist = 0.35
)

edges.irregular.torus(
  major_rings = 8,
  tube_count = 16,
  count_irregularity = 0.2,
  major_irregularity = 0.25,
  phase_twist = 0.35
)

edges.sphere(h, w = h)

edges.irregular.annulus(
  rings = 6,
  outer_count = 28,
  outer_radius = 1,
  inner_radius = 0.45,
  count_irregularity = 0.2,
  radial_irregularity = 0.35,
  phase_twist = 0.35
)

edges.irregular.pair.of.pants(
  slices = 11,
  outer_count = 28,
  outer_radius = 1.1,
  hole_radius = 0.24,
  hole_offset = 0.38,
  hole_height = 0.18,
  count_irregularity = 0.2,
  vertical_irregularity = 0.35,
  phase_twist = 0.35
)

edges.irregular.double.torus(
  slices = 11,
  tube_count = 14,
  branch_length = 0.85,
  branch_offset = 0.72,
  tube_radius = 0.28,
  transition_width = 0.42,
  count_irregularity = 0.2,
  axial_irregularity = 0.3,
  phase_twist = 0.35
```

```
)

edges.irregular.sphere(
  bands = 6,
  equator_count = 28,
  count_irregularity = 0.2,
  lat_irregularity = 0.35,
  phase_twist = 0.35
)

edges.cube(side = 2)

edges.kary.tree(k = 2, depth = 2)

edges.recursive.mask.grid(mask, level = 2)

edges.recursive.triangle.mask(mask = mask.triangle.classic(), level = 2)

edges.recursive.tetrahedron.mask(mask = mask.tetrahedron.classic(), level = 2)

edges.recursive.cube.mask(mask, level = 2)

edges.triangulated.polyhedron(
  base = c("tetrahedron", "octahedron", "icosahedron"),
  level = 1
)

edges.triangulated.annulus(
  resolution = 12,
  outer_radius = 1,
  inner_radius = 0.45
)

edges.triangulated.pair.of.pants(
  resolution = 12,
  outer_radius = 1.1,
  hole_radius = 0.24,
  hole_offset = 0.38,
  hole_height = 0.18
)

edges.vicsek(level = 2)

edges.menger.sponge(level = 2)

edges.cube.periodic.tunnels(
  level = 2,
  side = 5,
```

```
  tunnel_width = 1,
  tunnel_period = 2,
  tunnel_offset = 2
)

edges.cube.asymmetric.cavities(
  level = 2,
  side = 5,
  cavity_size = 2,
  pocket_size = max(1L, cavity_size - 1L)
)

edges.cube.channel.network(
  level = 2,
  side = 5,
  channel_width = 1,
  branch_offset = 2
)

edges.sierpinski.triangle(level = 2)

edges.sierpinski.tetrahedron(level = 2)

edges.sierpinski.carpet(level = 2)
```

Arguments

n	Number of vertices.
h	Number of rows.
w	Number of columns. Defaults to h.
connectivity	Mesh neighborhood rule. "orthogonal" keeps the 4-neighbor grid; "diagonal" also adds both diagonals of every unit square.
keep	Logical or numeric occupancy matrix. Non-zero entries are kept.
base	Base polyhedron for edges.triangulated.polyhedron(). One of "tetrahedron", "octahedron", or "icosahedron".
level	Recursion depth. For edges.recursive.mask.grid(), edges.recursive.cube.mask(), edges.vicsek(), edges.menger.sponge(), and edges.sierpinski.carpet(), level must be at least 1; edges.recursive.triangle.mask(), edges.recursive.tetrahedron.mask and edges.triangulated.polyhedron() also allow level = 0.
layers	Number of non-center radial layers for edges.irregular.ball() and number of inner-to-outer layers for edges.irregular.shell().
outer_radius	Positive outer boundary radius for edges.triangulated.annulus() and edges.triangulated.pair.o
radial_irregularity	Within-ring radial irregularity level for edges.irregular.annulus().
layer_twist	Finite z-axis twist applied across radial layers in edges.irregular.ball() and edges.irregular.shell().

`inner_radius` Positive inner annulus radius for `edges.triangulated.annulus()`.

`major_rings` Number of cyclic major rings for `edges.irregular.torus()`.

`tube_count` Approximate number of vertices around each minor cycle for `edges.irregular.torus()` and around each tube-like loop for `edges.irregular.double.torus()`.

`count_irregularity` Irregularity level for sample counts in `edges.irregular.torus()`, `edges.irregular.annulus()`, `edges.irregular.pair.of.pants()`, `edges.irregular.double.torus()`, and `edges.irregular.sphere()`.

`major_irregularity` Major-angle ring-spacing irregularity level for `edges.irregular.torus()`.

`phase_twist` Angular phase offset used to desynchronize neighboring rings, slice samples, or latitude bands in the irregular torus, irregular annulus, irregular pair-of-pants, irregular double torus, and irregular sphere families.

`rings` Number of concentric sample rings for `edges.irregular.annulus()`.

`outer_count` Approximate number of vertices on the outer boundary for `edges.irregular.annulus()` and across the widest slices for `edges.irregular.pair.of.pants()`.

`slices` Number of horizontal sample slices for `edges.irregular.pair.of.pants()`.

`hole_radius` Positive radius of each interior hole for `edges.triangulated.pair.of.pants()`.

`hole_offset` Positive horizontal offset of the two hole centers for `edges.triangulated.pair.of.pants()`.

`hole_height` Shared vertical coordinate of the two hole centers for `edges.triangulated.pair.of.pants()`.

`vertical_irregularity` Slice-spacing irregularity level for `edges.irregular.pair.of.pants()`.

`branch_length` Half-length of the three-loop central region for `edges.irregular.double.torus()`.

`branch_offset` Offset of the outer loop centers from the middle loop for `edges.irregular.double.torus()`.

`tube_radius` Baseline radius of each tube-like loop for `edges.irregular.double.torus()`.

`transition_width` Width of the single-loop to three-loop transition regions for `edges.irregular.double.torus()`.

`axial_irregularity` Slice-spacing irregularity level for `edges.irregular.double.torus()`.

`bands` Number of non-pole latitude bands for `edges.irregular.sphere()`.

`equator_count` Approximate number of vertices near the equator for `edges.irregular.sphere()`.

`lat_irregularity` Latitude-band spacing irregularity level for `edges.irregular.sphere()`.

`side` Number of lattice points along each cube edge.

`k` Branching factor.

`depth` Number of levels below the root.

`mask` Keep-mask describing which recursive cells are retained. For `edges.recursive.mask.grid()`, `mask` must be a square logical or numeric keep-matrix whose non-zero entries are kept at each recursive subdivision step. For `edges.recursive.triangle.mask()`, `mask` must instead be a four-entry vector in `left`, `right`, `top`, `center` order, and for `edges.recursive.tetrahedron.mask()`, `mask` must be a four-entry vector in `base_left`, `base_right`, `base_back`, `apex` order. For `edges.recursive.cube.mask()`, `mask` must be a cubic logical or numeric keep-array whose non-zero entries are kept at each recursive subdivision step.

resolution	Positive lattice-resolution control used by edges.triangulated.annulus() and edges.triangulated.pair.of.pants().
tunnel_width	Width of each removed tunnel band in edges.cube.periodic.tunnels().
tunnel_period	Spacing between successive tunnel bands in edges.cube.periodic.tunnels().
tunnel_offset	Starting index of the first tunnel band in edges.cube.periodic.tunnels().
cavity_size	Side length of the larger interior cavity block in edges.cube.asymmetric.cavities().
pocket_size	Side length of the smaller secondary cavity block in edges.cube.asymmetric.cavities().
channel_width	Width of each removed channel in edges.cube.channel.network().

Details

The occupied-grid, recursive masked-grid, and Sierpinski families are exposed explicitly rather than overloading a single generator with layout-dimension-dependent behavior: edges.occupied.mesh() builds a finite perforated-mesh family from an occupancy matrix, edges.recursive.mask.grid() builds a generic square-mask family, edges.recursive.triangle.mask() builds a generic triangle-mask family, edges.recursive.tetrahedron.mask() builds a generic tetrahedron-mask family, edges.recursive.cube.mask() builds a generic cube-mask family, edges.vicsek() builds the connected axial-cross variant, edges.menger.sponge() builds the classic cubical sponge variant, edges.triangulated.polyhedron() builds a generic irregular triangulated-surface family, edges.sierpinski.triangle() builds the 2-simplex family, edges.sierpinski.tetrahedron() builds the 3-simplex family, and edges.sierpinski.carpet() builds a 2D cell-adjacency carpet graph.

Value

A two-column integer matrix of undirected edges. Vertex labels are consecutive integers starting at 1.

Functions

- edges.path(): Path graph on n vertices.
- edges.cycle(): Cycle graph on n vertices.
- edges.mesh(): Rectangular grid graph with h rows and w columns.
- edges.occupied.mesh(): Rectangular occupied-grid graph whose vertices are kept cells and whose edges connect orthogonally adjacent kept cells. With connectivity = "diagonal", both diagonals are added for every fully occupied 2-by-2 block.
- edges.cylinder(): Cylindrical grid graph with h rows and wrapped width w.
- edges.torus(): Toroidal grid graph with wrapped height and width.
- edges.irregular.ball(): Deterministically irregular tetrahedralized ball graph built from nested subdivided polyhedral shells connected by a layered prism-to-tetrahedra edge pattern.
- edges.irregular.shell(): Deterministically irregular tetrahedralized shell graph built from nested subdivided polyhedral shells connected by a layered prism-to-tetrahedra edge pattern.
- edges.irregular.torus(): Deterministically irregular torus graph built from major-cycle rings with varying sample counts and stitched into a locally triangulated closed surface.

- edges.sphere(): Sphere surface graph with h latitude levels (including the poles) and wrapped longitude w.
- edges.irregular.annulus(): Deterministically irregular annulus graph built from concentric sample rings with varying sample counts and stitched into a locally triangulated surface-with-boundary graph.
- edges.irregular.pair.of.pants(): Deterministically irregular pair-of-pants graph built from horizontal slice intervals with varying sample counts and stitched into a locally triangulated surface-with-boundary graph.
- edges.irregular.double.torus(): Deterministically irregular double-torus graph built from cyclic slices whose cross-sections follow a '1 -> 3 -> 1' loop transition between two poles.
- edges.irregular.sphere(): Deterministically irregular sphere graph built from latitude bands with varying sample counts and stitched into a locally triangulated closed surface.
- edges.cube(): Cube surface graph on the boundary of a side x side x side lattice.
- edges.kary.tree(): Full k-ary tree of depth depth.
- edges.recursive.mask.grid(): Recursively refined square-mask grid graph whose vertices are occupied cells and whose edges connect orthogonally adjacent cells.
- edges.recursive.triangle.mask(): Recursively refined triangle-mask graph whose vertices are the retained subdivision vertices of an equilateral triangle.
- edges.recursive.tetrahedron.mask(): Recursively refined tetrahedron-mask graph whose vertices are the retained subdivision vertices of a tetrahedron.
- edges.recursive.cube.mask(): Recursively refined cube-mask graph whose vertices are occupied subcubes and whose edges connect face-adjacent occupied cells.
- edges.triangulated.polyhedron(): Triangulated closed-surface graph obtained by repeatedly splitting the triangular faces of a tetrahedron, octahedron, or icosahedron.
- edges.triangulated.annulus(): Triangulated annulus graph obtained by clipping a regular triangular lattice to the region between two concentric circles.
- edges.triangulated.pair.of.pants(): Triangulated pair-of-pants graph obtained by clipping a regular triangular lattice to a disk with two interior circular holes.
- edges.vicsek(): Connected Vicsek-style cross family derived from a 3 x 3 axial-cross keep-mask.
- edges.menger.sponge(): Classic Menger-sponge cubical cell-adjacency graph derived from the 3 x 3 x 3 keep-mask that removes the center cube and the six face-center cubes at each recursion step.
- edges.cube.periodic.tunnels(): Periodic cubical tunnel family derived from a repeated tunnel-band keep-mask. The classic Menger sponge appears as the side = 3, tunnel_width = 1 special case.
- edges.cube.asymmetric.cavities(): Cubical porous family with two offset interior cavity blocks repeated recursively.
- edges.cube.channel.network(): Cubical porous family with a deterministic branched channel network carved through each recursive block.
- edges.sierpinski.triangle(): Two-dimensional Sierpinski triangle graph at recursion depth level.

- `edges.sierpinski.tetrahedron()`: Three-dimensional tetrahedral Sierpinski graph at recursion depth `level`.
- `edges.sierpinski.carpet()`: Two-dimensional Sierpinski carpet graph whose vertices are occupied cells and whose edges connect orthogonally adjacent cells.

Examples

```
edges <- edges.path(6)
coords <- grip(edges, n = 6, dim = 2, seed = 1)
plot.layout(coords, edges, main = "Path graph", pch = 16, cex = 0.8)
edges <- edges.sierpinski.triangle(2)
n <- max(edges)
coords <- grip(edges, n = n, dim = 2,
                      placement = "circle",
                      seed = 1)
plot.layout(coords, edges, main = "Sierpinski triangle", pch = 16, cex = 0.7)
```

`grip` *Compute a GRIP layout*

Description

This is the primary layout API. It uses the quality-first multiscale GRIP engine with extra coarse-level global repulsion to reduce foldovers while preserving the usual GRIP refinement structure. With `preset = NULL`, the default profile is tuned for higher-quality layouts and automatically tapers `final_rounds` on larger graphs.

Usage

```
grip(
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  dim = 3,
  placement = c("barycenter", "circle"),
  preset = NULL,
  rounds = 160,
  final_rounds = 384,
  num_init = 24,
  num_nbrs = 20,
  r = 0.03,
  s = 7.5,
  repulsion_factor = 2.5,
  coarse_repulsion_factor = 1.5,
  coarse_repulsion_sample = 16,
  coarse_repulsion_exact_below = 64,
```

```
  final_anchor_factor = 0,
  final_move_scale_after_first = 1,
  final_mode = c("fr", "kk_repulse"),
  insertion_anchor_count = 3,
  insertion_anchor_scope = c("any_higher", "prev_misf"),
 insertion_anchor_strategy = c("first", "distance_band", "balanced_band", "spread_prev"),
  level0_insertion_mode = c("inherit", "barycenter", "least_squares"),
  level0_anchor_count = insertion_anchor_count,
  level0_local_kk_steps = 3,
  lgkk_polish_rounds = 0L,
  lgkk_multiscale_rounds = 0L,
  lgkk_rounds_coarse = NULL,
  lgkk_rounds_pre_final = NULL,
  lgkk_rounds_final = NULL,
  lgkk_local_nbrs = 20L,
  lgkk_landmark_count = 8L,
  lgkk_multiscale_scope = c("all", "coarse"),
  lgkk_active_limit = 4096L,
  tinit_factor = 6,
  seed = 6,
  disconnected = c("components", "error"),
  metric = c("hop", "edge_length"),
  metric_neighbor_cap = NULL,
  length_normalization = c("median", "mean", "none")
)
```

Arguments

edges	Two-column integer matrix of edges (1-based vertex ids).
n	Number of vertices.
adj_list	Adjacency list (1-based) for undirected graphs.
weight_list	Parallel list of edge lengths for adj_list. The edge-length-metric engine requires it; in the hop-metric engine, NULL treats all edges as length 1. All supplied lengths must be finite and strictly positive.
edge_weights	Vector of edge lengths for edges, in the same order as its rows. The edge-length-metric engine requires it; in the hop-metric engine, NULL treats all edges as length 1. All supplied lengths must be finite and strictly positive. The selected engine determines whether lengths also define graph distances.
dim	Layout dimension (2 or 3). Default is 3.
placement	Initial placement strategy. "circle" is only used for 2D.
preset	Optional tuning preset. NULL uses the quality-first defaults. "carpet" applies a preset tuned for Sierpinski-carpet-like graphs and validated on carpet levels 3 and 4. "mesh" applies a preset tuned for rectangular lattice graphs and validated on 8x8 and 12x12 mesh layouts. "torus" applies a preset tuned for 3D torus layouts and validated on torus sizes from 8x8 through 20x20. "tree" applies

a preset tuned for symmetric force-directed layouts of tree-like graphs and validated on binary trees of depths 5 and 6. Presets only fill in tuning arguments that you did not supply explicitly.

`rounds` Initial rounds for refinement.

`final_rounds` Final rounds for refinement.

`num_init` Number of initial vertices in the coarsest level.

`num_nbrs` Maximum number of graph-distance neighbors retained for local refinement at each filtration level.

`r` Main local temperature adaptation rate in `[0, 1]`.

`s` Non-negative boost factor applied when successive displacements have a consistent direction.

`repulsion_factor`
Non-negative multiplier applied to GRIP's finest-level repulsive force scale.

`coarse_repulsion_factor`
Non-negative multiplier applied to the extra coarse-level active-set repulsion term. `0` disables that extra term.

`coarse_repulsion_sample`
Positive integer sample size used to approximate active-set-wide repulsion on larger coarse levels.

`coarse_repulsion_exact_below`
Positive integer threshold. When the active set size is at most this value, the coarse repulsion is computed exactly against all currently active vertices instead of being sampled.

`final_anchor_factor`
Non-negative multiplier for an anchor term that pulls the final FR stage back toward the pre-final full-graph layout. '0' disables the anchor and preserves the current behavior.

`final_move_scale_after_first`
Scalar in '[0, 1]' applied to the final FR displacement after the first finest-level round. Values below '1' damp later full-graph movement while keeping the first FR round unchanged.

`final_mode` Final full-graph refinement mode. `"fr"` keeps the current Fruchterman-Reingold-style final stage. `"kk_repulse"` uses a KK-style local distance-matching update with explicit active-set repulsion instead of the final FR phase.

`insertion_anchor_count`
Positive integer number of anchor vertices used during multiscale insertion on non-initial MISF refinement levels. This is the closest current implementation to a global `K_mish` parameter.

`insertion_anchor_scope`
Anchor-eligibility rule used during multiscale insertion. `"any_higher"` matches the historical GRIP behavior and allows anchors from any already placed higher MISF level. `"prev_misf"` restricts anchors to the immediately previous MISF level only.

`insertion_anchor_strategy`
Anchor-selection rule used during multiscale insertion. `"first"` keeps the historical first-anchors-found BFS behavior. `"distance_band"` keeps exploring

until the K_mish-th anchor distance band is exhausted, then places the new vertex from that less order-sensitive anchor pool. "balanced_band" uses the same band expansion, then explicitly selects a subset whose centroid stays centered in the candidate cloud while remaining geometrically spread out. "spread_prev" is a symmetry-oriented band strategy intended to be paired with insertion_anchor_scope = "prev_misf"; it selects anchors with broad angular and geometric coverage before placement.

level0_insertion_mode
: Level-0 insertion placement override used only when the finest filtration level is first populated. "inherit" keeps the current GRIP behavior. "barycenter" disables the 2D circle heuristic at level 0 and uses barycentric anchor placement. "least_squares" uses a multi-anchor least-squares distance fit at level 0 before any local micro-polish.

level0_anchor_count
: Positive integer number of already placed anchors to collect for level-0 insertion experiments. By default this inherits insertion_anchor_count. The legacy behavior uses 3.

level0_local_kk_steps
: Non-negative integer number of tiny local KK micro-polish steps applied immediately after each level-0 insertion. The legacy behavior uses 3.

lgkk_polish_rounds
: Non-negative integer number of experimental landmark-geodesic KK polish iterations applied after the main GRIP solve. 0 disables the polish.

lgkk_multiscale_rounds
: Non-negative integer number of compiled landmark-geodesic KK refinement rounds applied inside the multiscale solver after each eligible MISF level completes its standard GRIP rounds. This legacy shared budget is used as a fallback when any of the more specific per-stage budgets below are left NULL.

lgkk_rounds_coarse
: Optional non-negative integer number of compiled LGKK rounds applied on coarse MISF levels with misf_level > 1. When NULL, this falls back to lgkk_multiscale_rounds.

lgkk_rounds_pre_final
: Optional non-negative integer number of compiled LGKK rounds applied on the last coarse level just before the full graph is opened (misf_level == 1). When NULL, this falls back to lgkk_multiscale_rounds.

lgkk_rounds_final
: Optional non-negative integer number of compiled LGKK rounds applied after the full graph level completes its standard GRIP rounds (misf_level == 0). When NULL, this falls back to lgkk_multiscale_rounds.

lgkk_local_nbrs
: Number of nearest graph-metric neighbors retained per vertex in the LGKK sparse local set when either LGKK stage is enabled.

lgkk_landmark_count
: Number of farthest-point landmarks retained per vertex in the LGKK sparse long-range set when either LGKK stage is enabled.

`lgkk_multiscale_scope`
: Scope for the compiled multiscale LGKK stage. `"all"` applies it after every eligible MISF level, including the final full-graph level. `"coarse"` applies it only on coarse levels.

`lgkk_active_limit`
: Positive integer upper bound on the active-set size for compiled multiscale LGKK cache construction. Levels larger than this skip the multiscale LGKK stage.

`tinit_factor`
: Initial temperature factor.

`seed`
: Optional RNG seed for reproducibility. If NULL, uses current time.

`disconnected`
: How to handle disconnected graphs: `"components"` (default) lays out each connected component separately and packs them into one coordinate matrix; `"error"` stops with an error.

`metric`
: Graph metric used by the multiscale GRIP engine. `"hop"` (default) uses unweighted shortest-path hop counts to build the MISF hierarchy and graph neighborhoods. `"edge_length"` uses shortest-path distances obtained by summing the supplied positive edge lengths. See **Edge-length semantics** for the exact role of edge lengths in each mode.

`metric_neighbor_cap`
: Weighted-metric search limit used only when `metric = "edge_length"`. `NULL` performs the exact weighted neighborhood search and stops once the required neighbors and anchors are filled. A positive integer enables an approximate search by limiting the number of settled vertices per search. It is an error to supply this argument with `metric = "hop"`.

`length_normalization`
: Global normalization applied only when `metric = "edge_length"`: `"median"` (default) divides every edge length by their median, `"mean"` divides by their mean, and `"none"` preserves the supplied numerical scale. It is an error to supply this argument with `metric = "hop"`.

Details

Edge-length semantics

The arguments `edge_weights` and `weight_list` are historically named but represent positive edge *lengths* or traversal costs, not connection strengths, capacities, or similarities. A larger value requests a longer geometric edge. If the available values are strengths for which a larger value means a closer connection, convert them to positive lengths before calling `grip()`, for example with a scientifically appropriate reciprocal or other monotone decreasing transformation.

With `metric = "hop"`, the standard GRIP hierarchy, insertion anchors, and retained local neighborhoods use combinatorial shortest-path distance: every traversed edge contributes one hop. When edge lengths are supplied, they are nevertheless used as the desired lengths of adjacent vertex pairs in the attractive force calculation. Thus this mode is useful when topology should determine the multiscale organization but adjacent edges should have unequal target lengths. No global length normalization is performed in this mode. If no lengths are supplied, every edge has desired length one. Optional LGKK stages use the supplied edge lengths for their geodesic distances even though the standard GRIP stages remain hop based.

With `metric = "edge_length"`, positive edge lengths are required. The same lengths determine desired adjacent-edge lengths and the shortest-path metric used for the MISF hierarchy, insertion

anchors, retained graph neighborhoods, and LGKK stages. Dijkstra-style weighted shortest paths are used instead of hop-count breadth-first searches. By default, all lengths are divided by their median before layout; this preserves relative geometry while placing the numerical scale near the solver's unit scale. Use `length_normalization = "mean"` for mean scaling or `"none"` when the absolute supplied scale is intentional. Multiplying all input lengths by the same positive constant therefore leaves the default normalized solve unchanged.

For `edges` input, provide one value per row through `edge_weights`. For `adj_list` input, provide a parallel `weight_list`: `weight_list[[i]][j]` is the length of the edge from vertex `i` to `adj_list[[i]][j]`. For an undirected graph, the adjacency and length entries should be symmetric.

Value

A numeric matrix with ‘n‘ rows and ‘dim‘ columns.

References

Gajer, P. and Kobourov, S.G. (2002). GRIP: Graph dRawing with Intelligent Placement. *Journal of Graph Algorithms and Applications*, 6(3), 203–224. doi:10.7155/jgaa.00052.

Gajer, P., Goodrich, M.T. and Kobourov, S.G. (2004). A multi-dimensional approach to force-directed layouts of large graphs. *Computational Geometry*, 29(1), 3–18. doi:10.1016/j.comgeo.2004.03.014.

Examples

