Package 'grip'

November 8, 2026

Type Package

Title Graph Drawing with Intelligent Placement (GRIP)

Version 0.3.0

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>. Includes derived graph datasets from the SuiteSparse Matrix Collection, with source attribution and transformation metadata.

License GPL (>= 3)

Copyright See inst/COPYRIGHTS for software and dataset attribution. Bundled SuiteSparse-derived graph data are distributed under CC BY 4.0; see inst/extdata/suitesparse/PROVENANCE.md.

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, dgraphs (>= 0.4.0), FNN, geometry, igraph, htmltools, htmlwidgets, httpuv, knitr, later, plotly, rmarkdown, rgl, shiny, smacof (>= 2.1-7), testthat (>= 3.0.0), withr

Enhances ivue (>= 0.1.0)

Config/Needs/website pgajer/ivue@70ce17bb73f315da7565ce425405369c6a8edd48, pgajer/linf

LinkingTo Rcpp

SystemRequirements C++17

Config/testthat/edition 3

VignetteBuilder knitr

Config/roxygen2/version 8.0.0

NeedsCompilation yes

Author Pawel Gajer [aut, cre],
Jonathan Zheng [ctb, cph] (Author and copyright holder of MIT-licensed s_gd2 code adapted for the metric-MDS SGD backend),
Daniel Hangan [ctb] (Author of the SGD-MDS implementation in his scikit-learn fork adapted for the metric-MDS SGD backend),
The scikit-learn developers [cph] (Copyright holders of BSD-3-Clause code adapted from Daniel Hangan's scikit-learn fork for the metric-MDS SGD backend)

Maintainer Pawel Gajer <pgajer@gmail.com>

Repository CRAN

Date/Publication 2026-11-08 21:20:19 UTC

Contents

grip-package . . . 4
build.misf . . . 6
build.weighted.misf . . . 8
classical.mds . . . 9
compare.layouts . . . 10
cube_mask_pattern_helpers . . . 13
cylinder_surface_helpers . . . 14
edge.kk . . . 15
edge.length.density.stiffness . . . 18
edge.repulsive.stage . . . 19
edge.repulsive.state . . . 21
estimate.preparation . . . 23
geodesic.kk . . . 24
geometry.diagnostics . . . 26
globalrep.grip . . . 27
globalrep.weighted.grip . . . 31
graph.riemannian.star.structure . . . 35
graph_generators . . . 37
grip . . . 39
grip-0.2-migration . . . 44
grip-mds-migration . . . 46
gripui.app . . . 47
gripui.family.app . . . 48
gripui.graph.family.catalog . . . 48
gripui.project . . . 49
gripui.project.from.compare . . . 50
gripui.project.from.dir . . . 51
gripui.validate.project . . . 52

hmp.gc . . . 53
hmp.u01.gc.coarse . . . 54
irregular_annulus_surface_helpers . . . 55
irregular_ball_solid_helpers . . . 57
irregular_double_torus_surface_helpers . . . 58
irregular_pair_of_pants_surface_helpers . . . 60
irregular_rectangle_surface_helpers . . . 62
irregular_shell_solid_helpers . . . 64
irregular_sphere_surface_helpers . . . 65
irregular_torus_surface_helpers . . . 66
kary_tree_weighted_graph_helpers . . . 68
kernel.gram.gkk . . . 70
landmark.geodesic.kk . . . 72
landmark.mds.constraints . . . 74
layout.coords . . . 75
legacy.grip . . . 77
mask_pattern_helpers . . . 79
menger_sponge_surface_helpers . . . 80
mesh_surface_helpers . . . 81
metric.mds . . . 82
misf.geodesic.kk . . . 90
occupied_mesh_surface_helpers . . . 94
params.from.summary . . . 96
perforated_grid_helpers . . . 96
plot.layout . . . 98
porous_cube_surface_helpers . . . 99
prepare.edge.kk . . . 102
prepare.geodesic.kk . . . 103
prepare.graph.geodesic.mds . . . 104
prepare.landmark.geodesic.kk . . . 105
prepare.misf.geodesic.kk . . . 106
prepare.sparse.mds . . . 108
print.grip_gmds_layout . . . 109
print.grip_misf . . . 110
project.3d . . . 111
recursive_cube_mask_surface_helpers . . . 111
recursive_mask_grid_surface_helpers . . . 113
recursive_tetrahedron_mask_surface_helpers . . . 115
recursive_triangle_mask_surface_helpers . . . 116
repulsive.stage . . . 118
repulsive.state . . . 119
run.gripui . . . 120
run.gripui.family . . . 121
sampled_rectangle_surface_helpers . . . 122
score.coordinates . . . 125
score.geodesic.kk . . . 126
score.gmds . . . 128
score.landmark.geodesic.kk . . . 130