```
edges <- edges.mesh(4, 4)
coords <- grip(edges, n = max(edges), dim = 2,
                      coarse_repulsion_factor = 0.2,
                      coarse_repulsion_sample = 8,
                      coarse_repulsion_exact_below = 32,
                      seed = 1)
round(coords, 3)

# Use edge lengths throughout the multiscale graph metric.
path <- cbind(1:5, 2:6)
lengths <- c(1, 1, 2, 1, 1)
weighted.coords <- grip(
  path, n = 6, edge_weights = lengths,
  metric = "edge_length", dim = 2,
  rounds = 4, final_rounds = 4, num_init = 3, seed = 1
)
```

---

grip.compare.layouts *Deprecated layout comparison name*

Description

This long name is deprecated. Use [compare.layouts()] instead.

Usage

```
grip.compare.layouts(...)
```

Arguments

...	Arguments passed to [compare.layouts()].

Value

See [compare.layouts()].

---

grip.optimize.edge.kk.layout

Deprecated edge-KK layout names

Description

These long names are deprecated. Use [edge.kk()] instead.

Usage

```
grip.optimize.edge.kk.layout(...)

grip.optimize.edge.gkk.layout(...)

grip.optimize.edge.isometric.layout(...)
```

Arguments

...	Arguments passed to [edge.kk()].

Value

A ‘"grip_gmds_layout"‘ object.

grip.prepare.edge.kk *Deprecated edge-KK preparation name*

Description

This long name is deprecated. Use [prepare.edge.kk()] instead.

Usage

```
grip.prepare.edge.kk(...)
```

Arguments

...	Arguments passed to [prepare.edge.kk()].

Value

See [prepare.edge.kk()].

grip.score.layout *Deprecated layout scoring name*

Description

This long name is deprecated. Use [score.layout()] instead.

Usage

```
grip.score.layout(...)
```

Arguments

...	Arguments passed to [score.layout()].

Value

See [score.layout()].

gripui_app *Build the ‘gripui‘ Shiny application*

Description

Build the ‘gripui‘ Shiny application

Usage

```
gripui_app(project)
```

Arguments

project	A ‘gripui_project‘.

Value

A ‘shiny.appobj‘.

Examples

```
graph <- list(adj_list = list(2L, c(1L, 3L), 2L))
layouts <- data.frame(
  candidate = "toy.layout",
  stage = "layout",
  seed = 1L,
  status = "ok",
  stringsAsFactors = FALSE
)
project <- gripui_project(graph = graph, layouts = layouts, title = "Toy project")
app <- gripui_app(project)
inherits(app, "shiny.appobj")
```

gripui_family_app *Build the graph-family geometry explorer Shiny application*

Description

Build the graph-family geometry explorer Shiny application

Usage

```
gripui_family_app(
  catalog = gripui_graph_family_catalog(),
  title = "Graph Family Geometry Explorer",
  subtitle = "Interactive geometry browser for synthetic benchmark families."
)
```

Arguments

catalog	Family catalog, usually ‘gripui_graph_family_catalog()‘.
title	Application title.
subtitle	Optional subtitle shown in the sidebar.

Value

A ‘shiny.appobj‘.

Examples

```
app <- gripui_family_app()
inherits(app, "shiny.appobj")
```

---

gripui_graph_family_catalog

Catalog of graph families for the geometry explorer app

Description

The catalog is a registry used by ‘gripui_family_app()‘ to populate its family selector, presets, parameter controls, builder calls, and source references.

Usage

```
gripui_graph_family_catalog()
```

Value

A named list of family descriptors.

Examples

```
catalog <- gripui_graph_family_catalog()
names(catalog)
catalog$mesh$function_name
```

gripui_project *Create a normalized ‘gripui‘ project object*

Description

Create a normalized ‘gripui‘ project object

Usage

```
gripui_project(
  graph = NULL,
  layouts,
  title = NULL,
  subtitle = NULL,
  notes = NULL
)
```

Arguments

graph	Optional graph object used by layout viewers. This may be a single list containing fields such as ‘adj_list‘, ‘weight_list‘, ‘vertex_data‘, and ‘graph_info‘, a named list of such graph objects for multi-stage projects, or ‘NULL‘ when only catalog exploration is needed.
layouts	Data frame with one row per realized layout.
title	Optional project title.
subtitle	Optional subtitle.
notes	Optional notes string.

Value

An object of class ‘gripui_project‘.

Examples

```
graph <- list(adj_list = list(2L, c(1L, 3L), c(2L, 4L), 3L))
layouts <- data.frame(
  candidate = "path.default",
  stage = "layout",
  seed = 1L,
  status = "ok",
  stringsAsFactors = FALSE
)
project <- gripui_project(graph = graph, layouts = layouts, title = "Path graph")
project$meta$title
```

gripui_project_from_compare *Convert ‘compare.layouts()‘ output into a ‘gripui_project‘*

Description

Convert ‘compare.layouts()‘ output into a ‘gripui_project‘

Usage

```
gripui_project_from_compare(
  compare_obj,
  graph = NULL,
  vertex_data = NULL,
  graph_info = NULL,
  title = NULL
)
```

Arguments

compare_obj	Result of ‘compare.layouts()‘.
graph	Optional graph object, named graph list, or path to a graph RDS. Defaults to ‘NULL‘ when only catalog exploration is needed.
vertex_data	Optional vertex metadata added to ‘graph‘.
graph_info	Optional graph-level metadata added to ‘graph‘.
title	Optional project title.

Value

A ‘gripui_project‘.

Examples

```
edges <- edges.path(5)
cmp <- compare.layouts(
  edges = edges,
  n = 5,
  candidates = "default",
  seeds = 1L,
  return.layouts = TRUE
)
graph <- list(
  adj_list = list(2L, c(1L, 3L), c(2L, 4L), c(3L, 5L), 4L)
)
project <- gripui_project_from_compare(cmp, graph = graph, title = "Path compare")
nrow(project$layouts)
```

gripui_project_from_dir *Create a 'gripui_project' from a directory of saved artifacts*

Description

Create a 'gripui_project' from a directory of saved artifacts

Usage

```
gripui_project_from_dir(root, graph = NULL, title = NULL, subtitle = NULL)
```

Arguments

root	Directory containing either a normalized 'gripui' bundle or a saved search output tree such as the current HMP/U01 layout-selection outputs. The normalized bundle format is the preferred long-term producer contract; support for HMP-style run tables and manifests is a compatibility adapter for older outputs.
graph	Optional graph object, named graph list, or path to a graph RDS.
title	Optional project title.
subtitle	Optional subtitle.

Value

A 'gripui_project'.

Examples

```
root <- tempfile("gripui-bundle-")
dir.create(root)
on.exit(unlink(root, recursive = TRUE, force = TRUE), add = TRUE)
dir.create(file.path(root, "catalog"), recursive = TRUE, showWarnings = FALSE)
dir.create(file.path(root, "graph"), recursive = TRUE, showWarnings = FALSE)
dir.create(file.path(root, "artifacts", "layouts"), recursive = TRUE, showWarnings = FALSE)

graph <- list(adj_list = list(2L, c(1L, 3L), 2L))
saveRDS(graph, file.path(root, "graph", "graph.rds"))

coords.path <- file.path(root, "artifacts", "layouts", "toy_embedding.tsv")
utils::write.table(
  data.frame(
    vertex_id = c("v1", "v2", "v3"),
    x = c(1, 2, 3),
    y = c(0, 0, 0),
    stringsAsFactors = FALSE
  ),
  file = coords.path,
  sep = "\t",
```

```
  quote = TRUE,
  row.names = FALSE,
  col.names = TRUE
)
utils::write.table(
  data.frame(
    candidate = "toy.layout",
    stage = "bundle",
    seed = 1L,
    status = "ok",
    coords_path = file.path("artifacts", "layouts", "toy_embedding.tsv"),
    stringsAsFactors = FALSE
  ),
  file = file.path(root, "catalog", "layout_catalog.tsv"),
  sep = "\t",
  quote = TRUE,
  row.names = FALSE,
  col.names = TRUE
)

project <- gripui_project_from_dir(root, title = "My project")
project$layouts$availability
```

`gripui_validate_project` *Validate a 'gripui_project'*

Description

Validate a 'gripui_project'

Usage

```
gripui_validate_project(project)
```

Arguments

`project`	Object to validate.

Value

Invisibly returns 'TRUE' when validation succeeds.

Examples

```
layouts <- data.frame(
  candidate = "test.layout",
  stage = "layout",
  seed = 1L,
  status = "ok",
```

```
  stringsAsFactors = FALSE
)
project <- gripui_project(graph = NULL, layouts = layouts, title = "Validation example")
gripui_validate_project(project)
```

`hmp.u01.gc.coarse` *HMP/U01 Coarsened Giant-Component Graph*

Description

A bundled coarsened graph derived from the giant connected component of the HMP+U01 16S amplicon iKNN graph used in the HMP/U01 layout-selection study. The upstream graph came from the '>=1 with 'k = 3', after restricting to the biologically meaningful major component. The graph was then greedily coarsened in two rounds from '6474' vertices to '1828' supernodes.

Usage

```
hmp.u01.gc.coarse
```

Format

A named list with components:

adj_list Adjacency list of length '1828', with 1-based integer neighbor indices.

weight_list Parallel list of positive edge weights.

vertex_data Data frame with one row per coarse vertex and columns 'vertex_id', 'size', 'cst', 'subcst', 'dcst.depth1.absorb', 'dcst.depth1.rare', 'dcst.depth2.absorb', 'dcst.depth2.rare', 'ph', and 'log10_reads'.

graph_info Named list summarizing provenance and graph-level metadata, including the original and coarsened vertex counts, selected 'k', edge count, and coarsening rounds.

Details

The object is intended for examples and vignettes on real-world 3D layout comparison. It is small enough to ship with the package while preserving the arm-like graph structure that motivated the tuned preset analysis.

Source

Derived from the publicly available HMP+U01 16S amplicon analysis workflow used in the 'chm_paper' project. Raw bundled artifacts and a provenance note are available under 'inst/extdata/hmp_u01_gc_coarse/'.

Examples

```
data(hmp.u01.gc.coarse)
length(hmp.u01.gc.coarse$adj_list)
head(hmp.u01.gc.coarse$vertex_data[, c("vertex_id", "size", "cst", "subcst")])
```

`irregular_annulus_surface_helpers` *Weighted irregular annulus surface helpers*

Description

Convenience helpers that build an annulus from deterministically irregular concentric rings with varying sample counts, stitch adjacent rings into a locally triangulated graph, and then optionally lift the resulting irregular planar parameterization into $\mathbb{R}^3$. This provides a non-lattice annulus family with boundary and nonuniform local valence.

Usage

```
irregular.annulus.surface.embedding(
  rings = 6,
  outer_count = 28,
  outer_radius = 1,
  inner_radius = 0.45,
  count_irregularity = 0.2,
  radial_irregularity = 0.35,
  phase_twist = 0.35,
  surface = c("flat", "saddle", "paraboloid", "ripple", "folded"),
  amplitude = 0.6,
  freq_u = 1,
  freq_v = 1
)

irregular.annulus.surface.graph(
  rings = 6,
  outer_count = 28,
  outer_radius = 1,
  inner_radius = 0.45,
  count_irregularity = 0.2,
  radial_irregularity = 0.35,
  phase_twist = 0.35,
  surface = c("flat", "saddle", "paraboloid", "ripple", "folded"),
  amplitude = 0.6,
  freq_u = 1,
  freq_v = 1,
  normalize = c("median", "mean", "none")
)
```

Arguments

`rings`	Number of concentric sample rings, including the inner and outer boundary cycles.
`outer_count`	Approximate number of vertices on the outer boundary.

`outer_radius` Positive outer annulus radius.

`inner_radius` Positive inner annulus radius.

`count_irregularity`
: Irregularity level for the per-ring sample counts. Must lie in `[0, 1)`.

`radial_irregularity`
: Irregularity level for within-ring radial perturbations. Must lie in `[0, 1]`.

`phase_twist` Finite angular phase offset used to desynchronize neighboring rings.

`surface` Geometry family used for the 3D lift. One of `"flat"`, `"saddle"`, `"paraboloid"`, `"ripple"`, or `"folded"`.

`amplitude` Finite deformation amplitude.

`freq_u` Positive ripple frequency in the first planar coordinate. Used only when `surface = "ripple"`.

`freq_v` Positive ripple frequency in the second planar coordinate. Used only when `surface = "ripple"`.

`normalize` Normalization applied to the induced edge lengths. One of `"median"`, `"mean"`, or `"none"`.

Value

`irregular.annulus.surface.embedding()` returns an `n x 3` numeric matrix with columns `x`, `y`, and `z`.

`irregular.annulus.surface.graph()` returns a list with components:

- `edges`: the irregular annulus edges,
- `n`: number of vertices,
- `edge_weights`: induced positive edge lengths,
- `coords_surface`: the 3D embedding,
- `coords_param`: the irregular planar annulus coordinates,
- `weight_scale`: the normalization constant applied to the raw edge lengths,
- `family`: always `"irregular.annulus"`,
- `surface`: the chosen surface name,
- `rings`: number of concentric rings,
- `ring_sizes`: sample counts on each ring,
- `label`: a human-readable family label.

irregular_ball_solid_helpers
Weighted irregular ball solid helpers

Description

Convenience helpers that build a layered simplicial ball by taking a subdivided triangulated polyhedron, normalizing its vertices to directions on the sphere, placing those directions on nested radial layers, and connecting adjacent layers by a deterministic prism-to-tetrahedra edge pattern. The center vertex is included explicitly, so the resulting graph is genuinely volumetric rather than just a closed surface.

Usage

```
irregular.ball.solid.embedding(
  base = c("tetrahedron", "octahedron", "icosahedron"),
  level = 1,
  layers = 3,
  outer_radius = 1,
  radial_irregularity = 0.25,
  layer_twist = 0.35,
  surface = c("standard", "bulged", "twisted", "wavy"),
  amplitude = 0.2,
  freq_theta = 2,
  freq_phi = 2,
  twist = 0.6
)

irregular.ball.solid.graph(
  base = c("tetrahedron", "octahedron", "icosahedron"),
  level = 1,
  layers = 3,
  outer_radius = 1,
  radial_irregularity = 0.25,
  layer_twist = 0.35,
  surface = c("standard", "bulged", "twisted", "wavy"),
  amplitude = 0.2,
  freq_theta = 2,
  freq_phi = 2,
  twist = 0.6,
  normalize = c("median", "mean", "none")
)
```

Arguments

base	Base polyhedron. One of "tetrahedron", "octahedron", or "icosahedron".
level	Surface subdivision depth used for each radial layer.

layers	Number of non-center radial layers.
outer_radius	Positive outer radius of the ball.
radial_irregularity	Irregularity level for the radial layer spacing. Must lie in [0, 1].
layer_twist	Finite z-axis twist applied smoothly across radial layers.
surface	Solid geometry family. One of "standard", "bulged", "twisted", or "wavy".
amplitude	Finite deformation amplitude.
freq_theta	Positive azimuthal modulation frequency used only when surface = "wavy".
freq_phi	Positive polar modulation frequency used only when surface = "wavy".
twist	Finite twist strength used only when surface = "twisted".
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Value

irregular.ball.solid.embedding() returns an n x 3 numeric matrix with columns x, y, and z.

irregular.ball.solid.graph() returns a list with components:

- edges: the irregular ball edges,
- n: number of vertices,
- edge_weights: induced positive edge lengths,
- coords_surface: the 3D embedding,
- coords_param: the canonical 3D ball coordinates,
- weight_scale: the normalization constant applied to the raw edge lengths,
- family: always "irregular.ball",
- surface: the chosen solid geometry family,
- base: the chosen base polyhedron,
- level: the surface subdivision depth,
- layers: number of non-center radial layers,
- radii: the radial layer values,
- label: a human-readable family label.

`irregular_double_torus_surface_helpers` *Weighted irregular double-torus surface helpers*

Description

Convenience helpers that build a closed genus-2 surface from a deterministically irregular slice family whose cyclic cross-sections follow a ‘1 -> 3 -> 1‘ loop transition between two poles. This gives a non-lattice, point-sampled double-torus graph together with either the canonical 3D embedding or simple bulged, twisted, and wavy deformations of that geometry.

Usage

```
irregular.double.torus.surface.embedding(
  slices = 11,
  tube_count = 14,
  branch_length = 0.85,
  branch_offset = 0.72,
  tube_radius = 0.28,
  transition_width = 0.42,
  count_irregularity = 0.2,
  axial_irregularity = 0.3,
  phase_twist = 0.35,
  surface = c("standard", "bulged", "twisted", "wavy"),
  amplitude = 0.25,
  freq_x = 2,
  freq_theta = 2,
  twist = 0.6
)

irregular.double.torus.surface.graph(
  slices = 11,
  tube_count = 14,
  branch_length = 0.85,
  branch_offset = 0.72,
  tube_radius = 0.28,
  transition_width = 0.42,
  count_irregularity = 0.2,
  axial_irregularity = 0.3,
  phase_twist = 0.35,
  surface = c("standard", "bulged", "twisted", "wavy"),
  amplitude = 0.25,
  freq_x = 2,
  freq_theta = 2,
  twist = 0.6,
  normalize = c("median", "mean", "none")
)
```

Arguments

slices	Number of non-pole slices through the double torus.
tube_count	Approximate number of vertices around each tube-like loop.
branch_length	Half-length of the three-loop central region.
branch_offset	Offset of the outer loop centers from the central loop.
tube_radius	Baseline radius of each tube-like loop.
transition_width	Width of the left and right transition regions between the single-loop and three-loop slices.
count_irregularity	Irregularity level for the per-component sample counts. Must lie in [0, 1).
axial_irregularity	Irregularity level for the slice spacing. Must lie in [0, 1].
phase_twist	Finite phase offset used to desynchronize neighboring cyclic slices.
surface	Double-torus geometry family. One of "standard", "bulged", "twisted", or "wavy".
amplitude	Finite deformation amplitude.
freq_x	Positive modulation frequency along the axial direction. Used only when surface = "wavy".
freq_theta	Positive angular modulation frequency around the local tube direction. Used only when surface = "wavy".
twist	Finite twist strength used only when surface = "twisted".
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Value

irregular.double.torus.surface.embedding() returns an n x 3 numeric matrix with columns x, y, and z.

irregular.double.torus.surface.graph() returns a list with components:

- edges: the irregular double-torus edges,
- n: number of vertices,
- edge_weights: induced positive edge lengths,
- coords_surface: the 3D embedding,
- coords_param: the canonical 3D double-torus coordinates,
- weight_scale: the normalization constant applied to the raw edge lengths,
- family: always "irregular.double.torus",
- surface: the chosen surface name,
- slices: number of non-pole slices,
- slice_sizes: vertex counts in each slice,
- slice_components: number of cyclic components in each slice,
- label: a human-readable family label.

`irregular_pair_of_pants_surface_helpers` *Weighted irregular pair-of-pants surface helpers*

Description

Convenience helpers that build a pair-of-pants domain from deterministically irregular horizontal slices, sample each surviving slice interval with varying point counts, and stitch adjacent slices into a locally triangulated surface-with-boundary graph. The resulting irregular planar parameterization can then be lifted into $\mathbb{R}^3$ using the same planar surface families as the triangulated annulus and irregular annulus helpers.

Usage

```
irregular.pair.of.pants.surface.embedding(
  slices = 11,
  outer_count = 28,
  outer_radius = 1.1,
  hole_radius = 0.24,
  hole_offset = 0.38,
  hole_height = 0.18,
  count_irregularity = 0.2,
  vertical_irregularity = 0.35,
  phase_twist = 0.35,
  surface = c("flat", "saddle", "paraboloid", "ripple", "folded"),
  amplitude = 0.6,
  freq_u = 1,
  freq_v = 1
)

irregular.pair.of.pants.surface.graph(
  slices = 11,
  outer_count = 28,
  outer_radius = 1.1,
  hole_radius = 0.24,
  hole_offset = 0.38,
  hole_height = 0.18,
  count_irregularity = 0.2,
  vertical_irregularity = 0.35,
  phase_twist = 0.35,
  surface = c("flat", "saddle", "paraboloid", "ripple", "folded"),
  amplitude = 0.6,
  freq_u = 1,
  freq_v = 1,
  normalize = c("median", "mean", "none")
)
```

Arguments

slices	Number of horizontal sample slices through the pair-of-pants domain.
outer_count	Approximate number of vertices across the widest slices.
outer_radius	Positive outer boundary radius.
hole_radius	Positive radius of each interior hole.
hole_offset	Positive horizontal offset of the two hole centers.
hole_height	Shared vertical coordinate of the two hole centers.
count_irregularity	Irregularity level for the per-slice sample counts. Must lie in [0, 1).
vertical_irregularity	Irregularity level for slice spacing. Must lie in [0, 1].
phase_twist	Finite phase offset used to desynchronize neighboring slice samples.
surface	Geometry family used for the 3D lift. One of "flat", "saddle", "paraboloid", "ripple", or "folded".
amplitude	Finite deformation amplitude.
freq_u	Positive ripple frequency in the first planar coordinate. Used only when surface = "ripple".
freq_v	Positive ripple frequency in the second planar coordinate. Used only when surface = "ripple".
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Value

irregular.pair.of.pants.surface.embedding() returns an n x 3 numeric matrix with columns x, y, and z.

irregular.pair.of.pants.surface.graph() returns a list with components:

- edges: the irregular pair-of-pants edges,
- n: number of vertices,
- edge_weights: induced positive edge lengths,
- coords_surface: the 3D embedding,
- coords_param: the irregular planar pair-of-pants coordinates,
- weight_scale: the normalization constant applied to the raw edge lengths,
- family: always "irregular.pair.of.pants",
- surface: the chosen surface name,
- slices: number of horizontal sample slices,
- slice_sizes: vertex counts in each slice,
- slice_components: number of connected slice intervals in each slice,
- label: a human-readable family label.

irregular_rectangle_surface_helpers
Irregular rectangle surface helpers

Description

Convenience helpers that keep ordinary rectangular-mesh adjacency but replace the regular parameter grid by a deterministic irregular rectangle. The resulting family stays simply connected and rectangular in topology while breaking the strongest row/column symmetries of the lattice.

Usage

```
irregular.rectangle.param.coords(
  h,
  w = h,
  x_scale = 1,
  y_scale = 1,
  row_irregularity = 0.2,
  col_irregularity = 0.2,
  row_phase = 0.35,
  col_phase = 0.65,
  interior_warp = 0.08,
  shear = 0,
  min_step_ratio = 0.3
)

irregular.rectangle.surface.embedding(
  h,
  w = h,
  surface = c("flat", "saddle", "paraboloid", "ripple"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  x_scale = 1,
  y_scale = 1,
  row_irregularity = 0.2,
  col_irregularity = 0.2,
  row_phase = 0.35,
  col_phase = 0.65,
  interior_warp = 0.08,
  shear = 0,
  min_step_ratio = 0.3
)

irregular.rectangle.surface.graph(
  h,
  w = h,
```

```
  surface = c("flat", "saddle", "paraboloid", "ripple"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  x_scale = 1,
  y_scale = 1,
  row_irregularity = 0.2,
  col_irregularity = 0.2,
  row_phase = 0.35,
  col_phase = 0.65,
  interior_warp = 0.08,
  shear = 0,
  min_step_ratio = 0.3,
  connectivity = c("orthogonal", "diagonal"),
  normalize = c("median", "mean", "none")
)
```

Arguments

h	Number of rows.
w	Number of columns. Defaults to h.
x_scale	Positive horizontal scaling of the parameter domain.
y_scale	Positive vertical scaling of the parameter domain.
row_irregularity	Irregularity level for the row spacing. Must lie in [0, 1].
col_irregularity	Irregularity level for the column spacing. Must lie in [0, 1].
row_phase	Finite phase shift for the deterministic row-spacing perturbation.
col_phase	Finite phase shift for the deterministic column-spacing perturbation.
interior_warp	Non-negative interior warp strength. Must lie in [0, 1]. The warp vanishes on the boundary.
shear	Finite affine shear applied after the boundary-vanishing warp. Larger values can force a validation error if the mesh cells invert.
min_step_ratio	Positive lower bound for the perturbed row and column interval lengths, expressed relative to the unperturbed interval scale. Must lie in (0, 1].
surface	Surface family used for the lift. One of "flat", "saddle", "paraboloid", or "ripple".
amplitude	Finite numeric amplitude controlling the non-flat displacement.
freq_u	Positive ripple frequency in the horizontal parameter direction. Used only when surface = "ripple".
freq_v	Positive ripple frequency in the vertical parameter direction. Used only when surface = "ripple".
connectivity	Mesh neighborhood rule passed to edges.mesh().
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Details

'irregular.rectangle.param.coords()' returns the irregular planar parameter coordinates. 'irregular.rectangle.surface.embedding()' lifts those coordinates into $\mathbb{R}^3$. 'irregular.rectangle.surface.graph()' returns a reusable weighted-graph bundle with the induced Euclidean edge lengths.

Value

`irregular.rectangle.param.coords()` returns an `n x 2` numeric matrix with columns `u` and `v`.

`irregular.rectangle.surface.embedding()` returns an `n x 3` numeric matrix with columns `x`, `y`, and `z`.

`irregular.rectangle.surface.graph()` returns a list with components:

- `edges`: the undirected rectangular-mesh edges,
- `n`: number of vertices,
- `edge_weights`: induced positive edge lengths,
- `coords_surface`: the 3D surface embedding,
- `coords_param`: the irregular 2D parameter coordinates,
- `coords_regular_param`: the underlying regular 2D parameter coordinates used to define the boundary-vanishing warp,
- `row_breaks`: per-row parameter coordinates in row order,
- `col_breaks`: per-column parameter coordinates in column order,
- `weight_scale`: the normalization constant applied to the raw edge lengths,
- `family`: always `"irregular.rectangle"`,
- `surface`: the chosen surface name,
- `connectivity`: the chosen mesh connectivity rule,
- `label`: a human-readable family label.

`irregular_shell_solid_helpers` *Weighted irregular shell solid helpers*

Description

Convenience helpers that build a layered simplicial shell by placing the vertices of a subdivided triangulated polyhedron on nested inner-to-outer radial layers and connecting adjacent layers by the same deterministic prism-to-tetrahedra edge pattern used for `irregular.ball.solid.*()`. The result is a genuinely volumetric shell graph with a hollow interior.

Usage

```
irregular.shell.solid.embedding(
  base = c("tetrahedron", "octahedron", "icosahedron"),
  level = 1,
  layers = 3,
  inner_radius = 0.45,
  outer_radius = 1,
  radial_irregularity = 0.25,
  layer_twist = 0.35,
  surface = c("standard", "bulged", "twisted", "wavy"),
  amplitude = 0.2,
  freq_theta = 2,
  freq_phi = 2,
  twist = 0.6
)

irregular.shell.solid.graph(
  base = c("tetrahedron", "octahedron", "icosahedron"),
  level = 1,
  layers = 3,
  inner_radius = 0.45,
  outer_radius = 1,
  radial_irregularity = 0.25,
  layer_twist = 0.35,
  surface = c("standard", "bulged", "twisted", "wavy"),
  amplitude = 0.2,
  freq_theta = 2,
  freq_phi = 2,
  twist = 0.6,
  normalize = c("median", "mean", "none")
)
```

Arguments

`base`	Base polyhedron. One of `"tetrahedron"`, `"octahedron"`, or `"icosahedron"`.
`level`	Surface subdivision depth used for each radial layer.
`layers`	Number of non-center radial layers.
`inner_radius`	Positive inner radius of the shell.
`outer_radius`	Positive outer radius of the ball.
`radial_irregularity`	Irregularity level for the radial layer spacing. Must lie in `[0, 1]`.
`layer_twist`	Finite z-axis twist applied smoothly across radial layers.
`surface`	Solid geometry family. One of `"standard"`, `"bulged"`, `"twisted"`, or `"wavy"`.
`amplitude`	Finite deformation amplitude.
`freq_theta`	Positive azimuthal modulation frequency used only when `surface = "wavy"`.

freq_phi	Positive polar modulation frequency used only when surface = "wavy".
twist	Finite twist strength used only when surface = "twisted".
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Value

irregular.shell.solid.embedding() returns an n x 3 numeric matrix with columns x, y, and z.

irregular.shell.solid.graph() returns the same components as irregular.ball.solid.graph(), with family set to "irregular.shell".

---

irregular_sphere_surface_helpers

Weighted irregular sphere surface helpers

Description

Convenience helpers that build a sphere from deterministically irregular latitude bands with varying sample counts, stitch adjacent bands into a locally triangulated graph, and then realize that graph in $\mathbb{R}^3$ using either a standard sphere, an ellipsoid, or a wavy radial modulation.

Usage

```
irregular.sphere.surface.embedding(
  bands = 6,
  equator_count = 28,
  count_irregularity = 0.2,
  lat_irregularity = 0.35,
  phase_twist = 0.35,
  surface = c("standard", "ellipsoid", "wavy"),
  radius = 1,
  amplitude = 0.2,
  freq_theta = 3,
  freq_lat = 2,
  twist = 0.25
)

irregular.sphere.surface.graph(
  bands = 6,
  equator_count = 28,
  count_irregularity = 0.2,
  lat_irregularity = 0.35,
  phase_twist = 0.35,
  surface = c("standard", "ellipsoid", "wavy"),
  radius = 1,
  amplitude = 0.2,
```

```
  freq_theta = 3,
  freq_lat = 2,
  twist = 0.25,
  normalize = c("median", "mean", "none")
)
```

Arguments

bands	Number of non-pole latitude bands.
equator_count	Approximate number of vertices near the equator.
count_irregularity	Irregularity level for per-band sample counts. Must lie in [0, 1).
lat_irregularity	Irregularity level for latitude-band spacing. Must lie in [0, 1].
phase_twist	Finite angular phase offset used to desynchronize neighboring latitude bands.
surface	Sphere geometry family. One of "standard", "ellipsoid", or "wavy".
radius	Positive baseline radius.
amplitude	Finite deformation amplitude. For "ellipsoid", positive values make the shape oblate and negative values make it prolate.
freq_theta	Positive longitudinal frequency used only when surface = "wavy".
freq_lat	Positive latitudinal frequency used only when surface = "wavy".
twist	Finite longitude twist applied smoothly by latitude.
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Value

irregular.sphere.surface.embedding() returns an n x 3 numeric matrix with columns x, y, and z.

irregular.sphere.surface.graph() returns a list with components:

- edges: the irregular sphere edges,
- n: number of vertices,
- edge_weights: induced positive edge lengths,
- coords_surface: the 3D embedding,
- coords_param: the irregular angular parameter coordinates,
- weight_scale: the normalization constant applied to the raw edge lengths,
- family: always "irregular.sphere",
- surface: the chosen surface name,
- bands: number of non-pole latitude bands,
- band_sizes: sample counts in each latitude band,
- label: a human-readable family label.

irregular_torus_surface_helpers *Weighted irregular torus surface helpers*

Description

Convenience helpers that build a torus from deterministically irregular major-cycle rings with varying sample counts, stitch adjacent rings into a locally triangulated toroidal graph, and then realize that graph in $\mathbb{R}^3$ using either a standard, pinched, or wavy tube geometry.

Usage

```
irregular.torus.surface.embedding(
  major_rings = 8,
  tube_count = 16,
  count_irregularity = 0.2,
  major_irregularity = 0.25,
  phase_twist = 0.35,
  surface = c("standard", "pinched", "wavy"),
  major_radius = 2,
  minor_radius = 0.75,
  amplitude = 0.2,
  freq_major = 2,
  freq_minor = 1,
  twist = 0.25
)

irregular.torus.surface.graph(
  major_rings = 8,
  tube_count = 16,
  count_irregularity = 0.2,
  major_irregularity = 0.25,
  phase_twist = 0.35,
  surface = c("standard", "pinched", "wavy"),
  major_radius = 2,
  minor_radius = 0.75,
  amplitude = 0.2,
  freq_major = 2,
  freq_minor = 1,
  twist = 0.25,
  normalize = c("median", "mean", "none")
)
```

Arguments

major_rings	Number of cyclic major rings around the torus.
tube_count	Approximate number of vertices around each minor cycle.

count_irregularity
: Irregularity level for the per-ring sample counts. Must lie in [0, 1).

major_irregularity
: Irregularity level for the major-angle ring spacing. Must lie in [0, 1].

phase_twist
: Finite phase offset used to desynchronize neighboring minor cycles.

surface
: Torus geometry family. One of "standard", "pinched", or "wavy".

major_radius
: Positive distance from the torus center to the center of the tube.

minor_radius
: Positive baseline radius of the torus tube.

amplitude
: Finite deformation amplitude.

freq_major
: Positive angular frequency around the major cycle, used by "wavy" and the periodic twist.

freq_minor
: Positive angular frequency around the minor cycle, used by "wavy".

twist
: Finite phase twist applied periodically to the minor angle as a function of the major angle.

normalize
: Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Value

irregular.torus.surface.embedding() returns an n x 3 numeric matrix with columns x, y, and z.

irregular.torus.surface.graph() returns a list with components:

- edges: the irregular torus edges,
- n: number of vertices,
- edge_weights: induced positive edge lengths,
- coords_surface: the 3D embedding,
- coords_param: the irregular angular parameter coordinates,
- weight_scale: the normalization constant applied to the raw edge lengths,
- family: always "irregular.torus",
- surface: the chosen surface name,
- major_rings: number of major-cycle rings,
- ring_sizes: sample counts around each ring,
- label: a human-readable family label.

kary_tree_weighted_graph_helpers
Intrinsically weighted k-ary tree helpers

Description

Convenience helpers that keep the exact `edges.kary.tree()` topology but assign edge lengths directly from combinatorial tree metadata rather than from an ambient Euclidean embedding. Each edge weight is constructed as

$$\text{base\_length} \times \text{depth\_factor}(\text{child depth}) \times \text{branch\_factor}(\text{child slot})$$

so the user can control tapering with depth separately from child-slot asymmetry.

Usage

```
kary.tree.weighted.graph(
  k = 2,
  depth = 2,
  base_length = 1,
  depth_rule = c("geometric", "constant", "custom"),
  depth_decay = 0.85,
  depth_factors = NULL,
  branch_rule = c("linear", "uniform", "custom"),
  branch_spread = 0.3,
  branch_factors = NULL,
  normalize = c("median", "mean", "none")
)
```

Arguments

`k`	Branching factor. Must be at least `1`.
`depth`	Number of levels below the root. May be `0`.
`base_length`	Positive global edge-length multiplier before normalization.
`depth_rule`	Rule used to build the per-depth multipliers. One of `"geometric"`, `"constant"`, or `"custom"`.
`depth_decay`	Positive decay factor used when `depth_rule = "geometric"`.
`depth_factors`	Positive custom depth multipliers. Used only when `depth_rule = "custom"`. Must have length `1` or `depth`.
`branch_rule`	Rule used to build the per-child-slot multipliers. One of `"linear"`, `"uniform"`, or `"custom"`.
`branch_spread`	Non-negative spread used when `branch_rule = "linear"`.
`branch_factors`	Positive custom branch multipliers. Used only when `branch_rule = "custom"`. Must have length `1` or `k`.
`normalize`	Normalization applied to the raw intrinsic edge lengths. One of `"median"`, `"mean"`, or `"none"`.

Details

The returned object is intended as an intrinsic weighted-tree benchmark family. It exposes the same edge matrix as `edges.kary.tree()` together with edge weights, vertex depths, parent indices, and an edge table that records the depth and branch slot associated with each tree edge.

Value

A list with components:

- `edges`: the k-ary tree edges,
- `n`: number of vertices,
- `edge_weights`: normalized positive intrinsic edge lengths,
- `weight_scale`: normalization constant applied to the raw edge lengths,
- `vertex_depth`: depth of each vertex, with the root at depth `0`,
- `parent`: parent vertex index for each vertex, with the root parent recorded as `0`,
- `edge_table`: data frame containing `parent`, `child`, `parent_depth`, `child_depth`, `branch_index`, `raw_weight`, and normalized `edge_weight`,
- `depth_factors`: the applied per-depth multipliers,
- `branch_factors`: the applied per-child-slot multipliers,
- `family`: always `"kary.tree.weighted"`,
- `k`: the branching factor,
- `depth`: the requested tree depth,
- `label`: a human-readable family label.

`kernel.gram.gkk` *Optimize a kernel Gram-gKK layout*

Description

‘kernel.gram.gkk()‘ extends edge-only gKK with a local Riemannian star penalty. The off-diagonal Gram term preserves target inner products between pairs of incident edge directions:

$$\frac{\lambda_{\mathrm{gram}}}{2} \sum_{u} \sum_{v<v' \in N(u)} \omega_u(v,v') \left(\langle z_v - z_u, z_{v'} - z_u \rangle - s^2 w_{uv} w_{uv'} \cos \alpha_u(v,v') \right)^2 .$$

The star weights are built by [graph.riemannian.star.structure()], usually with an antipodal kernel controlled by ‘angle.power‘.

Usage

```
kernel.gram.gkk(
  coords = NULL,
  prepared = NULL,
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  X = NULL,
  star = NULL,
  dim = 2L,
  init = c("metric_mds", "random"),
  angle.power = 4,
  reliability = c("length.balance", "none"),
  min.angle.weight = 0,
  star.quantile = 0,
  lambda.edge = 1,
  lambda.gram = 1,
  stiffness_method = c("density", "uniform", "distance_power"),
  stiffness_transform = c("identity", "sqrt", "log"),
  density_mix = 1,
  bandwidth = NULL,
  density_n = 512L,
  distance_power = 0,
  stiffness_floor = 0,
  stiffness_ceiling = Inf,
  scale_mode = c("profiled", "identity", "user"),
  scale = NULL,
  max_iter = 50L,
  initial_step = 0.1,
  step_shrink = 0.5,
  armijo_factor = 1e-04,
  grad_tol = 1e-08,
  min_step = 1e-08,
  edge_length_epsilon = 1e-08,
  distance_floor = 1e-08,
  recenter = TRUE,
  return_trace = TRUE,
  diagnostics = TRUE,
  seed = 1L,
  engine = c("cpp", "R")
)
```

Arguments

coords	Optional starting coordinates. If omitted, ‘init‘ is used.
prepared	Optional object returned by [prepare.edge.kk()], [prepare.graph.geodesic.mds()]

or [prepare.geodesic.kk()]. Edge-only objects from [prepare.edge.kk()] report only edge diagnostics; all-pairs GMDS path and chord diagnostics are unavailable.

`edges` Two-column edge matrix used when ‘prepared‘ is omitted.

`n` Number of vertices used when ‘prepared‘ is omitted.

`adj_list` Optional adjacency list used when ‘prepared‘ is omitted.

`weight_list` Optional edge-weight list parallel to ‘adj_list‘.

`edge_weights` Optional positive edge weights parallel to ‘edges‘.

`X` Optional ambient/source coordinates used to build ‘star‘ when ‘star‘ is omitted.

`star` Optional object from [graph.riemannian.star.structure()].

`dim` Target embedding dimension.

`init` Starting layout used when ‘coords‘ is omitted. ‘"metric_mds"‘ uses ordinary metric MDS from an all-pairs prepared object and ‘"random"‘ uses centered Gaussian coordinates.

`angle.power`, `reliability`, `min.angle.weight`
Passed to [graph.riemannian.star.structure()] when ‘star‘ is omitted.

`star.quantile` Optional quantile filter passed to [graph.riemannian.star.structure()] when ‘star‘ is omitted.

`lambda.edge`, `lambda.gram`
Non-negative weights for the diagonal edge-length and off-diagonal Gram penalties.

`stiffness_method`, `stiffness_transform`, `density_mix`, `bandwidth`, `density_n`
Parameters passed to [edge.length.density.stiffness()] to construct edge-length stiffnesses for the diagonal edge term.

`distance_power`, `stiffness_floor`, `stiffness_ceiling`
Additional stiffness constructor parameters.

`scale_mode` Scale policy for edge targets. ‘"profiled"‘ analytically refits ‘s‘ at every state evaluation, ‘"identity"‘ fixes ‘s = 1‘, ‘"fixed_initial"‘ fits ‘s‘ once at the first continuation stage, and ‘"user"‘ uses ‘scale‘.

`scale` User scale for ‘scale_mode = "user"‘.

`max_iter` Maximum iterations per continuation stage.

`initial_step`, `step_shrink`, `armijo_factor`, `grad_tol`, `min_step`
Line-search controls.

`edge_length_epsilon`
Small stabilizer for fixed-path embedded lengths.

`distance_floor` Positive floor for relative residuals.

`recenter` If ‘TRUE‘, recenter the layout after accepted steps.

`return_trace` If ‘TRUE‘, keep per-iteration trace rows and coordinate frames.

`diagnostics` If ‘TRUE‘, attach the common GMDS diagnostic panel.

`seed` Random seed used only for ‘init = "random"‘.

`engine` Optimizer engine. ‘"cpp"‘ uses the Rcpp backend for the edge-stress loop; ‘"R"‘ uses the reference implementation.

Value

A ‘"grip_gmds_layout"‘ object with method ‘"kernel_gram_gkk"‘.

`landmark.geodesic.kk` *Optimize a layout under the landmark geodesic KK energy*

Description

`landmark.geodesic.kk()` applies a deterministic warm-started gradient-descent polish under the sparse landmark geodesic KK energy. It starts from an existing layout and refines it, rather than replacing the full multiscale GRIP refinement pipeline.

Usage

```
landmark.geodesic.kk(
  coords,
  prepared = NULL,
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  local_nbrs = 20L,
  landmark_count = 8L,
  max_iter = 16L,
  stiffness = 1,
  distance_floor = 1e-08,
  edge_length_epsilon = 1e-08,
  initial_step = 1,
  step_shrink = 0.5,
  armijo_factor = 1e-04,
  grad_tol = 1e-08,
  min_step = 1e-08,
  recenter = TRUE,
  return_trace = FALSE
)
```

Arguments

`coords`	Numeric coordinate matrix with 2 or 3 columns.
`prepared`	Optional object returned by `prepare.landmark.geodesic.kk()`.
`edges`	Two-column integer matrix of edges (1-based vertex ids).
`n`	Number of vertices.
`adj_list`	Adjacency list (1-based) for an undirected graph.
`weight_list`	Optional parallel list of positive edge weights.

edge_weights	Optional positive edge-weight vector parallel to edges.
local_nbrs	Number of nearest graph-metric neighbors retained per vertex when prepared is not supplied.
landmark_count	Number of farthest-point landmarks retained per vertex when prepared is not supplied.
max_iter	Maximum number of gradient-descent iterations.
stiffness	Global stiffness constant \(K\).
distance_floor	Small positive floor used in k_ij = K / max(g_ij, distance_floor)^2.
edge_length_epsilon	Small positive stabilizer added inside each embedded edge length.
initial_step	Initial line-search step size.
step_shrink	Multiplicative shrink factor in ‘(0, 1)‘ for backtracking.
armijo_factor	Non-negative Armijo decrease constant.
grad_tol	Non-negative stopping tolerance on the gradient norm.
min_step	Positive minimum accepted line-search step before giving up.
recenter	If TRUE, recenter the layout to zero mean after each accepted step.
return_trace	If TRUE, include per-iteration diagnostics and the accepted intermediate coordinate frames.

Details

The implementation fits the LGKK target scale L0 once from the starting layout and then optimizes against those fixed target path lengths. That keeps the gradient simple and makes the resulting line search robust for an initial implementation.

Value

A list with coords, trace, frames, prepared, and score.

---

legacy.grip *Compute the legacy GRIP layout*

Description

This function preserves the original local-force wrapper and historical default values that were previously exposed as grip(). Use it for backwards-compatible comparisons or when you explicitly want the pre-global-repulsion behavior.

Usage

```
legacy.grip(
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  dim = 3,
  placement = c("barycenter", "circle"),
  preset = NULL,
  rounds = 20,
  final_rounds = 25,
  num_init = 36,
  num_nbrs = 10,
  r = 0.15,
  s = 3,
  repulsion_factor = 1,
  tinit_factor = 6,
  seed = 6,
  disconnected = c("components", "error")
)
```

Arguments

edges	Two-column integer matrix of edges (1-based vertex ids).
n	Number of vertices.
adj_list	Adjacency list (1-based) for undirected graphs.
weight_list	Optional parallel list of edge weights (edge lengths). If NULL, all edges are treated as weight 1. All weights must be finite and strictly positive.
edge_weights	Optional vector of edge weights for edges. All weights must be finite and strictly positive.
dim	Layout dimension (2 or 3). Default is 3.
placement	Initial placement strategy. "circle" is only used for 2D.
preset	Optional tuning preset. NULL uses the historical defaults. "carpet" applies a preset tuned for Sierpinski-carpet-like graphs and validated on carpet levels 3 and 4. "mesh" applies a preset tuned for rectangular lattice graphs and validated on 8x8 and 12x12 mesh layouts. "torus" applies a preset tuned for 3D torus layouts and validated on torus sizes from 8x8 through 20x20. "tree" applies a preset tuned for symmetric force-directed layouts of tree-like graphs and validated on binary trees of depths 5 and 6. Presets only fill in tuning arguments that you did not supply explicitly.
rounds	Initial rounds for refinement.
final_rounds	Final rounds for refinement.
num_init	Number of initial vertices in the coarsest level.

num_nbrs	Maximum number of graph-distance neighbors retained for local refinement at each filtration level.
r	Main local temperature adaptation rate in [0, 1].
s	Non-negative boost factor applied when successive displacements have a consistent direction.
repulsion_factor	Non-negative multiplier applied to GRIP's finest-level repulsive force scale. 1 keeps the historical repulsion strength; 0 disables that repulsive term.
tinit_factor	Initial temperature factor.
seed	Optional RNG seed for reproducibility. If NULL, uses current time.
disconnected	How to handle disconnected graphs: "components" (default) lays out each connected component separately and packs them into one coordinate matrix; "error" stops with an error.

Value

A numeric matrix with 'n' rows and 'dim' columns.

References

Gajer, P. and Kobourov, S.G. (2002). GRIP: Graph dRawing with Intelligent Placement. *Journal of Graph Algorithms and Applications*, 6(3), 203–224. doi:10.7155/jgaa.00052.

Gajer, P., Goodrich, M.T. and Kobourov, S.G. (2004). A multi-dimensional approach to force-directed layouts of large graphs. *Computational Geometry*, 29(1), 3–18. doi:10.1016/j.comgeo.2004.03.014.

Examples

```
edges <- cbind(1:5, 2:6)
coords <- legacy.grip(edges, n = 6, dim = 2,
                              placement = "barycenter",
                              rounds = 5, final_rounds = 5,
                              num_init = 3, num_nbrs = 4,
                              seed = 1)
round(coords, 3)
```

---

mask_pattern_helpers *Mask pattern helpers for recursive grid families*

Description

Convenience constructors for connected $k \times k$ keep-masks that can be passed to edges.recursive.mask.grid() or recursive.mask.grid.surface.graph() to build generalized mask carpets. These helpers are meant to provide explicit benchmark motifs such as cross, border, corner, and asymmetric-hole patterns.

Usage

```
mask.cross(k = 5, arm_width = 1)

mask.border(k = 5, thickness = 1)

mask.corner(
  k = 5,
  width = 2,
  corner = c("top_left", "top_right", "bottom_left", "bottom_right")
)

mask.asymmetric.holes(k = 5, hole_size = 1)
```

Arguments

k	Mask side length.
arm_width	Width of the retained cross arms.
thickness	Border thickness for mask.border().
width	Side length of the retained corner block for mask.corner().
corner	Which corner to retain in mask.corner().
hole_size	Side length of each removed interior hole block for mask.asymmetric.holes().

Details

The returned masks use standard matrix display orientation: rows run from top to bottom and columns run from left to right.

Value

A logical $k \times k$ keep-mask.

---

menger_sponge_surface_helpers

Weighted Menger sponge surface helpers

Description

Convenience wrappers around recursive.cube.mask.surface.*() using the classic $3 \times 3 \times 3$ Menger-sponge keep-mask. These helpers provide a named cubical 3D benchmark family whose vertices are occupied subcubes and whose edges connect face-adjacent occupied cells.

Usage

```
menger.sponge.surface.embedding(
  level = 2,
  surface = c("standard", "bulged", "twisted", "wavy"),
  amplitude = 0.2,
  freq = 2,
  twist = 0.6,
  x_scale = 1,
  y_scale = 1,
  z_scale = 1
)

menger.sponge.surface.graph(
  level = 2,
  surface = c("standard", "bulged", "twisted", "wavy"),
  amplitude = 0.2,
  freq = 2,
  twist = 0.6,
  x_scale = 1,
  y_scale = 1,
  z_scale = 1,
  normalize = c("median", "mean", "none")
)
```

Arguments

level	Recursion depth. Must be at least 1.
surface	Cube-mask geometry family. One of "standard", "bulged", "twisted", or "wavy".
amplitude	Finite deformation amplitude.
freq	Positive modulation frequency used only when surface = "wavy".
twist	Finite twist strength used only when surface = "twisted".
x_scale	Positive horizontal scaling applied to the canonical cube coordinates.
y_scale	Positive vertical scaling applied to the canonical cube coordinates.
z_scale	Positive depth scaling applied to the canonical cube coordinates.
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Value

menger.sponge.surface.embedding() returns an n x 3 numeric matrix with columns x, y, and z, where n = 20^level.

menger.sponge.surface.graph() returns the same components as recursive.cube.mask.surface.graph(), with family set to "menger.sponge" and a family-specific class and label.

mesh_surface_helpers *Weighted mesh surface helpers*

Description

Convenience helpers that lift a rectangular mesh parameter grid into $\mathbb{R}^3$ and use the induced Euclidean edge lengths as positive graph weights. These helpers are intended for benchmark families where the topology is a plain mesh but the intended metric comes from a curved ambient geometry.

Usage

```
mesh.surface.embedding(
  h,
  w = h,
  surface = c("saddle", "paraboloid", "ripple"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  x_scale = 1,
  y_scale = 1
)

mesh.surface.graph(
  h,
  w = h,
  surface = c("saddle", "paraboloid", "ripple"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  x_scale = 1,
  y_scale = 1,
  connectivity = c("orthogonal", "diagonal"),
  normalize = c("median", "mean", "none")
)
```

Arguments

h	Number of rows.
w	Number of columns. Defaults to h.
surface	Surface family used for the lift. One of "saddle", "paraboloid", or "ripple".
amplitude	Finite numeric amplitude controlling the non-flat displacement.
freq_u	Positive ripple frequency in the horizontal parameter direction. Used only when surface = "ripple".
freq_v	Positive ripple frequency in the vertical parameter direction. Used only when surface = "ripple".

x_scale	Positive horizontal scaling of the parameter domain.
y_scale	Positive vertical scaling of the parameter domain.
connectivity	Occupied-mesh neighborhood rule passed to edges.occupied.mesh().
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Details

'mesh.surface.embedding()' returns the 3D coordinates of the lifted grid. 'mesh.surface.graph()' returns a reusable weighted-graph bundle containing the mesh edges, induced edge weights, the 3D surface coordinates, and the 2D parameter coordinates.

Value

mesh.surface.embedding() returns an n x 3 numeric matrix with columns x, y, and z.

mesh.surface.graph() returns a list with components:

- edges: the undirected mesh edges,
- n: number of vertices,
- edge_weights: induced positive edge lengths,
- coords_surface: the 3D surface embedding,
- coords_param: the 2D parameter-grid coordinates,
- weight_scale: the normalization constant applied to the raw edge lengths,
- family: always "mesh",
- surface: the chosen surface name,
- connectivity: the chosen mesh connectivity rule,
- label: a human-readable family label.

---

metric.mds *Metric-MDS baseline layout for GMDS experiments*

Description

'metric.mds()' computes the current metric-MDS baseline using 'stats::cmdscale()' on the graph shortest-path distance matrix and returns a common '"grip_gmds_layout"' result.

Usage

```
metric.mds(
  prepared = NULL,
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  dim = 2L,
  add = FALSE,
  eig = TRUE,
  diagnostics = TRUE,
  scale_mode = c("profiled", "identity", "user"),
  distance_floor = 1e-08,
  edge_length_epsilon = 1e-08,
  band_quantiles = c(1/3, 2/3)
)
```

Arguments

prepared	Optional all-pairs prepared object returned by [prepare.graph.geodesic.mds()] or [prepare.geodesic.kk()]. Edge-only objects from [prepare.edge.kk()] are intentionally rejected because metric MDS requires a graph distance matrix.
edges	Two-column edge matrix used when ‘prepared‘ is omitted.
n	Number of vertices used when ‘prepared‘ is omitted.
adj_list	Optional adjacency list used when ‘prepared‘ is omitted.
weight_list	Optional edge-weight list parallel to ‘adj_list‘.
edge_weights	Optional positive edge weights parallel to ‘edges‘.
dim	Target embedding dimension. Values greater than 3 are supported for GMDS and edge-KK workflows.
add, eig	Passed to ‘stats::cmdscale()‘.
diagnostics	If ‘TRUE‘, attach the common GMDS diagnostic panel. If ‘FALSE‘ and raw graph inputs are supplied, ‘metric.mds()‘ uses a distance-matrix-only preparation path and does not build the full all-pairs path cache.
scale_mode	Scale policy for edge, path, and chord targets: ‘"profiled"‘ fits one scalar for each diagnostic family, ‘"identity"‘ uses scale one, and ‘"user"‘ uses the corresponding supplied scale.
distance_floor	Positive floor for relative residuals.
edge_length_epsilon	Small stabilizer for fixed-path embedded lengths.
band_quantiles	Two quantiles splitting graph distances into short, mid, and long bands.

Value

A ‘"grip_gmds_layout"‘ object.

`misf.geodesic.kk` *Optimize an embedding with the MISF-based geodesic-KK pipeline*

Description

'misf.geodesic.kk()' runs a multiscale MISF-based geodesic-KK pipeline. It solves the top MISF graph with either full GKK or sparse LGKK, inserts lower-level vertices with the existing MISF placement helpers, refines each active MISF level under a GKK/LGKK objective, and finishes with a final full-graph polish.

Usage

```
misf.geodesic.kk(
  prepared = NULL,
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  tie_mode = NULL,
  num_init = 24L,
  num_nbrs = 20L,
  dim = NULL,
  top_level_pair_mode = NULL,
  top_level_full_limit = NULL,
  top_level_local_nbrs = NULL,
  top_level_landmark_count = NULL,
  top_level_restarts = NULL,
  top_level_max_iter = NULL,
  top_level_init = NULL,
  insertion_anchor_policy = NULL,
  insertion_anchor_count = NULL,
  insertion_anchor_weight_mode = NULL,
  insertion_max_iter = NULL,
  insertion_mode = NULL,
  insertion_layout_k = NULL,
  insertion_weighted_preset = NULL,
  insertion_grip_args = NULL,
  insertion_weighted_args = NULL,
  insertion_fr_niter = NULL,
  refinement_pair_mode = NULL,
  refinement_full_limit = NULL,
  refinement_local_nbrs = NULL,
  refinement_landmark_count = NULL,
  refinement_anchor_weight = NULL,
  refinement_anchor_weight_end = NULL,
  refinement_continuation = NULL,
```

```
  refinement_max_iter = NULL,
  final_pair_mode = NULL,
  final_full_limit = NULL,
  final_local_nbrs = NULL,
  final_landmark_count = NULL,
  final_max_iter = NULL,
  stiffness = 1,
  distance_floor = 1e-08,
  edge_length_epsilon = 1e-08,
  initial_step = 1,
  step_shrink = 0.5,
  armijo_factor = 1e-04,
  grad_tol = 1e-08,
  min_step = 1e-08,
  recenter = TRUE,
  return_trace = FALSE,
  return_frames = FALSE,
  seed = 6L
)
```

Arguments

prepared	Optional prepared object. This can be either a full geodesic-KK prepared object or a MISF-GKK prepared object.
edges	Two-column integer matrix of edges (1-based vertex ids).
n	Number of vertices. If omitted with 'adj_list', defaults to 'length(adj_list)'. If omitted with 'edges', defaults to 'max(edges)'.
adj_list	Adjacency list (1-based) for an undirected graph.
weight_list	Optional parallel list of positive edge weights.
edge_weights	Optional positive edge-weight vector parallel to 'edges'.
tie_mode	Shortest-path aggregation mode inherited from [prepare.geodesic.kk()].
num_init	Target top-level active-set size passed to 'build.misf'.
num_nbrs	Per-level local-neighborhood schedule metadata passed to 'build.misf'.
dim	Target embedding dimension ('2' or '3') for the multiscale solve.
top_level_pair_mode	Optional override for the top-level pair policy.
top_level_full_limit	Optional override for the top-level exact/sparse threshold.
top_level_local_nbrs	Optional override for the sparse top-level local neighborhood size.
top_level_landmark_count	Optional override for the sparse top-level landmark count.
top_level_restarts	Number of top-level restarts used by the coarse MISF-GKK solve.

`top_level_max_iter`
: Top-level iteration budget.

`top_level_init`
: Optional override for the top-level initializer.

`insertion_anchor_policy`
: Optional insertion anchor policy.

`insertion_anchor_count`
: Optional insertion anchor count.

`insertion_anchor_weight_mode`
: Optional insertion anchor weighting mode.

`insertion_max_iter`
: Optional per-vertex insertion iteration budget.

`insertion_mode`
: Optional insertion warm-start mode.

`insertion_layout_k`
: Optional active-level layout graph size used by layout-based insertion modes.

`insertion_weighted_preset`
: Optional preset passed to weighted GRIP insertion.

`insertion_grip_args`
: Optional named list of extra arguments passed to combinatorial GRIP insertion.

`insertion_weighted_args`
: Optional named list of extra arguments for weighted GRIP insertion.

`insertion_fr_niter`
: Optional FR iteration budget for FR-based insertion.

`refinement_pair_mode`
: Optional active-level pair policy.

`refinement_full_limit`
: Optional active-level exact/sparse threshold.

`refinement_local_nbrs`
: Optional active-level sparse local neighborhood size.

`refinement_landmark_count`
: Optional active-level sparse landmark count.

`refinement_anchor_weight`
: Optional initial anchor weight used to pin inactive and coarser-level vertices during KK refinement.

`refinement_anchor_weight_end`
: Optional final anchor weight used at the end of the active-level continuation schedule.

`refinement_continuation`
: Optional continuation schedule used for the active-level anchor penalty.

`refinement_max_iter`
: Optional active-level refinement iteration budget.

`final_pair_mode`
: Optional final full-graph pair policy.

`final_full_limit`
: Optional final full-graph exact/sparse threshold.

`final_local_nbrs`
: Optional final full-graph sparse local neighborhood size.

`final_landmark_count`
: Optional final full-graph sparse landmark count.

`final_max_iter` Optional final polish iteration budget.

`stiffness` Global stiffness constant \(K\).

`distance_floor` Small positive floor used in `k_ij = K / max(g_ij, distance_floor)^2`.

`edge_length_epsilon`
: Small positive stabilizer added inside each embedded edge length.

`initial_step` Initial line-search step size.

`step_shrink` Multiplicative shrink factor in ‘(0, 1)‘ for backtracking.

`armijo_factor` Non-negative Armijo decrease constant.

`grad_tol` Non-negative stopping tolerance on the gradient norm.

`min_step` Positive minimum accepted line-search step before giving up.

`recenter` If ‘TRUE‘, recenter accepted proposals to zero mean after each accepted step.

`return_trace` If ‘TRUE‘, retain detailed stage traces.

`return_frames` If ‘TRUE‘, retain intermediate coordinate frames.

`seed` Optional integer seed reused for MISF extraction.

Value

A list with ‘coords‘, ‘prepared‘, the per-stage multiscale results, the stage trace, optional trace/frame details, timing, and the final MISF score summary.

`occupied_mesh_surface_helpers`
Occupied-mesh and perforated-grid helpers

Description

`edges.occupied.mesh()` builds a finite occupied-grid graph on the retained cells of a rectangular occupancy matrix. The corresponding surface helpers lift those occupied cells into $\mathbb{R}^3$ using the same `"saddle"`, `"paraboloid"`, and `"ripple"` families used for regular meshes.

Usage

```
occupied.mesh.surface.embedding(
  keep,
  surface = c("saddle", "paraboloid", "ripple"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  x_scale = 1,
```

```
  y_scale = 1
)

occupied.mesh.surface.graph(
  keep,
  surface = c("saddle", "paraboloid", "ripple"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  x_scale = 1,
  y_scale = 1,
  connectivity = c("orthogonal", "diagonal"),
  normalize = c("median", "mean", "none")
)
```

Arguments

keep	Logical or numeric occupancy matrix. Non-zero entries are kept.
surface	Surface family used for the lift. One of "saddle", "paraboloid", or "ripple".
amplitude	Finite numeric amplitude controlling the non-flat displacement.
freq_u	Positive ripple frequency in the horizontal parameter direction. Used only when surface = "ripple".
freq_v	Positive ripple frequency in the vertical parameter direction. Used only when surface = "ripple".
x_scale	Positive horizontal scaling of the parameter domain.
y_scale	Positive vertical scaling of the parameter domain.
connectivity	Mesh neighborhood rule passed to edges.mesh().
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Value

occupied.mesh.surface.embedding() returns an n x 3 numeric matrix with columns x, y, and z.

occupied.mesh.surface.graph() returns a list with components:

- edges: the occupied-cell mesh edges,
- n: number of vertices,
- edge_weights: induced positive edge lengths,
- coords_surface: the 3D surface embedding,
- coords_param: the 2D parameter coordinates,
- weight_scale: the normalization constant applied to the raw edge lengths,
- family: always "occupied.mesh",
- surface: the chosen surface name,
- connectivity: the chosen occupied-mesh connectivity rule,
- keep: the logical occupancy matrix,
- label: a human-readable family label.

params.from.summary *Extract layout parameters from a comparison summary row*

Description

Converts one row of the summary data frame returned by `compare.layouts` into a named list of parameters suitable for passing to `grip` or back into a `compare.layouts(candidates = ...)` call.

Usage

```
params.from.summary(summary.row)
```

Arguments

`summary.row`	A one-row data frame (or the first row is used) from the `$summary` element of `compare.layouts()`.

Value

A named list with elements `placement`, `rounds`, `final_rounds`, `num_init`, `num_nbrs`, `r`, `s`, `repulsion_factor`, `tinit_factor`, and optionally `preset` (included only when the summary row records a non-empty preset).

Examples

```
edges <- edges.path(5)
cmp <- compare.layouts(edges, n = 5, dim = 2,
                           candidates = "default",
                           seeds = 1L)
params <- params.from.summary(cmp$summary[1, ])
coords <- do.call(
  grip,
  c(list(edges = edges, n = 5, dim = 2, seed = 42L), params)
)
round(coords, 2)
```

perforated_grid_helpers *Deterministic perforated-grid occupancy helpers*

Description

Convenience constructors for finite occupied grids with non-random holes or channels. These matrices can be passed to `edges.occupied.mesh()` or `occupied.mesh.surface.graph()`.

Usage

```
keep.periodic.holes(
  h,
  w = h,
  hole_period = 4,
  hole_height = 1,
  hole_width = hole_height,
  row_offset = 2,
  col_offset = 2
)

keep.staggered.windows(
  h,
  w = h,
  window_height = 1,
  window_width = 2,
  row_period = 4,
  col_period = 5,
  row_offset = 2,
  col_offset = 2
)

keep.slit.channels(
  h,
  w = h,
  orientation = c("vertical", "horizontal"),
  slit_period = 5,
  slit_width = 1,
  bridge_spacing = 4,
  bridge_size = 1,
  offset = 2
)

keep.asymmetric.notches(h, w = h, notch_depth = 3, notch_width = 2)
```

Arguments

h	Number of rows.
w	Number of columns. Defaults to h.
hole_period	Spacing between periodic holes.
hole_height	Height of each removed rectangular hole.
hole_width	Width of each removed rectangular hole.
row_offset	Starting row index for the first removed block.
col_offset	Starting column index for the first removed block.
window_height	Height of each staggered removed window.
window_width	Width of each staggered removed window.

row_period	Vertical spacing between staggered window bands.
col_period	Horizontal spacing between staggered windows within a band.
orientation	Whether slit channels run "vertical" or "horizontal".
slit_period	Spacing between repeated slit channels.
slit_width	Width of each slit channel.
bridge_spacing	Spacing between preserved bridge segments.
bridge_size	Size of each preserved bridge segment along a slit.
offset	Starting row or column index for the first slit channel.
notch_depth	Depth of each asymmetric notch cut from a boundary.
notch_width	Width of the notched opening along that boundary.

Value

A logical occupancy matrix whose TRUE entries represent retained cells.

---

plot.layout *Quick plot for graph layouts*

Description

For 2D layouts, draws vertices and edges using base graphics. For 3D layouts, the behaviour depends on the projection argument:

"rgl" **(default)** Opens an interactive **rgl** scene when that package is installed. Falls back to "ortho" with a warning if **rgl** is not available.

"ortho" Projects the 3D coordinates to 2D with project.3d and draws them with base graphics, giving a static figure suitable for vignettes and non-interactive reports.

Usage

```
## S3 method for class 'layout'
plot(x, ..., edges = NULL,
  projection = c("rgl", "ortho"), azimuth = 35, elevation = 22,
  vertex.col = "black", edge.col = "gray70")
```

Arguments

x	Numeric coordinate matrix with at least two columns.
...	Additional parameters passed to the underlying plot() call for 2D layouts or the **rgl** 3D plotting call.
edges	Optional two-column matrix of edges (1-based vertex ids).
projection	Character string controlling how 3D layouts are displayed: "rgl" (default) for an interactive **rgl** widget, or "ortho" for a static orthographic projection via base graphics. Ignored for 2D layouts.

azimuth	Rotation around the vertical axis in degrees. Only used when projection = "ortho". Default 35.
elevation	Rotation around the horizontal axis in degrees. Only used when projection = "ortho". Default 22.
vertex.col	Colour(s) for the vertices. Recycled to match the number of vertices. Default "black".
edge.col	Colour for the edges. Default "gray70".

Value

NULL (called for side effects).

Examples

```
edges <- cbind(1:5, 2:6)
coords <- grip(edges, n = 6, dim = 2,
                      placement = "barycenter",
                      rounds = 5, final_rounds = 5,
                      num_init = 3, num_nbrs = 4,
                      seed = 1)
plot.layout(coords, edges, main = "Path graph", pch = 16, cex = 0.8)
```

---

porous_cube_surface_helpers

Weighted porous cube-mask surface helpers

Description

Convenience wrappers around recursive.cube.mask.surface.*() for three named porous cubical families: a periodic tunnel lattice, an asymmetric cavity family, and a deterministic channel network. Each family uses a reusable base keep-array and then recurses it in exactly the same way as the generic cube-mask helpers.

Usage

```
cube.periodic.tunnels.surface.embedding(
  level = 2,
  side = 5,
  tunnel_width = 1,
  tunnel_period = 2,
  tunnel_offset = 2,
  surface = c("standard", "bulged", "twisted", "wavy"),
  amplitude = 0.2,
  freq = 2,
  twist = 0.6,
  x_scale = 1,
  y_scale = 1,
```

```
  z_scale = 1
)

cube.periodic.tunnels.surface.graph(
  level = 2,
  side = 5,
  tunnel_width = 1,
  tunnel_period = 2,
  tunnel_offset = 2,
  surface = c("standard", "bulged", "twisted", "wavy"),
  amplitude = 0.2,
  freq = 2,
  twist = 0.6,
  x_scale = 1,
  y_scale = 1,
  z_scale = 1,
  normalize = c("median", "mean", "none")
)

cube.asymmetric.cavities.surface.embedding(
  level = 2,
  side = 5,
  cavity_size = 2,
  pocket_size = max(1L, cavity_size - 1L),
  surface = c("standard", "bulged", "twisted", "wavy"),
  amplitude = 0.2,
  freq = 2,
  twist = 0.6,
  x_scale = 1,
  y_scale = 1,
  z_scale = 1
)

cube.asymmetric.cavities.surface.graph(
  level = 2,
  side = 5,
  cavity_size = 2,
  pocket_size = max(1L, cavity_size - 1L),
  surface = c("standard", "bulged", "twisted", "wavy"),
  amplitude = 0.2,
  freq = 2,
  twist = 0.6,
  x_scale = 1,
  y_scale = 1,
  z_scale = 1,
  normalize = c("median", "mean", "none")
)
```

```
cube.channel.network.surface.embedding(
  level = 2,
  side = 5,
  channel_width = 1,
  branch_offset = 2,
  surface = c("standard", "bulged", "twisted", "wavy"),
  amplitude = 0.2,
  freq = 2,
  twist = 0.6,
  x_scale = 1,
  y_scale = 1,
  z_scale = 1
)

cube.channel.network.surface.graph(
  level = 2,
  side = 5,
  channel_width = 1,
  branch_offset = 2,
  surface = c("standard", "bulged", "twisted", "wavy"),
  amplitude = 0.2,
  freq = 2,
  twist = 0.6,
  x_scale = 1,
  y_scale = 1,
  z_scale = 1,
  normalize = c("median", "mean", "none")
)
```

Arguments

`level`	Recursion depth. Must be at least `1`.
`side`	Side length of the base cubic keep-array.
`tunnel_width`	Width of each removed tunnel band.
`tunnel_period`	Spacing between successive tunnel bands.
`tunnel_offset`	Starting index of the first tunnel band.
`surface`	Cube-mask geometry family. One of `"standard"`, `"bulged"`, `"twisted"`, or `"wavy"`.
`amplitude`	Finite deformation amplitude.
`freq`	Positive modulation frequency used only when `surface = "wavy"`.
`twist`	Finite twist strength used only when `surface = "twisted"`.
`x_scale`	Positive horizontal scaling applied to the canonical cube coordinates.
`y_scale`	Positive vertical scaling applied to the canonical cube coordinates.
`z_scale`	Positive depth scaling applied to the canonical cube coordinates.
`normalize`	Normalization applied to the induced edge lengths. One of `"median"`, `"mean"`, or `"none"`.

cavity_size	Side length of the larger interior cavity block.
pocket_size	Side length of the smaller secondary cavity block.
channel_width	Width of each removed channel in the channel-network family.
branch_offset	Interior offset of the extra branch channel in the channel-network family.

Value

Each *.surface.embedding() helper returns an n x 3 numeric matrix with columns x, y, and z.

Each *.surface.graph() helper returns the same components as recursive.cube.mask.surface.graph(), with a family-specific family field, class, and label.

---

prepare.edge.kk *Prepare an edge-only graph for edge-KK repair*

Description

prepare.edge.kk() validates an undirected weighted graph and returns the lightweight prepared object used by edge.kk() when only graph-edge targets are needed. Unlike prepare.graph.geodesic.mds(), this helper does not compute all-pairs shortest paths, path caches, or a dense graph distance matrix.

Usage

```
prepare.edge.kk(
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL
)
```

Arguments

edges	Two-column integer matrix of edges (1-based vertex ids).
n	Number of vertices. If omitted with adj_list, defaults to length(adj_list). If omitted with edges, defaults to max(edges).
adj_list	Adjacency list (1-based) for an undirected graph.
weight_list	Optional parallel list of positive edge weights.
edge_weights	Optional positive edge-weight vector parallel to edges.

Details

Use this helper when a starting layout is already available, for example from grip(..., metric = "edge_length"), and the next step is scalable edge-KK local repair. Edge-only objects support edge-fidelity diagnostics through score.gmds(); all-pairs GMDS path and chord diagnostics are unavailable and are reported as NA.

Value

A lightweight prepared object of class "grip_edge_kk_prepared" layered on the common "grip_gmds_prepared" graph class. It contains canonical graph edges and edge targets but no all-pairs geodesic cache.

---

prepare.geodesic.kk *Prepare full geodesic KK data for repeated layout evaluation*

Description

prepare.geodesic.kk() builds the deterministic all-pairs shortest path cache needed to evaluate or optimize the full geodesic Kamada–Kawai objective repeatedly on the same connected graph.

Usage

```
prepare.geodesic.kk(
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  tie_mode = c("single", "average")
)
```

Arguments

edges	Two-column integer matrix of edges (1-based vertex ids).
n	Number of vertices. If omitted with adj_list, defaults to length(adj_list). If omitted with edges, defaults to max(edges).
adj_list	Adjacency list (1-based) for an undirected graph.
weight_list	Optional parallel list of positive edge weights.
edge_weights	Optional positive edge-weight vector parallel to edges.
tie_mode	Shortest-path aggregation mode. "single" uses one deterministic chosen shortest path per pair. "average" replaces each tied shortest-path family by the exact uniform average over all shortest paths between the pair.

Details

For each unordered vertex pair, the prepared object stores either one deterministic chosen graph shortest path (tie_mode = "single") or the exact uniform average over all tied shortest paths (tie_mode = "average"), together with the graph distance and the corresponding cached edge realization. This is the full all-pairs analogue of the sparse landmark cache used by prepare.landmark.geodesic.kk().

Value

A list with the all-pairs graph distances, chosen paths, and cached path-edge realizations. The object has class "grip_gkk_prepared".

`prepare.graph.geodesic.mds` *Prepare a graph-first geodesic-MDS path cache*

Description

`prepare.graph.geodesic.mds()` prepares the full all-pairs chosen geodesic cache for an arbitrary connected weighted graph. This is the graph-first entry point corresponding to the manuscript's definition of GMDS on a connected weighted graph together with a chosen geodesic family (G, Γ).

Usage

```
prepare.graph.geodesic.mds(
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  tie_mode = c("single", "average")
)
```

Arguments

`edges`	Two-column integer matrix of edges (1-based vertex ids).
`n`	Number of vertices. If omitted with `adj_list`, defaults to `length(adj_list)`. If omitted with `edges`, defaults to `max(edges)`.
`adj_list`	Adjacency list (1-based) for an undirected graph.
`weight_list`	Optional parallel list of positive edge weights.
`edge_weights`	Optional positive edge-weight vector parallel to `edges`.
`tie_mode`	Shortest-path aggregation mode. `"single"` uses one deterministic chosen shortest path per pair. `"average"` replaces each tied shortest-path family by the exact uniform average over all shortest paths between the pair.

Details

The graph can be supplied either as an edge list plus parallel weights or as an adjacency-list representation. The returned object stores the all-pairs graph distances, the chosen shortest-path family, and the flattened edge-path cache reused by the GMDS scorer and optimizer.

Value

A prepared object with class `"grip_gmds_prepared"` layered on top of the existing full geodesic path-cache structure.

prepare.landmark.geodesic.kk *Prepare sparse landmark-geodesic KK data for repeated layout evaluation*

Description

`prepare.landmark.geodesic.kk()` builds the deterministic shortest path trees, graph-distance cache, sparse local-plus-landmark pair set, and chosen path realizations needed to evaluate the landmark geodesic KK energy repeatedly on the same graph. This is intended as a reusable preparation step for experiments where many layouts of the same graph are compared.

Usage

```
prepare.landmark.geodesic.kk(
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  local_nbrs = 20L,
  landmark_count = 8L
)
```

Arguments

`edges`	Two-column integer matrix of edges (1-based vertex ids).
`n`	Number of vertices. If omitted with `adj_list`, defaults to `length(adj_list)`. If omitted with `edges`, defaults to `max(edges)`.
`adj_list`	Adjacency list (1-based) for an undirected graph.
`weight_list`	Optional parallel list of positive edge weights.
`edge_weights`	Optional positive edge-weight vector parallel to `edges`.
`local_nbrs`	Number of nearest graph-metric neighbors retained per vertex.
`landmark_count`	Number of farthest-point landmarks retained per vertex.

Details

The sparse set follows the implementation choice recorded in `landmark\_geodesic\_kk\_spec\_2026-03-30.tex`: each vertex contributes its `local_nbrs` nearest active vertices in graph distance and its `landmark_count` farthest-point landmarks, both selected deterministically.

Value

A list with the sparse pair set, graph distances, chosen paths, and other cached data. The object has class `"grip_lgkk_prepared"`.

prepare.misf.geodesic.kk *Prepare a MISF-based multiscale geodesic-KK object*

Description

'prepare.misf.geodesic.kk()' builds the prepared state for a MISF-based geodesic-KK pipeline by layering the maximal independent set filtration (MISF) on top of the existing full geodesic-KK cache. The prepared object stores the graph metadata, the coarsest admissible top-level graph, the exact and sparse top-level prepared caches, and the default controls used by the multiscale MISF-GKK optimizer.

Usage

```
prepare.misf.geodesic.kk(
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  tie_mode = c("single", "average"),
  num_init = 24L,
  num_nbrs = 20L,
  dim = 2L,
  top_level_mode = c("solve", "skip"),
  top_level_pair_mode = c("auto", "full", "landmark"),
  top_level_full_limit = 512L,
  top_level_local_nbrs = 20L,
  top_level_landmark_count = 8L,
  top_level_restarts = 8L,
  top_level_max_iter = 16L,
  top_level_init = c("geometric", "cmdscale", "random"),
  seed = 6L
)
```

Arguments

edges	Two-column integer matrix of edges (1-based vertex ids).
n	Number of vertices. If omitted with 'adj_list', defaults to 'length(adj_list)'. If omitted with 'edges', defaults to 'max(edges)'.
adj_list	Adjacency list (1-based) for an undirected graph.
weight_list	Optional parallel list of positive edge weights.
edge_weights	Optional positive edge-weight vector parallel to 'edges'.
tie_mode	Shortest-path aggregation mode inherited from [prepare.geodesic.kk()].
num_init	Target top-level active-set size passed to 'build.misf'.

`num_nbrs` Per-level local-neighborhood schedule metadata passed to ‘build.misf‘.

`dim` Target embedding dimension (‘2‘ or ‘3‘) for the multiscale solve.

`top_level_mode` Either ‘"solve"‘ or ‘"skip"‘. When set to ‘"solve"‘, the coarsest admissible MISF level is optimized immediately and stored in ‘prepared$top_level_fit‘.

`top_level_pair_mode`
Pair policy for the top MISF level: ‘"auto"‘, ‘"full"‘, or ‘"landmark"‘.

`top_level_full_limit`
Active-set size threshold used when ‘top_level_pair_mode = "auto"‘.

`top_level_local_nbrs`
Number of nearest graph-metric neighbors retained per vertex in the sparse top-level LGKK cache.

`top_level_landmark_count`
Number of farthest-point landmarks retained per vertex in the sparse top-level LGKK cache.

`top_level_restarts`
Number of top-level restarts used by the coarse MISF-GKK solve.

`top_level_max_iter`
Top-level iteration budget.

`top_level_init` Top-level initializer (‘"geometric"‘, ‘"cmdscale"‘, or ‘"random"‘).

`seed` Optional integer seed reused for MISF extraction.

Value

An object of class ‘"grip_misf_gkk_prepared"‘ layered on top of the full geodesic-KK prepared structure, with added MISF metadata, top-level coarse-graph caches, and stored default controls for the MISF-GKK optimizer.

Examples

```
edges <- edges.mesh(4, 4)
prepared <- prepare.misf.geodesic.kk(
  edges = edges,
  n = 16,
  tie_mode = "average",
  num_init = 4,
  top_level_mode = "skip",
  seed = 1
)
prepared$top_level_vertices
```

project.3d *Project 3D coordinates to 2D for static plotting*

Description

Applies a rotation defined by azimuth and elevation angles and returns the first two columns of the rotated matrix.

Usage

```
project.3d(coords, azimuth = 35, elevation = 22)
```

Arguments

coords	Numeric matrix with at least 3 columns (only the first 3 are used).
azimuth	Rotation around the vertical axis in degrees. Default 35.
elevation	Rotation around the horizontal axis in degrees. Default 22.

Details

This is the projection used internally by plot.layout when projection = "ortho" and is exported so that users can pre-project coordinates for custom plotting.

Value

A two-column numeric matrix of projected (x, y) coordinates.

Examples

```
edges <- edges.torus(6, 10)
coords3d <- grip(edges, n = max(edges), dim = 3,
                         preset = "torus", seed = 1)
xy <- project.3d(coords3d)
plot(xy, asp = 1, pch = 16, cex = 0.5)
```

recursive_cube_mask_surface_helpers *Weighted recursive cube-mask surface helpers*

Description

Convenience helpers that recursively subdivide a cube into a regular $k \times k \times k$ array of subcubes according to a cubic keep-mask, retain the selected cells at each recursion step, and deform the resulting face-adjacency graph inside $\mathbb{R}^3$. This provides a generic cubical 3D family that includes Menger-sponge-style cell adjacency as a named special case.

Usage

```
recursive.cube.mask.surface.embedding(
  mask,
  level = 2,
  surface = c("standard", "bulged", "twisted", "wavy"),
  amplitude = 0.2,
  freq = 2,
  twist = 0.6,
  x_scale = 1,
  y_scale = 1,
  z_scale = 1
)

recursive.cube.mask.surface.graph(
  mask,
  level = 2,
  surface = c("standard", "bulged", "twisted", "wavy"),
  amplitude = 0.2,
  freq = 2,
  twist = 0.6,
  x_scale = 1,
  y_scale = 1,
  z_scale = 1,
  normalize = c("median", "mean", "none")
)
```

Arguments

mask	Cubic logical or numeric keep-array. Non-zero entries are retained at each recursive step.
level	Recursion depth. Must be at least 1.
surface	Cube-mask geometry family. One of "standard", "bulged", "twisted", or "wavy".
amplitude	Finite deformation amplitude.
freq	Positive modulation frequency used only when surface = "wavy".
twist	Finite twist strength used only when surface = "twisted".
x_scale	Positive horizontal scaling applied to the canonical cube coordinates.
y_scale	Positive vertical scaling applied to the canonical cube coordinates.
z_scale	Positive depth scaling applied to the canonical cube coordinates.
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Details

The mask is interpreted as a cubic array whose first dimension runs from top to bottom, second from left to right, and third from front to back. Non-zero entries are retained at each recursive step.

Value

`recursive.cube.mask.surface.embedding()` returns an `n` x `3` numeric matrix with columns `x`, `y`, and `z`.

`recursive.cube.mask.surface.graph()` returns a list with components:

- `edges`: the retained recursive cube-mask face-adjacency edges,
- `n`: number of vertices,
- `edge_weights`: induced positive edge lengths,
- `coords_surface`: the 3D embedding,
- `coords_param`: the canonical 3D voxel-center coordinates,
- `weight_scale`: the normalization constant applied to the raw edge lengths,
- `family`: always `"recursive.cube.mask"`,
- `surface`: the chosen surface name,
- `level`: the recursion depth,
- `mask`: the logical keep-array,
- `mask_side`: side length of the base keep-mask,
- `side`: side length of the fully refined cube grid,
- `label`: a human-readable family label.

`recursive_mask_grid_surface_helpers`
Weighted recursive mask-grid surface helpers

Description

Convenience helpers that recursively subdivide a square grid according to a square keep-mask, retain the occupied cells, and then lift those cells into $\mathbb{R}^3$. The induced Euclidean edge lengths become positive graph weights. This provides a generic mesh-derived family that includes the Sierpinski carpet and related masked-grid fractals such as Vicsek-style cross families.

Usage

```
recursive.mask.grid.surface.embedding(
  mask,
  level = 2,
  surface = c("saddle", "paraboloid", "ripple"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  x_scale = 1,
  y_scale = 1
)
```

```
recursive.mask.grid.surface.graph(
  mask,
  level = 2,
  surface = c("saddle", "paraboloid", "ripple"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  x_scale = 1,
  y_scale = 1,
  normalize = c("median", "mean", "none")
)
```

Arguments

mask	Square logical or numeric keep-mask with at least one retained cell. Non-zero entries are kept at each recursive step.
level	Recursion depth. Must be at least 1.
surface	Surface family used for the lift. One of "saddle", "paraboloid", or "ripple".
amplitude	Finite numeric amplitude controlling the non-flat displacement.
freq_u	Positive ripple frequency in the horizontal parameter direction. Used only when surface = "ripple".
freq_v	Positive ripple frequency in the vertical parameter direction. Used only when surface = "ripple".
x_scale	Positive horizontal scaling of the parameter domain.
y_scale	Positive vertical scaling of the parameter domain.
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Details

The mask is interpreted in standard matrix display orientation: rows run from top to bottom and columns run from left to right. Non-zero entries are retained at each recursive subdivision step.

‘recursive.mask.grid.surface.embedding()‘ returns the 3D coordinates of the occupied cells in the same vertex order as edges.recursive.mask.grid(). ‘recursive.mask.grid.surface.graph()‘ returns a reusable weighted-graph bundle containing the masked-grid edges, induced edge weights, the 3D surface coordinates, and the 2D parameter coordinates of the occupied cells.

Value

recursive.mask.grid.surface.embedding() returns an n x 3 numeric matrix with columns x, y, and z.

recursive.mask.grid.surface.graph() returns a list with components:

- edges: the occupied-cell masked-grid edges,
- n: number of vertices,
- edge_weights: induced positive edge lengths,

- `coords_surface`: the 3D surface embedding,
- `coords_param`: the 2D parameter coordinates,
- `weight_scale`: the normalization constant applied to the raw edge lengths,
- `family`: always `"recursive.mask.grid"`,
- `surface`: the chosen surface name,
- `level`: the recursion depth,
- `mask`: the logical keep-mask,
- `mask_size`: the side length of the mask,
- `side`: the side length of the fully refined grid,
- `label`: a human-readable family label.

`recursive_tetrahedron_mask_surface_helpers`
Weighted recursive tetrahedron-mask surface helpers

Description

Convenience helpers that recursively subdivide a tetrahedron into four corner subtetrahedra according to a four-corner keep-mask, retain the selected subtetrahedra at each recursion step, and deform the resulting simplicial graph inside $\mathbb{R}^3$. This provides a generic tetrahedral gasket family that includes the classic Sierpinski tetrahedron together with corner-omission variants.

Usage

```
recursive.tetrahedron.mask.surface.embedding(
  mask = mask.tetrahedron.classic(),
  level = 2,
  surface = c("standard", "squashed", "twisted", "wavy"),
  amplitude = 0.3,
  freq = 2,
  twist = 0.6
)

recursive.tetrahedron.mask.surface.graph(
  mask = mask.tetrahedron.classic(),
  level = 2,
  surface = c("standard", "squashed", "twisted", "wavy"),
  amplitude = 0.3,
  freq = 2,
  twist = 0.6,
  normalize = c("median", "mean", "none")
)
```

Arguments

mask	Four-entry logical or numeric keep-mask. Non-zero entries are retained at each recursive step.
level	Recursion depth. May be 0 or larger.
surface	Tetrahedron surface family. One of "standard", "squashed", "twisted", or "wavy".
amplitude	Finite deformation amplitude.
freq	Positive modulation frequency used only when surface = "wavy".
twist	Finite twist strength used only when surface = "twisted".
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Details

The mask entries are interpreted in the order base_left, base_right, base_back, and apex. Named masks are reordered automatically to match that convention.

Value

recursive.tetrahedron.mask.surface.embedding() returns an n x 3 numeric matrix with columns x, y, and z.

recursive.tetrahedron.mask.surface.graph() returns a list with components:

- edges: the retained recursive tetrahedron-mask edges,
- n: number of vertices,
- edge_weights: induced positive edge lengths,
- coords_surface: the 3D surface embedding,
- coords_param: the canonical 3D tetrahedron coordinates,
- weight_scale: the normalization constant applied to the raw edge lengths,
- family: always "recursive.tetrahedron.mask",
- surface: the chosen surface name,
- level: the recursion depth,
- mask: the logical keep-mask,
- label: a human-readable family label.

`recursive_triangle_mask_surface_helpers` *Weighted recursive triangle-mask surface helpers*

Description

Convenience helpers that recursively subdivide an equilateral triangle into three corner subtriangles plus one central inverted subtriangle according to a four-slot keep-mask, retain the selected subtriangles at each recursion step, and lift the resulting vertex graph into $\mathbb{R}^3$. This provides a generic gasket-style triangular family that includes the classic Sierpinski triangle together with bridge and asymmetric variants.

Usage

```
recursive.triangle.mask.surface.embedding(
  mask = mask.triangle.classic(),
  level = 2,
  surface = c("flat", "saddle", "paraboloid", "ripple", "folded"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  x_scale = 1,
  y_scale = 1
)

recursive.triangle.mask.surface.graph(
  mask = mask.triangle.classic(),
  level = 2,
  surface = c("flat", "saddle", "paraboloid", "ripple", "folded"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  x_scale = 1,
  y_scale = 1,
  normalize = c("median", "mean", "none")
)
```

Arguments

`mask`	Four-entry logical or numeric keep-mask. Non-zero entries are retained at each recursive step.
`level`	Recursion depth. May be `0` or larger.
`surface`	Triangle surface family. One of `"flat"`, `"saddle"`, `"paraboloid"`, `"ripple"`, or `"folded"`.
`amplitude`	Finite deformation amplitude.

freq_u	Positive ripple frequency in the first canonical triangle coordinate. Used only when surface = "ripple".
freq_v	Positive ripple frequency in the second canonical triangle coordinate. Used only when surface = "ripple".
x_scale	Positive horizontal scaling applied to the canonical triangle coordinates.
y_scale	Positive vertical scaling applied to the canonical triangle coordinates.
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Details

The mask entries are interpreted in the order left, right, top, and center. Named masks are reordered automatically to match that convention.

Value

recursive.triangle.mask.surface.embedding() returns an n x 3 numeric matrix with columns x, y, and z.

recursive.triangle.mask.surface.graph() returns a list with components:

- edges: the retained recursive triangle-mask edges,
- n: number of vertices,
- edge_weights: induced positive edge lengths,
- coords_surface: the 3D surface embedding,
- coords_param: the canonical 2D triangle coordinates,
- weight_scale: the normalization constant applied to the raw edge lengths,
- family: always "recursive.triangle.mask",
- surface: the chosen surface name,
- level: the recursion depth,
- mask: the logical keep-mask,
- label: a human-readable family label.

---

repulsive.stage *Optimize one pure repulsive layout stage*

Description

‘repulsive.stage()‘ performs a gradient-descent stage for the pure repulsion objective evaluated by [repulsive.state()]. It is intended for GMDS experiments that need to inspect repulsion separately from edge-length repair.

Usage

```
repulsive.stage(
  coords,
  lambda = 1,
  pair.index = NULL,
  pair.weights = NULL,
  repulsion.family = c("log", "inverse_power"),
  repulsion.delta = 0.001,
  repulsion.power = 1,
  max.iter = 80L,
  initial.step = 0.02,
  step.shrink = 0.5,
  armijo = 1e-04,
  min.step = 1e-08,
  grad.tol = 1e-07,
  recenter = TRUE,
  distance.eps = 1e-10,
  return.frames = FALSE,
  engine = c("cpp", "R")
)
```

Arguments

coords	Numeric ‘n‘ by ‘dim‘ coordinate matrix.
lambda	Repulsion strength. The returned total energy is ‘lambda * repel.energy‘; the gradient is scaled by ‘lambda‘.
pair.index	Optional two-column matrix of 1-based vertex pairs. If ‘NULL‘ and ‘lambda > 0‘, all unordered pairs are used.
pair.weights	Optional non-negative pair weights parallel to ‘pair.index‘.
repulsion.family	Repulsion potential family, either ‘"log"‘ or ‘"inverse_power"‘.
repulsion.delta	Small positive softening parameter for pair distances.
repulsion.power	Power used by the ‘"inverse_power"‘ repulsion.
max.iter	Maximum number of gradient-descent iterations.
initial.step	Initial gradient step size.
step.shrink	Multiplicative shrink factor used by backtracking.
armijo	Armijo sufficient-decrease coefficient.
min.step	Minimum allowable step before the stage stops.
grad.tol	Gradient-norm stopping tolerance.
recenter	Logical; whether to recenter coordinates after each proposal.
distance.eps	Small positive distance floor used in derivatives.
return.frames	Logical; whether to return accepted coordinate frames. Frame 0 is the starting coordinate matrix and later frames are accepted updates.
engine	Backend engine. ‘"cpp"‘ is the default; ‘"R"‘ uses the reference implementation.

Value

A list with ‘coords‘, final ‘state‘, a data-frame ‘trace‘, and, when requested, a list of coordinate ‘frames‘.

`repulsive.state` *Evaluate a pure repulsive layout objective*

Description

‘repulsive.state()‘ evaluates only the repulsion term used by the experimental GMDS repulsive-unfolding operators. Unlike [edge.repulsive.state()], this function has no graph-edge term and does not require placeholder edges or edge lengths.

Usage

```
repulsive.state(
  coords,
  lambda = 1,
  pair.index = NULL,
  pair.weights = NULL,
  repulsion.family = c("log", "inverse_power"),
  repulsion.delta = 0.001,
  repulsion.power = 1,
  distance.eps = 1e-10,
  engine = c("cpp", "R")
)
```

Arguments

`coords`	Numeric ‘n‘ by ‘dim‘ coordinate matrix.
`lambda`	Repulsion strength. The returned total energy is ‘lambda * repel.energy‘; the gradient is scaled by ‘lambda‘.
`pair.index`	Optional two-column matrix of 1-based vertex pairs. If ‘NULL‘ and ‘lambda > 0‘, all unordered pairs are used.
`pair.weights`	Optional non-negative pair weights parallel to ‘pair.index‘.
`repulsion.family`	Repulsion potential family, either ‘"log"‘ or ‘"inverse_power"‘.
`repulsion.delta`	Small positive softening parameter for pair distances.
`repulsion.power`	Power used by the ‘"inverse_power"‘ repulsion.
`distance.eps`	Small positive distance floor used in derivatives.
`engine`	Backend engine. ‘"cpp"‘ is the default; ‘"R"‘ uses the reference implementation.

Value

A list containing energy, unscaled repulsion energy, gradient, gradient norm, and embedded pair lengths.

`run_gripui` *Run the 'gripui' Shiny application*

Description

Run the 'gripui' Shiny application

Usage

```
run_gripui(
  project,
  host = "127.0.0.1",
  port = getOption("shiny.port"),
  launch.browser = interactive(),
  auto.stop.after = NULL,
  ...
)
```

Arguments

`project`	A 'gripui_project'.
`host`	Host passed to 'shiny::runApp()'.
`port`	Port passed to 'shiny::runApp()'.
`launch.browser`	Whether to launch a browser.
`auto.stop.after`	Optional delay, in seconds, after which the app stops itself. This is mainly useful for automated examples and tests.
`...`	Additional arguments passed to 'shiny::runApp()'.

Value

Invisibly returns the result of 'shiny::runApp()'.

Examples

```
graph <- list(adj_list = list(2L, c(1L, 3L), 2L))
layouts <- data.frame(
  candidate = "toy.layout",
  stage = "layout",
  seed = 1L,
  status = "ok",
  stringsAsFactors = FALSE
```

```
)
project <- gripui_project(graph = graph, layouts = layouts, title = "Toy project")
run_gripui(
  project,
  launch.browser = FALSE,
  quiet = TRUE,
  auto.stop.after = 0.1
)
```

`run_gripui_family` *Run the graph-family geometry explorer Shiny application*

Description

Run the graph-family geometry explorer Shiny application

Usage

```
run_gripui_family(
  catalog = gripui_graph_family_catalog(),
  title = "Graph Family Geometry Explorer",
  subtitle = "Interactive geometry browser for synthetic benchmark families.",
  host = "127.0.0.1",
  port = getOption("shiny.port"),
  launch.browser = interactive(),
  auto.stop.after = NULL,
  ...
)
```

Arguments

`catalog`	Family catalog, usually ‘gripui_graph_family_catalog()‘.
`title`	Application title.
`subtitle`	Optional subtitle shown in the sidebar.
`host`	Host passed to ‘shiny::runApp()‘.
`port`	Port passed to ‘shiny::runApp()‘.
`launch.browser`	Whether to launch a browser.
`auto.stop.after`	Optional delay, in seconds, after which the app stops itself. This is mainly useful for automated examples and tests.
`...`	Additional arguments passed to ‘shiny::runApp()‘.

Value

Invisibly returns the result of ‘shiny::runApp()‘.

Examples

```
run_gripui_family(launch.browser = FALSE, quiet = TRUE, auto.stop.after = 0.1)
```

---

sampled_rectangle_surface_helpers
Sampled rectangle surface helpers

Description

Convenience helpers that sample points uniformly from a user-specified rectangle, lift the sampled planar coordinates into $\mathbb{R}^3$, and then build exact intersection-kNN graphs on either the planar or embedded coordinates. The lifted surface uses normalized rectangle coordinates for the non-flat z-displacement while preserving the original rectangle in the returned x and y coordinates.

Usage

```
sampled.rectangle.param.coords(
  n,
  xmin = -1,
  xmax = 1,
  ymin = -1,
  ymax = 1,
  seed = NULL
)

sampled.rectangle.surface.embedding(
  n,
  xmin = -1,
  xmax = 1,
  ymin = -1,
  ymax = 1,
  seed = NULL,
  surface = c("flat", "saddle", "paraboloid", "ripple", "folded"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1
)

sampled.rectangle.surface.graph(
  n,
  k,
  xmin = -1,
  xmax = 1,
  ymin = -1,
  ymax = 1,
```

```
  seed = NULL,
  surface = c("flat", "saddle", "paraboloid", "ripple", "folded"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  graph_space = c("surface", "param"),
  max.path.edge.ratio.deviation.thld = 0.1,
  path.edge.ratio.percentile = 0.5,
  threshold.percentile = 0,
  normalize = c("median", "mean", "none")
)

sampled.rectangle.surface.graphs(
  n,
  k,
  xmin = -1,
  xmax = 1,
  ymin = -1,
  ymax = 1,
  seed = NULL,
  surface = c("flat", "saddle", "paraboloid", "ripple", "folded"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  graph_space = c("surface", "param"),
  max.path.edge.ratio.deviation.thld = 0.1,
  path.edge.ratio.percentile = 0.5,
  threshold.percentile = 0,
  normalize = c("median", "mean", "none")
)
```

Arguments

n	Number of sampled points.
xmin	Left rectangle boundary.
xmax	Right rectangle boundary. Must satisfy xmax > xmin.
ymin	Bottom rectangle boundary.
ymax	Top rectangle boundary. Must satisfy ymax > ymin.
seed	Optional integer seed used only for the rectangle sampling.
surface	Geometry family used for the 3D lift. One of "flat", "saddle", "paraboloid", "ripple", or "folded".
amplitude	Finite deformation amplitude.
freq_u	Positive ripple frequency in the horizontal rectangle direction. Used only when surface = "ripple".
freq_v	Positive ripple frequency in the vertical rectangle direction. Used only when surface = "ripple".

k — Integer iKNN neighborhood size (single graph) or vector of neighborhood sizes (graph sequence).

graph_space — Coordinate system used to build the iKNN graph and its raw edge weights. "surface" uses the 3D embedding and "param" uses the sampled planar coordinates.

max.path.edge.ratio.deviation.thld — Geometric-pruning deviation threshold in [0, 0.2).

path.edge.ratio.percentile — Edge-length percentile in [0, 1] used to select candidates for geometric pruning.

threshold.percentile — Optional long-edge pruning percentile in [0, 0.5]. A value of 0 disables this stage.

normalize — Normalization applied to the final positive edge weights. One of "median", "mean", or "none".

Details

For the graph constructors, the iKNN relation determines which edges are present, while the returned edge lengths are taken from the 3D endpoint distances in the lifted embedding so that the graph metric reflects the chosen surface geometry.

‘sampled.rectangle.param.coords()‘ returns the sampled planar coordinates. ‘sampled.rectangle.surface.embedding()‘ returns the corresponding 3D embedding. ‘sampled.rectangle.surface.graph()‘ returns a single weighted iKNN graph for one k value, while ‘sampled.rectangle.surface.graphs()‘ reuses the same sample and embedding across a sequence of k values.

Value

sampled.rectangle.param.coords() returns an n x 2 numeric matrix with columns u and v.

sampled.rectangle.surface.embedding() returns an n x 3 numeric matrix with columns x, y, and z.

sampled.rectangle.surface.graph() returns a list with components:

- edges: the undirected iKNN edges,
- n: number of vertices,
- edge_weights: normalized positive 3D chord lengths on the retained iKNN edges,
- raw_edge_weights: the unnormalized 3D chord lengths before normalize,
- iknn_witness_edge_weights: raw iKNN witness weights based on common-neighbor distances, retained for diagnostics,
- coords_surface: the sampled 3D embedding,
- coords_param: the sampled planar rectangle coordinates,
- coords_param_unit: the centered unit-rectangle coordinates used for the surface lift,
- weight_scale: normalization constant applied to the raw edge weights,
- family: always "sampled.rectangle",

- surface: the chosen surface name,
- graph_space: the coordinate system used for iKNN graph construction,
- k: the iKNN neighborhood size,
- label: a human-readable family label.

sampled.rectangle.surface.graphs() returns a list with components:

- graphs: named list of single-k graph bundles,
- k: integer vector of retained k values,
- k_statistics: per-k edge-count summary,
- coords_surface: the shared sampled 3D embedding,
- coords_param: the shared sampled planar coordinates,
- coords_param_unit: the centered unit-rectangle coordinates used for the surface lift,
- family: always "sampled.rectangle",
- surface: the chosen surface name,
- graph_space: the coordinate system used for iKNN graph construction,
- label: a human-readable family label.

---

score.geodesic.kk *Score a layout under the full geodesic KK energy*

Description

score.geodesic.kk() evaluates a layout using the full all-pairs geodesic Kamada–Kawai objective. Distances are measured along fixed chosen graph shortest paths, not by straight-line chord length.

Usage

```
score.geodesic.kk(
  coords,
  prepared = NULL,
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  stiffness = 1,
  distance_floor = 1e-08,
  edge_length_epsilon = 1e-08,
  scale_mode = c("profiled", "user"),
  scale.L0 = NULL,
  return_pair_details = FALSE
)
```

Arguments

coords	Numeric coordinate matrix with 2 or 3 columns.
prepared	Optional object returned by prepare.geodesic.kk().
edges	Two-column integer matrix of edges (1-based vertex ids).
n	Number of vertices.
adj_list	Adjacency list (1-based) for an undirected graph.
weight_list	Optional parallel list of positive edge weights.
edge_weights	Optional positive edge-weight vector parallel to edges.
stiffness	Global stiffness constant \(K\).
distance_floor	Small positive floor used in k_ij = K / max(g_ij, distance_floor)^2.
edge_length_epsilon	Small positive stabilizer added inside each embedded edge length.
scale_mode	Either "profiled" to fit L0 analytically for the supplied layout or "user" to use scale.L0.
scale.L0	Optional user-supplied geodesic KK scale, required when scale_mode = "user".
return_pair_details	If TRUE, include pairwise path lengths and residuals in a list column.

Details

By default the global target scale L0 is fit analytically for the supplied layout. Alternatively, the score can be evaluated at a user-supplied scale.

Value

A one-row data frame with the fitted or user-supplied scale factor and full geodesic KK energy summary.

---

score.gmds *Score a layout with the common GMDS diagnostic panel*

Description

'score.gmds()' provides a common diagnostic panel for GMDS-oriented layouts. It summarizes edge-length fidelity, fixed-path geodesic stress, short/mid/long geodesic bands, ordinary metric-MDS chord stress, and simple spread/shortcut diagnostics for any coordinate matrix on a prepared graph.

Usage

```
score.gmds(
  coords,
  prepared = NULL,
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  scale_mode = c("profiled", "identity", "user"),
  edge_scale = NULL,
  path_scale = NULL,
  chord_scale = NULL,
  distance_floor = 1e-08,
  edge_length_epsilon = 1e-08,
  band_quantiles = c(1/3, 2/3)
)
```

Arguments

coords	Numeric coordinate matrix.
prepared	Optional object returned by [prepare.edge.kk()], [prepare.graph.geodesic.mds()] or [prepare.geodesic.kk()]. Edge-only objects from [prepare.edge.kk()] report only edge diagnostics; all-pairs GMDS path and chord diagnostics are unavailable.
edges	Two-column edge matrix used when ‘prepared‘ is omitted.
n	Number of vertices used when ‘prepared‘ is omitted.
adj_list	Optional adjacency list used when ‘prepared‘ is omitted.
weight_list	Optional edge-weight list parallel to ‘adj_list‘.
edge_weights	Optional positive edge weights parallel to ‘edges‘.
scale_mode	Scale policy for edge, path, and chord targets: ‘"profiled"‘ fits one scalar for each diagnostic family, ‘"identity"‘ uses scale one, and ‘"user"‘ uses the corresponding supplied scale.
edge_scale, path_scale, chord_scale	Optional user scales used when ‘scale_mode = "user"‘.
distance_floor	Positive floor for relative residuals.
edge_length_epsilon	Small stabilizer for fixed-path embedded lengths.
band_quantiles	Two quantiles splitting graph distances into short, mid, and long bands.

Value

A one-row data frame with the common diagnostic columns.

score.landmark.geodesic.kk *Score a layout under the landmark geodesic KK energy*

Description

score.landmark.geodesic.kk() evaluates a layout using the landmark geodesic KK objective described in landmark_geodesic_kk_spec_2026-03-30.tex. Distances are measured along fixed chosen graph shortest paths, not by straight-line chord length. The target scale factor L0 is fit analytically for the supplied layout so that rankings are not dominated by an arbitrary global drawing scale.

Usage

```
score.landmark.geodesic.kk(
  coords,
  prepared = NULL,
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  local_nbrs = 20L,
  landmark_count = 8L,
  stiffness = 1,
  distance_floor = 1e-08,
  edge_length_epsilon = 1e-08,
  return_pair_details = FALSE
)
```

Arguments

coords	Numeric coordinate matrix with 2 or 3 columns.
prepared	Optional object returned by prepare.landmark.geodesic.kk().
edges	Two-column integer matrix of edges (1-based vertex ids).
n	Number of vertices.
adj_list	Adjacency list (1-based) for an undirected graph.
weight_list	Optional parallel list of positive edge weights.
edge_weights	Optional positive edge-weight vector parallel to edges.
local_nbrs	Number of nearest graph-metric neighbors retained per vertex when prepared is not supplied.
landmark_count	Number of farthest-point landmarks retained per vertex when prepared is not supplied.
stiffness	Global stiffness constant \(K\).

`distance_floor` Small positive floor used in `k_ij = K / max(g_ij, distance_floor)^2`.

`edge_length_epsilon`
: Small positive stabilizer added inside each embedded edge length.

`return_pair_details`
: If `TRUE`, include pairwise path lengths and residuals in a list column.

Details

This is a scoring and comparison helper, not an optimizer.

Value

A one-row data frame with the fitted scale factor and landmark geodesic KK energy summary.

---

score.layout *Score a single layout using graph-aware quality heuristics*

Description

`score.layout()` evaluates a realized layout without assuming a canonical embedding. It is the low-level scoring helper behind `compare.layouts()` and is most useful when you already have one realized layout in hand, for example from a cached run or another graph drawing tool. For real-world graphs, quality is judged by graph-distance faithfulness, edge-length consistency, separation of non-neighbors, and optionally edge crossings or cluster separation.

Usage

```
score.layout(
  coords,
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  clusters = NULL,
  sample.size.stress = 2000L,
  sample.size.nonedge = 5000L,
  stress.seed = 1L,
  nonedge.seed = 1L,
  edge.crossings = c("auto", "always", "never"),
  edge.crossings.max.edges = 1000L
)
```

Arguments

coords	Numeric coordinate matrix with 2 or 3 columns.
edges	Two-column integer matrix of edges (1-based vertex ids).
n	Number of vertices. If omitted with adj_list, defaults to length(adj_list). If omitted with edges, defaults to nrow(coords).
adj_list	Adjacency list (1-based) for undirected graphs.
weight_list	Optional parallel list of positive edge weights.
edge_weights	Optional positive edge-weight vector parallel to edges.
clusters	Optional cluster or community labels of length nrow(coords). When supplied, cluster.separation is reported.
sample.size.stress	Number of vertex pairs sampled for sampled.stress.
sample.size.nonedge	Number of non-edge pairs sampled for sampled.nonedge.sep.ratio.
stress.seed	RNG seed used for the stress sample.
nonedge.seed	RNG seed used for the non-edge sample.
edge.crossings	How to compute edge.crossings for 2D layouts: "auto" computes exact crossings only when the graph is small enough, "always" always computes them, and "never" skips them.
edge.crossings.max.edges	Edge-count threshold used by edge.crossings = "auto".

Value

A one-row data frame with dot-delimited metric names.

Examples

```
edges <- edges.mesh(5, 5)
coords <- grip(edges, n = 25, dim = 2, preset = "mesh", seed = 1)
score.layout(coords, edges = edges, n = 25)
```

---

score.misf.geodesic.kk

Score a MISF-based geodesic-KK fit

Description

'score.misf.geodesic.kk()' summarizes external coordinates against a MISF-GKK prepared object using either the exact full geodesic-KK scorer or the sparse landmark geodesic-KK scorer, depending on 'score_pair_mode'. When a full MISF-GKK fit is supplied, the function also carries over the multiscale stage metadata and any trace tables retained by the optimizer.

Usage

```
score.misf.geodesic.kk(
  fit = NULL,
  coords = NULL,
  prepared = NULL,
  stiffness = 1,
  distance_floor = 1e-08,
  edge_length_epsilon = 1e-08,
  score_pair_mode = c("full", "landmark", "auto"),
  score_full_limit = 2048L,
  score_local_nbrs = 20L,
  score_landmark_count = 8L,
  return_trace = FALSE
)
```

Arguments

`fit`	Optional fit from [misf.geodesic.kk()].
`coords`	Optional coordinate matrix used when ‘fit‘ is omitted.
`prepared`	Optional MISF-GKK prepared object used when ‘fit‘ is omitted.
`stiffness`	Global stiffness constant \(K\).
`distance_floor`	Small positive floor used in `k_ij = K / max(g_ij, distance_floor)^2`.
`edge_length_epsilon`	Small positive stabilizer added inside each embedded edge length.
`score_pair_mode`	Requested scoring pair policy: ‘"full"‘, ‘"landmark"‘, or ‘"auto"‘.
`score_full_limit`	Active-set size threshold used when ‘score_pair_mode = "auto"‘.
`score_local_nbrs`	Sparse local-neighborhood size used for landmark-based scoring.
`score_landmark_count`	Sparse landmark count used for landmark-based scoring.
`return_trace`	If ‘TRUE‘, attach any trace tables stored inside ‘fit‘.

Value

A one-row data frame with final MISF-GKK summary fields and either full-GKK or landmark-GKK score columns, depending on the resolved scoring policy. If ‘return_trace = TRUE‘, trace list-columns are attached when available.

`sierpinski_carpet_surface_helpers` *Weighted Sierpinski carpet surface helpers*

Description

Convenience wrappers around `recursive.mask.grid.surface.*()` using the classic 3×3 carpet mask with the center cell removed.

Usage

```
sierpinski.carpet.surface.embedding(
  level = 2,
  surface = c("saddle", "paraboloid", "ripple"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  x_scale = 1,
  y_scale = 1
)

sierpinski.carpet.surface.graph(
  level = 2,
  surface = c("saddle", "paraboloid", "ripple"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  x_scale = 1,
  y_scale = 1,
  normalize = c("median", "mean", "none")
)
```

Arguments

`level`	Recursion depth. Must be at least `1`.
`surface`	Surface family used for the lift. One of `"saddle"`, `"paraboloid"`, or `"ripple"`.
`amplitude`	Finite numeric amplitude controlling the non-flat displacement.
`freq_u`	Positive ripple frequency in the horizontal parameter direction. Used only when `surface = "ripple"`.
`freq_v`	Positive ripple frequency in the vertical parameter direction. Used only when `surface = "ripple"`.
`x_scale`	Positive horizontal scaling of the parameter domain.
`y_scale`	Positive vertical scaling of the parameter domain.
`normalize`	Normalization applied to the induced edge lengths. One of `"median"`, `"mean"`, or `"none"`.

Details

'sierpinski.carpet.surface.embedding()' returns the 3D coordinates of the occupied cells in the same vertex order as `edges.sierpinski.carpet()`. 'sierpinski.carpet.surface.graph()' returns a reusable weighted-graph bundle for the named Sierpinski carpet family.

Value

`sierpinski.carpet.surface.embedding()` returns an `n` x `3` numeric matrix with columns `x`, `y`, and `z`, where `n = 8^level`.

`sierpinski.carpet.surface.graph()` returns the same components as `recursive.mask.grid.surface.graph()`, with `family` set to `"sierpinski.carpet"` and a family-specific class and label.

`sierpinski_tetrahedron_surface_helpers`

Weighted Sierpinski tetrahedron surface helpers

Description

Convenience wrappers around `recursive.tetrahedron.mask.surface.*()` using the classic all-corner tetrahedron mask. These helpers are intended for benchmark families where the topology is the standard simplicial tetrahedral gasket but the intended metric comes from a squashed, twisted, or spatially varying realization.

Usage

```
sierpinski.tetrahedron.surface.embedding(
  level = 2,
  surface = c("standard", "squashed", "twisted", "wavy"),
  amplitude = 0.3,
  freq = 2,
  twist = 0.6
)

sierpinski.tetrahedron.surface.graph(
  level = 2,
  surface = c("standard", "squashed", "twisted", "wavy"),
  amplitude = 0.3,
  freq = 2,
  twist = 0.6,
  normalize = c("median", "mean", "none")
)
```

Arguments

`level`	Recursion depth. May be `0` or larger.

surface	Tetrahedron surface family. One of "standard", "squashed", "twisted", or "wavy".
amplitude	Finite deformation amplitude.
freq	Positive modulation frequency used only when surface = "wavy".
twist	Finite twist strength used only when surface = "twisted".
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Value

sierpinski.tetrahedron.surface.embedding() returns an n x 3 numeric matrix with columns x, y, and z.

sierpinski.tetrahedron.surface.graph() returns the same components as recursive.tetrahedron.mask.surface.g with family set to "sierpinski.tetrahedron" and a family-specific class and label.

---

sierpinski_triangle_surface_helpers

Weighted Sierpinski triangle surface helpers

Description

Convenience helpers that take the canonical recursive embedding of the Sierpinski triangle graph and lift it into $\mathbb{R}^3$. These helpers are intended for benchmark families where the topology is the usual triangular gasket but the intended metric comes from a curved or folded geometry rather than the flat equilateral reference drawing.

Usage

```
sierpinski.triangle.surface.embedding(
  level = 2,
  surface = c("flat", "saddle", "paraboloid", "ripple", "folded"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  x_scale = 1,
  y_scale = 1
)

sierpinski.triangle.surface.graph(
  level = 2,
  surface = c("flat", "saddle", "paraboloid", "ripple", "folded"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  x_scale = 1,
```

```
  y_scale = 1,
  normalize = c("median", "mean", "none")
)
```

Arguments

level	Recursion depth. May be 0 or larger.
surface	Triangle surface family. One of "flat", "saddle", "paraboloid", "ripple", or "folded".
amplitude	Finite deformation amplitude.
freq_u	Positive ripple frequency in the first canonical triangle coordinate. Used only when surface = "ripple".
freq_v	Positive ripple frequency in the second canonical triangle coordinate. Used only when surface = "ripple".
x_scale	Positive horizontal scaling applied to the canonical triangle coordinates.
y_scale	Positive vertical scaling applied to the canonical triangle coordinates.
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Value

sierpinski.triangle.surface.embedding() returns an n x 3 numeric matrix with columns x, y, and z.

sierpinski.triangle.surface.graph() returns a list with components:

- edges: the Sierpinski triangle edges,
- n: number of vertices,
- edge_weights: induced positive edge lengths,
- coords_surface: the 3D surface embedding,
- coords_param: the canonical 2D triangle coordinates,
- weight_scale: the normalization constant applied to the raw edge lengths,
- family: always "sierpinski.triangle",
- surface: the chosen surface name,
- level: the recursion depth,
- label: a human-readable family label.

sphere_surface_helpers *Weighted sphere surface helpers*

Description

Convenience helpers that embed the sampled sphere graph into $\mathbb{R}^3$ and use the induced Euclidean edge lengths as positive graph weights. These helpers are intended for benchmark families where the graph topology follows the current pole-plus-latitude-rings sphere graph but the intended metric comes from a curved or spatially varying 3D realization.

Usage

```
sphere.surface.embedding(
  h,
  w = h,
  surface = c("standard", "ellipsoid", "wavy"),
  radius = 1,
  amplitude = 0.2,
  freq_theta = 3,
  freq_lat = 2,
  twist = 0.25
)

sphere.surface.graph(
  h,
  w = h,
  surface = c("standard", "ellipsoid", "wavy"),
  radius = 1,
  amplitude = 0.2,
  freq_theta = 3,
  freq_lat = 2,
  twist = 0.25,
  normalize = c("median", "mean", "none")
)
```

Arguments

h	Number of latitude levels including the poles. Must be at least 3.
w	Wrapped longitude count. Defaults to h; must be at least 3.
surface	Sphere surface family. One of "standard", "ellipsoid", or "wavy".
radius	Positive baseline radius.
amplitude	Finite deformation amplitude. For "ellipsoid", positive values make the shape oblate and negative values make it prolate, while keeping all axis lengths positive. For "wavy", the local radius must remain positive everywhere.

freq_theta	Positive longitudinal frequency used only when surface = "wavy".
freq_lat	Positive latitudinal frequency used only when surface = "wavy".
twist	Finite longitude twist applied smoothly by latitude. The twist vanishes at the poles.
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Details

'sphere.surface.embedding()' returns the 3D coordinates of the sampled surface in the same vertex order as edges.sphere(): north pole first, then latitude rings from north to south, then the south pole. 'sphere.surface.graph()' returns a reusable weighted-graph bundle containing the sphere edges, induced edge weights, the 3D surface coordinates, and a 2D longitude-latitude parameterization.

Value

sphere.surface.embedding() returns an n x 3 numeric matrix with columns x, y, and z, where n = 2 + (h - 2) * w.

sphere.surface.graph() returns a list with components:

- edges: the undirected sphere-graph edges,
- n: number of vertices,
- edge_weights: induced positive edge lengths,
- coords_surface: the 3D surface embedding,
- coords_param: the 2D longitude-latitude parameter coordinates,
- weight_scale: the normalization constant applied to the raw edge lengths,
- family: always "sphere",
- surface: the chosen surface name,
- label: a human-readable family label.

---

tetrahedron_mask_helpers

Tetrahedron mask helpers for recursive gasket families

Description

Convenience constructors for the four-corner tetrahedron masks used by edges.recursive.tetrahedron.mask() and recursive.tetrahedron.mask.surface.graph(). The slots correspond to the four corner subtetrahedra created by one barycentric refinement of a tetrahedron.

Usage

```
mask.tetrahedron.classic()

mask.tetrahedron.corner.missing(
  omit = c("apex", "base_left", "base_right", "base_back")
)
```

Arguments

`omit`	Which tetrahedron corner is omitted in `mask.tetrahedron.corner.missing()`.

Details

The returned mask entries are ordered and named as `base_left`, `base_right`, `base_back`, and `apex`.

Value

A named logical vector of length 4.

`torus_surface_helpers` *Weighted torus surface helpers*

Description

Convenience helpers that embed a toroidal grid graph into $\mathbb{R}^3$ and use the induced Euclidean edge lengths as positive graph weights. These helpers are intended for benchmark families where the graph topology is toroidal but the intended metric comes from a curved or spatially varying 3D realization.

Usage

```
torus.surface.embedding(
  h,
  w = h,
  surface = c("standard", "pinched", "wavy"),
  major_radius = 2,
  minor_radius = 0.75,
  amplitude = 0.2,
  freq_major = 2,
  freq_minor = 1,
  twist = 0.25
)

torus.surface.graph(
  h,
  w = h,
```

```
  surface = c("standard", "pinched", "wavy"),
  major_radius = 2,
  minor_radius = 0.75,
  amplitude = 0.2,
  freq_major = 2,
  freq_minor = 1,
  twist = 0.25,
  normalize = c("median", "mean", "none")
)
```

Arguments

h	Number of wrapped rows. Must be at least 3.
w	Number of wrapped columns. Defaults to h; must be at least 3.
surface	Torus surface family. One of "standard", "pinched", or "wavy".
major_radius	Positive distance from the torus center to the center of the tube.
minor_radius	Positive baseline radius of the torus tube. The local tube radius must remain strictly smaller than major_radius everywhere.
amplitude	Finite numeric deformation amplitude.
freq_major	Positive angular frequency around the major cycle, used by "wavy" and the periodic twist.
freq_minor	Positive angular frequency around the minor cycle, used by "wavy".
twist	Finite phase twist applied periodically to the minor angle as a function of the major angle.
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Details

'torus.surface.embedding()' returns the 3D coordinates of the embedded toroidal grid. 'torus.surface.graph()' returns a reusable weighted-graph bundle containing the torus edges, induced edge weights, the 3D surface coordinates, and a 2D unwrapped parameterization.

Value

`torus.surface.embedding()` returns an n x 3 numeric matrix with columns x, y, and z.

`torus.surface.graph()` returns a list with components:

- edges: the undirected toroidal-grid edges,
- n: number of vertices,
- edge_weights: induced positive edge lengths,
- coords_surface: the 3D surface embedding,
- coords_param: the 2D unwrapped parameter coordinates,
- weight_scale: the normalization constant applied to the raw edge lengths,
- family: always "torus",

- surface: the chosen surface name,
- label: a human-readable family label.

`trace.grip` *Trace a GRIP layout*

Description

This is the tracing counterpart to `grip()`. The `metric` argument selects the hop-based or edge-length-based compiled engine while the remaining trace and diagnostic options keep the same meaning in both cases. See `grip()` for detailed edge-length semantics.

Usage

```
trace.grip(
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  dim = 3,
  placement = c("barycenter", "circle"),
  preset = NULL,
  rounds = 160,
  final_rounds = 384,
  num_init = 24,
  num_nbrs = 20,
  r = 0.03,
  s = 7.5,
  repulsion_factor = 2.5,
  coarse_repulsion_factor = 1.5,
  coarse_repulsion_sample = 16,
  coarse_repulsion_exact_below = 64,
  final_anchor_factor = 0,
  final_move_scale_after_first = 1,
  final_mode = c("fr", "kk_repulse"),
  insertion_anchor_count = 3,
  insertion_anchor_scope = c("any_higher", "prev_misf"),
 insertion_anchor_strategy = c("first", "distance_band", "balanced_band", "spread_prev"),
  level0_insertion_mode = c("inherit", "barycenter", "least_squares"),
  level0_anchor_count = insertion_anchor_count,
  level0_local_kk_steps = 3,
  lgkk_polish_rounds = 0L,
  lgkk_multiscale_rounds = 0L,
  lgkk_rounds_coarse = NULL,
  lgkk_rounds_pre_final = NULL,
  lgkk_rounds_final = NULL,
```

```
  lgkk_local_nbrs = 20L,
  lgkk_landmark_count = 8L,
  lgkk_multiscale_scope = c("all", "coarse"),
  lgkk_active_limit = 4096L,
  tinit_factor = 6,
  seed = 6,
  trace = c("round", "level"),
  trace.every = 1,
  diagnostics = c("none", "light", "full"),
  target_coords = NULL,
  diagnostic_sample_size_nonedge = 1000L,
  diagnostic_sample_size_stress = 500L,
  diagnostic_nonedge_seed = 1L,
  diagnostic_stress_seed = 1L,
  metric = c("hop", "edge_length"),
  metric_neighbor_cap = NULL,
  length_normalization = c("median", "mean", "none")
)
```

Arguments

edges	Two-column integer matrix of edges (1-based vertex ids).
n	Number of vertices.
adj_list	Adjacency list (1-based) for undirected graphs.
weight_list	Parallel list of edge lengths for adj_list. The edge-length-metric engine requires it; in the hop-metric engine, NULL treats all edges as length 1. All supplied lengths must be finite and strictly positive.
edge_weights	Vector of edge lengths for edges, in the same order as its rows. The edge-length-metric engine requires it; in the hop-metric engine, NULL treats all edges as length 1. All supplied lengths must be finite and strictly positive. The selected engine determines whether lengths also define graph distances.
dim	Layout dimension (2 or 3). Default is 3.
placement	Initial placement strategy. "circle" is only used for 2D.
preset	Optional tuning preset. NULL uses the quality-first defaults. "carpet" applies a preset tuned for Sierpinski-carpet-like graphs and validated on carpet levels 3 and 4. "mesh" applies a preset tuned for rectangular lattice graphs and validated on 8x8 and 12x12 mesh layouts. "torus" applies a preset tuned for 3D torus layouts and validated on torus sizes from 8x8 through 20x20. "tree" applies a preset tuned for symmetric force-directed layouts of tree-like graphs and validated on binary trees of depths 5 and 6. Presets only fill in tuning arguments that you did not supply explicitly.
rounds	Initial rounds for refinement.
final_rounds	Final rounds for refinement.
num_init	Number of initial vertices in the coarsest level.
num_nbrs	Maximum number of graph-distance neighbors retained for local refinement at each filtration level.

`r` Main local temperature adaptation rate in `[0, 1]`.

`s` Non-negative boost factor applied when successive displacements have a consistent direction.

`repulsion_factor`
Non-negative multiplier applied to GRIP's finest-level repulsive force scale.

`coarse_repulsion_factor`
Non-negative multiplier applied to the extra coarse-level active-set repulsion term. `0` disables that extra term.

`coarse_repulsion_sample`
Positive integer sample size used to approximate active-set-wide repulsion on larger coarse levels.

`coarse_repulsion_exact_below`
Positive integer threshold. When the active set size is at most this value, the coarse repulsion is computed exactly against all currently active vertices instead of being sampled.

`final_anchor_factor`
Non-negative multiplier for an anchor term that pulls the final FR stage back toward the pre-final full-graph layout. '0' disables the anchor and preserves the current behavior.

`final_move_scale_after_first`
Scalar in '[0, 1]' applied to the final FR displacement after the first finest-level round. Values below '1' damp later full-graph movement while keeping the first FR round unchanged.

`final_mode` Final full-graph refinement mode. `"fr"` keeps the current Fruchterman-Reingold-style final stage. `"kk_repulse"` uses a KK-style local distance-matching update with explicit active-set repulsion instead of the final FR phase.

`insertion_anchor_count`
Positive integer number of anchor vertices used during multiscale insertion on non-initial MISF refinement levels. This is the closest current implementation to a global `K_mish` parameter.

`insertion_anchor_scope`
Anchor-eligibility rule used during multiscale insertion. `"any_higher"` matches the historical GRIP behavior and allows anchors from any already placed higher MISF level. `"prev_misf"` restricts anchors to the immediately previous MISF level only.

`insertion_anchor_strategy`
Anchor-selection rule used during multiscale insertion. `"first"` keeps the historical first-anchors-found BFS behavior. `"distance_band"` keeps exploring until the `K_mish`-th anchor distance band is exhausted, then places the new vertex from that less order-sensitive anchor pool. `"balanced_band"` uses the same band expansion, then explicitly selects a subset whose centroid stays centered in the candidate cloud while remaining geometrically spread out. `"spread_prev"` is a symmetry-oriented band strategy intended to be paired with `insertion_anchor_scope = "prev_misf"`; it selects anchors with broad angular and geometric coverage before placement.

`level0_insertion_mode`
: Level-0 insertion placement override used only when the finest filtration level is first populated. `"inherit"` keeps the current GRIP behavior. `"barycenter"` disables the 2D circle heuristic at level 0 and uses barycentric anchor placement. `"least_squares"` uses a multi-anchor least-squares distance fit at level 0 before any local micro-polish.

`level0_anchor_count`
: Positive integer number of already placed anchors to collect for level-0 insertion experiments. By default this inherits `insertion_anchor_count`. The legacy behavior uses 3.

`level0_local_kk_steps`
: Non-negative integer number of tiny local KK micro-polish steps applied immediately after each level-0 insertion. The legacy behavior uses 3.

`lgkk_polish_rounds`
: Non-negative integer number of experimental landmark-geodesic KK polish iterations applied after the main GRIP solve. `0` disables the polish.

`lgkk_multiscale_rounds`
: Non-negative integer number of compiled landmark-geodesic KK refinement rounds applied inside the multiscale solver after each eligible MISF level completes its standard GRIP rounds. This legacy shared budget is used as a fallback when any of the more specific per-stage budgets below are left `NULL`.

`lgkk_rounds_coarse`
: Optional non-negative integer number of compiled LGKK rounds applied on coarse MISF levels with `misf_level > 1`. When `NULL`, this falls back to `lgkk_multiscale_rounds`.

`lgkk_rounds_pre_final`
: Optional non-negative integer number of compiled LGKK rounds applied on the last coarse level just before the full graph is opened (`misf_level == 1`). When `NULL`, this falls back to `lgkk_multiscale_rounds`.

`lgkk_rounds_final`
: Optional non-negative integer number of compiled LGKK rounds applied after the full graph level completes its standard GRIP rounds (`misf_level == 0`). When `NULL`, this falls back to `lgkk_multiscale_rounds`.

`lgkk_local_nbrs`
: Number of nearest graph-metric neighbors retained per vertex in the LGKK sparse local set when either LGKK stage is enabled.

`lgkk_landmark_count`
: Number of farthest-point landmarks retained per vertex in the LGKK sparse long-range set when either LGKK stage is enabled.

`lgkk_multiscale_scope`
: Scope for the compiled multiscale LGKK stage. `"all"` applies it after every eligible MISF level, including the final full-graph level. `"coarse"` applies it only on coarse levels.

`lgkk_active_limit`
: Positive integer upper bound on the active-set size for compiled multiscale LGKK cache construction. Levels larger than this skip the multiscale LGKK stage.

`tinit_factor` Initial temperature factor.

`seed` Optional RNG seed for reproducibility. If NULL, uses current time.

`trace` Snapshot granularity. `"round"` records the coarsest initialization, each level start, every `trace.every` completed rounds, and the final layout. `"level"` records the coarsest initialization, every `trace.everyth` level start, and the final layout.

`trace.every` Positive integer thinning factor for recorded rounds or levels. Initial and final snapshots are always included.

`diagnostics` Optional per-frame diagnostic mode. `"none"` skips extra scoring, `"light"` appends lightweight shape diagnostics, and `"full"` also computes sampled stress on each traced frame.

`target_coords` Optional numeric target coordinate matrix used to append per-frame Procrustes RMSE diagnostics. It must have `n` rows and `dim` columns.

`diagnostic_sample_size_nonedge`
Positive integer sample size used for per-frame non-edge separation diagnostics when `diagnostics != "none"`.

`diagnostic_sample_size_stress`
Positive integer sample size used for per-frame sampled stress when `diagnostics = "full"`.

`diagnostic_nonedge_seed`
RNG seed base used for per-frame non-edge separation diagnostics.

`diagnostic_stress_seed`
RNG seed base used for per-frame sampled stress diagnostics.

`metric` Graph metric traced by the multiscale engine: `"hop"` (default) or `"edge_length"`. See `grip()` for the exact role and normalization of supplied edge lengths in each mode.

`metric_neighbor_cap`
Weighted-metric search limit used only when `metric = "edge_length"`. `NULL` performs the exact weighted neighborhood search and stops once the required neighbors and anchors are filled. A positive integer enables an approximate search by limiting the number of settled vertices per search. It is an error to supply this argument with `metric = "hop"`.

`length_normalization`
Global normalization applied only when `metric = "edge_length"`: `"median"` (default) divides every edge length by their median, `"mean"` divides by their mean, and `"none"` preserves the supplied numerical scale. It is an error to supply this argument with `metric = "hop"`.

Value

A list containing the final layout, recorded coordinate frames, frame metadata, trace settings, canonical stage data, and any requested diagnostics.

Examples

```
edges <- cbind(1:5, 2:6)
tr <- trace.grip(
  edges, n = 6, metric = "hop", dim = 2,
```

```
  rounds = 3, final_rounds = 2, num_init = 3,
  trace = "level", diagnostics = "light", seed = 1
)
tr$meta
```

`trace.legacy.grip` *Compute a trace for the legacy GRIP layout*

Description

Compute a trace for the legacy GRIP layout

Usage

```
trace.legacy.grip(
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  dim = 3,
  placement = c("barycenter", "circle"),
  preset = NULL,
  rounds = 20,
  final_rounds = 25,
  num_init = 36,
  num_nbrs = 10,
  r = 0.15,
  s = 3,
  repulsion_factor = 1,
  tinit_factor = 6,
  seed = 6,
  trace = c("round", "level"),
  trace.every = 1
)
```

Arguments

`edges`	Two-column integer matrix of edges (1-based vertex ids).
`n`	Number of vertices.
`adj_list`	Adjacency list (1-based) for undirected graphs.
`weight_list`	Optional parallel list of edge weights (edge lengths). If NULL, all edges are treated as weight 1. All weights must be finite and strictly positive.
`edge_weights`	Optional vector of edge weights for `edges`. All weights must be finite and strictly positive.
`dim`	Layout dimension (2 or 3). Default is 3.

placement	Initial placement strategy. "circle" is only used for 2D.
preset	Optional tuning preset. NULL uses the historical defaults. "carpet" applies a preset tuned for Sierpinski-carpet-like graphs and validated on carpet levels 3 and 4. "mesh" applies a preset tuned for rectangular lattice graphs and validated on 8x8 and 12x12 mesh layouts. "torus" applies a preset tuned for 3D torus layouts and validated on torus sizes from 8x8 through 20x20. "tree" applies a preset tuned for symmetric force-directed layouts of tree-like graphs and validated on binary trees of depths 5 and 6. Presets only fill in tuning arguments that you did not supply explicitly.
rounds	Initial rounds for refinement.
final_rounds	Final rounds for refinement.
num_init	Number of initial vertices in the coarsest level.
num_nbrs	Maximum number of graph-distance neighbors retained for local refinement at each filtration level.
r	Main local temperature adaptation rate in [0, 1].
s	Non-negative boost factor applied when successive displacements have a consistent direction.
repulsion_factor	Non-negative multiplier applied to GRIP's finest-level repulsive force scale. 1 keeps the historical repulsion strength; 0 disables that repulsive term.
tinit_factor	Initial temperature factor.
seed	Optional RNG seed for reproducibility. If NULL, uses current time.
trace	Snapshot granularity. "round" records the coarsest initialization, each level start, every trace.every completed rounds, and the final layout. "level" records the coarsest initialization, every trace.everyth level start, and the final layout.
trace.every	Positive integer thinning factor for recorded rounds or levels. Initial and final snapshots are always included.

Value

A list with final, frames, meta, trace, and trace.every. final is the final coordinate matrix. frames is a list of coordinate matrices with NA rows for vertices that have not yet been introduced by GRIP. meta is a data frame describing each frame with columns frame, phase, level_index, misf_level, round_in_level, and active_vertices.

Examples

```
edges <- cbind(1:5, 2:6)
tr <- trace.legacy.grip(edges, n = 6, dim = 2,
                                placement = "barycenter",
                                rounds = 3, final_rounds = 2,
                                num_init = 3, num_nbrs = 4,
                                trace = "level",
                                trace.every = 1,
                                seed = 1)
tr$meta
```

triangle_mask_helpers *Triangle mask helpers for recursive gasket families*

Description

Convenience constructors for the four-slot triangle masks used by `edges.recursive.triangle.mask()` and `recursive.triangle.mask.surface.graph()`. The slots correspond to the three corner subtriangles and the central inverted subtriangle created by one barycentric refinement of an equilateral triangle.

Usage

```
mask.triangle.classic()

mask.triangle.bridge(missing = c("top", "left", "right"))
```

Arguments

`missing`	Which outer corner is omitted in `mask.triangle.bridge()`.

Details

The returned mask entries are ordered and named as `left`, `right`, `top`, and `center`.

Value

A named logical vector of length 4.

triangulated_annulus_surface_helpers *Weighted triangulated annulus surface helpers*

Description

Convenience helpers that triangulate a planar annulus by clipping a regular triangular lattice to the region between two concentric circles. The resulting graph is a triangulated manifold with boundary and a single hole, making it a useful complement to the closed triangulated-polyhedron families.

Usage

```
triangulated.annulus.surface.embedding(
  resolution = 12,
  outer_radius = 1,
  inner_radius = 0.45,
  surface = c("flat", "saddle", "paraboloid", "ripple", "folded"),
  amplitude = 0.6,
```

```
  freq_u = 1,
  freq_v = 1
)

triangulated.annulus.surface.graph(
  resolution = 12,
  outer_radius = 1,
  inner_radius = 0.45,
  surface = c("flat", "saddle", "paraboloid", "ripple", "folded"),
  amplitude = 0.6,
  freq_u = 1,
  freq_v = 1,
  normalize = c("median", "mean", "none")
)
```

Arguments

resolution	Positive lattice-resolution control. Larger values produce finer triangulations.
outer_radius	Positive outer annulus radius.
inner_radius	Positive inner annulus radius. Must be strictly smaller than outer_radius.
surface	Geometry family used for the 3D lift. One of "flat", "saddle", "paraboloid", "ripple", or "folded".
amplitude	Finite deformation amplitude.
freq_u	Positive ripple frequency in the first planar coordinate. Used only when surface = "ripple".
freq_v	Positive ripple frequency in the second planar coordinate. Used only when surface = "ripple".
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Details

triangulated.annulus.surface.embedding() returns the 3D coordinates of the clipped lattice vertices in the same vertex order as edges.triangulated.annulus(). triangulated.annulus.surface.graph() returns a reusable weighted-graph bundle with the annulus edges, induced edge weights, the 3D embedding, and the 2D annulus parameter coordinates.

Value

triangulated.annulus.surface.embedding() returns an n x 3 numeric matrix with columns x, y, and z.

triangulated.annulus.surface.graph() returns a list with components:

- edges: the triangulated-annulus edges,
- n: number of vertices,
- edge_weights: induced positive edge lengths,

- coords_surface: the 3D embedding,
- coords_param: the canonical 2D annulus coordinates,
- weight_scale: the normalization constant applied to the raw edge lengths,
- family: always "triangulated.annulus",
- surface: the chosen surface name,
- resolution: the lattice-resolution parameter,
- label: a human-readable family label.

`triangulated_pair_of_pants_surface_helpers` *Weighted triangulated pair-of-pants surface helpers*

Description

Convenience helpers that triangulate a planar pair-of-pants domain by clipping a regular triangular lattice to a disk with two interior circular holes. The resulting graph is a triangulated surface with three boundary components and a deterministic non-grid topology.

Usage

```
triangulated.pair.of.pants.surface.embedding(
  resolution = 12,
  outer_radius = 1.1,
  hole_radius = 0.24,
  hole_offset = 0.38,
  hole_height = 0.18,
  surface = c("flat", "saddle", "paraboloid", "ripple", "folded"),
  amplitude = 0.6,
  freq_u = 1,
  freq_v = 1
)

triangulated.pair.of.pants.surface.graph(
  resolution = 12,
  outer_radius = 1.1,
  hole_radius = 0.24,
  hole_offset = 0.38,
  hole_height = 0.18,
  surface = c("flat", "saddle", "paraboloid", "ripple", "folded"),
  amplitude = 0.6,
  freq_u = 1,
  freq_v = 1,
  normalize = c("median", "mean", "none")
)
```

Arguments

resolution	Positive lattice-resolution control. Larger values produce finer triangulations.
outer_radius	Positive radius of the outer boundary.
hole_radius	Positive radius of each interior hole.
hole_offset	Positive horizontal offset of the two hole centers from the vertical axis.
hole_height	Vertical coordinate shared by the two hole centers.
surface	Geometry family used for the 3D lift. One of "flat", "saddle", "paraboloid", "ripple", or "folded".
amplitude	Finite deformation amplitude.
freq_u	Positive ripple frequency in the first planar coordinate. Used only when surface = "ripple".
freq_v	Positive ripple frequency in the second planar coordinate. Used only when surface = "ripple".
normalize	Normalization applied to the induced edge lengths. One of "median", "mean", or "none".

Details

triangulated.pair.of.pants.surface.embedding() returns the 3D coordinates of the clipped lattice vertices in the same vertex order as edges.triangulated.pair.of.pants(). triangulated.pair.of.pants.surf returns a reusable weighted-graph bundle with the pair-of-pants edges, induced edge weights, 3D embedding, and the 2D parameter coordinates of the clipped domain.

Value

triangulated.pair.of.pants.surface.embedding() returns an n x 3 numeric matrix with columns x, y, and z.

triangulated.pair.of.pants.surface.graph() returns a list with components:

- edges: the triangulated pair-of-pants edges,
- n: number of vertices,
- edge_weights: induced positive edge lengths,
- coords_surface: the 3D embedding,
- coords_param: the canonical 2D pair-of-pants coordinates,
- weight_scale: the normalization constant applied to the raw edge lengths,
- family: always "triangulated.pair.of.pants",
- surface: the chosen surface name,
- resolution: the lattice-resolution parameter,
- label: a human-readable family label.

`triangulated_polyhedron_surface_helpers` *Weighted triangulated polyhedron surface helpers*

Description

Convenience helpers that take a triangular-face polyhedron, repeatedly split each face into four subtriangles, merge shared-edge vertices, and use the resulting triangulated-surface graph as a reusable benchmark family. The canonical geometry is a piecewise-flat triangulated surface, while the weighted variants can inflate, twist, or radially modulate that surface in $\mathbb{R}^3$.

Usage

```
triangulated.polyhedron.surface.embedding(
  base = c("tetrahedron", "octahedron", "icosahedron"),
  level = 1,
  surface = c("standard", "inflated", "twisted", "wavy"),
  amplitude = 0.25,
  freq = 2,
  twist = 0.6
)

triangulated.polyhedron.surface.graph(
  base = c("tetrahedron", "octahedron", "icosahedron"),
  level = 1,
  surface = c("standard", "inflated", "twisted", "wavy"),
  amplitude = 0.25,
  freq = 2,
  twist = 0.6,
  normalize = c("median", "mean", "none")
)
```

Arguments

`base`	Base polyhedron. One of `"tetrahedron"`, `"octahedron"`, or `"icosahedron"`.
`level`	Subdivision depth. `level = 0` returns the base triangulation, `level = 1` splits each face into four, and so on.
`surface`	Geometry family used for the 3D embedding. One of `"standard"`, `"inflated"`, `"twisted"`, or `"wavy"`.
`amplitude`	Finite deformation amplitude.
`freq`	Positive modulation frequency used only when `surface = "wavy"`.
`twist`	Finite twist strength used only when `surface = "twisted"`.
`normalize`	Normalization applied to the induced edge lengths. One of `"median"`, `"mean"`, or `"none"`.

Details

The current supported bases are the regular tetrahedron, octahedron, and icosahedron. These give closed triangulated surfaces with non-grid topology and a small number of extraordinary vertices, making them useful complements to the mesh-derived families already in the package.

Value

`triangulated.polyhedron.surface.embedding()` returns an `n x 3` numeric matrix with columns `x`, `y`, and `z`.

`triangulated.polyhedron.surface.graph()` returns a list with components:

- `edges`: the triangulated-surface edges,
- `n`: number of vertices,
- `edge_weights`: induced positive edge lengths,
- `coords_surface`: the 3D embedding,
- `coords_param`: the canonical piecewise-flat coordinates,
- `weight_scale`: the normalization constant applied to the raw edge lengths,
- `family`: always `"triangulated.polyhedron"`,
- `base`: the chosen base polyhedron,
- `surface`: the chosen surface name,
- `level`: the subdivision depth,
- `subdivision`: the linear face subdivision factor `2^level`,
- `label`: a human-readable family label.

`vicsek_surface_helpers` *Weighted Vicsek surface helpers*

Description

Convenience wrappers around `recursive.mask.grid.surface.*()` using the connected 3×3 axial-cross mask: center plus the four cardinal neighbors. This produces a mesh-derived fractal family with strong bottlenecks while remaining orthogonally connected at every level.

Usage

```
vicsek.surface.embedding(
  level = 2,
  surface = c("saddle", "paraboloid", "ripple"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  x_scale = 1,
```

```
  y_scale = 1
)

vicsek.surface.graph(
  level = 2,
  surface = c("saddle", "paraboloid", "ripple"),
  amplitude = 0.75,
  freq_u = 1,
  freq_v = 1,
  x_scale = 1,
  y_scale = 1,
  normalize = c("median", "mean", "none")
)
```

Arguments

`level`	Recursion depth. Must be at least `1`.
`surface`	Surface family used for the lift. One of `"saddle"`, `"paraboloid"`, or `"ripple"`.
`amplitude`	Finite numeric amplitude controlling the non-flat displacement.
`freq_u`	Positive ripple frequency in the horizontal parameter direction. Used only when `surface = "ripple"`.
`freq_v`	Positive ripple frequency in the vertical parameter direction. Used only when `surface = "ripple"`.
`x_scale`	Positive horizontal scaling of the parameter domain.
`y_scale`	Positive vertical scaling of the parameter domain.
`normalize`	Normalization applied to the induced edge lengths. One of `"median"`, `"mean"`, or `"none"`.

Details

'vicsek.surface.embedding()' returns the 3D coordinates of the occupied cells in the same vertex order as `edges.vicsek()`. 'vicsek.surface.graph()' returns a reusable weighted-graph bundle for the named Vicsek family.

Value

`vicsek.surface.embedding()` returns an `n x 3` numeric matrix with columns `x`, `y`, and `z`, where `n = 5^level`.

`vicsek.surface.graph()` returns the same components as `recursive.mask.grid.surface.graph()`, with `family` set to `"vicsek"` and a family-specific class and label.

`weighted.grip.nd` *Weighted GRIP layout in arbitrary dimensions*

Description

`weighted.grip.nd()` is an opt-in weighted GRIP layout backend for embeddings in dimensions `dim >= 2`. It is implemented beside the legacy 2D/3D GRIP code path so that existing weighted-GRIP entry points and their dimensionality checks are unchanged.

Usage

```
weighted.grip.nd(
  edges = NULL,
  n = NULL,
  adj_list = NULL,
  weight_list = NULL,
  edge_weights = NULL,
  dim = 3,
  preset = NULL,
  placement = c("barycenter", "circle"),
  rounds = 160,
  final_rounds = 256,
  num_init = NULL,
  num_nbrs = 24,
  r = 0.03,
  s = 6,
  repulsion_factor = 1.5,
  tinit_factor = 2,
  final_move_scale_after_first = 1,
  final_mode = c("fr", "kk_repulse"),
  metric_neighbor_cap = NULL,
  insertion_anchor_count = 3,
  insertion_anchor_scope = c("any_higher", "prev_misf"),
 insertion_anchor_strategy = c("first", "distance_band", "balanced_band", "spread_prev"),
  level0_insertion_mode = c("inherit", "barycenter", "least_squares"),
  level0_anchor_count = insertion_anchor_count,
  level0_local_kk_steps = 3,
  length_normalization = c("median", "mean", "none"),
  disconnected = c("components", "error"),
  seed = 6
)
```

Arguments

`edges`	Two-column integer matrix of edges (1-based vertex ids).
`n`	Number of vertices.

`adj_list` Adjacency list (1-based) for an undirected graph.

`weight_list` Parallel list of strictly positive edge lengths.

`edge_weights` Optional vector of edge lengths for `edges`.

`dim` Embedding dimension. Must be at least 2.

`preset` Optional weighted layout preset: `"carpet"`, `"mesh"`, `"cylinder"`, `"torus"`, `"sphere"`, `"irregular"`, or `"tree"`.

`placement` Initial insertion placement strategy. `"circle"` is only available for `dim = 2`; higher dimensions use `"barycenter"`.

`rounds` Number of weighted refinement rounds before the final phase.

`final_rounds` Number of final weighted refinement rounds.

`num_init` Coarsest-level size control. Defaults to at least `dim + 1`.

`num_nbrs` Local-neighborhood control used by the ND backend.

`r` Movement-rate parameter in `[0, 1]`.

`s` Repulsion scale parameter.

`repulsion_factor` Non-edge repulsion multiplier.

`tinit_factor` Initial spread multiplier.

`final_move_scale_after_first` Final-stage FR displacement multiplier applied after the first final round. Must be in `[0, 1]`.

`final_mode` Final refinement mode: `"fr"` or `"kk_repulse"`.

`metric_neighbor_cap` Optional cap on the number of settled Dijkstra vertices used when building weighted neighborhood caches for inserted vertices. `NULL` keeps the exact weighted neighborhood search.

`insertion_anchor_count` Number of already placed anchor vertices used when inserting non-final MISF levels.

`insertion_anchor_scope` Anchor eligibility rule: `"any_higher"` or `"prev_misf"`.

`insertion_anchor_strategy` Anchor selection rule: `"first"`, `"distance_band"`, `"balanced_band"`, or `"spread_prev"`.

`level0_insertion_mode` Level-0 insertion placement override: `"inherit"`, `"barycenter"`, or `"least_squares"`.

`level0_anchor_count` Number of anchors used during level-0 insertion.

`level0_local_kk_steps` Number of local weighted-KK polish steps used immediately after level-0 insertion.

`length_normalization` Global edge-length normalization: `"median"` (default), `"mean"`, or `"none"`.

`disconnected` How to handle disconnected graphs: `"components"` lays out each component separately and packs them; `"error"` rejects disconnected graphs.

`seed` Optional RNG seed for reproducibility. If NULL, uses current time.

Value

Numeric matrix with one row per vertex and `dim` columns.

Index