score.layout . . . 132
score.misf.geodesic.kk . . . 133
score.surface . . . 135
sierpinski_carpet_surface_helpers . . . 136
sierpinski_tetrahedron_surface_helpers . . . 137
sierpinski_triangle_surface_helpers . . . 138
sphere_surface_helpers . . . 139
suitesparse.graph . . . 141
tetrahedron_mask_helpers . . . 142
torus_surface_helpers . . . 143
trace.grip . . . 145
trace.legacy.grip . . . 150
triangle_mask_helpers . . . 152
triangulated_annulus_surface_helpers . . . 153
triangulated_pair_of_pants_surface_helpers . . . 154
triangulated_polyhedron_surface_helpers . . . 156
vicsek_surface_helpers . . . 157
weighted.grip.nd . . . 158

Index **161**

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

Description

Fast multiscale graph layouts in 2D and 3D.

Details

Use `grip()` to draw a graph, then `score.layout()` to assess the result.

Start here

Open `vignette("grip-examples", package = "grip")` for a small complete workflow, or `vignette("function-guide", package = "grip")` to choose functions by task. The example below draws a 20-vertex mesh and prints sampled graph-distance stress: zero means perfect agreement on the sampled pairs after fitting a common scale, not a certificate for every pair.

Use `metric = "hop"` for connectivity or `metric = "edge_length"` when positive lengths define the graph metric. See `vignette("weighted-grip-intro", package = "grip")`.

Find the next workflow

Choose settings and seeds `vignette("grip-real-data", package = "grip")`.

Trace the construction `vignette("grip-trace-and-diagnostics", package = "grip")`.

Construct graph examples `vignette("synthetic-graph-families", package = "grip")`.

List available guides with `vignette(package = "grip")`. Development installations from GitHub include these guides only when built with `build_vignettes = TRUE`. The website also provides rendered guides, the graph gallery, comparisons, case study, and explorer walkthrough from the same installed vignette sources. Check `packageVersion("grip")` when using development documentation.

Advanced refinement

The public experimental edge-KK and geodesic-KK helpers provide weighted-layout evaluation and refinement. Choose these through the function guide after trying the main layout, scoring, and tracing workflows.

Graph input validation

Vertex indices and counts must be finite integers within the R integer range. Adjacency must be reciprocal with matching lengths and multiplicity. Self-loops are rejected; remove them explicitly. Supply either `edges`/`edge.weights` or `adj.list`/`weight.list`, not both. Functions accepting a `prepared` graph reject additional raw graph inputs; an explicit `n` must match the stored vertex count. Rebuild the preparation when changing topology, vertex order, or edge lengths.

GMDS thread control

The internal compiled GMDS optimizer, also used by the GMDS exploration apps, uses at most two CPU threads. With automatic selection (`n.threads = 0`), `GRIP_GMDS_THREADS` can be set to `1` or `2` to choose the thread count before hardware concurrency is consulted. An explicit positive `n.threads` overrides the environment variable; values above two are capped at two. Non-positive or nonnumeric environment values fall back to hardware-based selection. The R optimization engine is serial. This setting is not a package-wide control for GRIP layouts or for third-party numerical libraries.

Author(s)

Maintainer: Pawel Gajer <pgajer@gmail.com>

Authors:

- Pawel Gajer <pgajer@gmail.com>

Other contributors:

- Jonathan Zheng (Author and copyright holder of MIT-licensed s_gd2 code adapted for the metric-MDS SGD backend) [contributor, copyright holder]
- Daniel Hangan (Author of the SGD-MDS implementation in his scikit-learn fork adapted for the metric-MDS SGD backend) [contributor]
- The scikit-learn developers (Copyright holders of BSD-3-Clause code adapted from Daniel Hangan's scikit-learn fork for the metric-MDS SGD backend) [copyright holder]

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

Examples

```
edges <- edges.mesh(4, 5)
coords <- grip(edges, n = 20, dim = 2, preset = "mesh", seed = 11)
plot.layout(coords, edges = edges, pch = 16, main = "A 4 by 5 mesh")
quality <- score.layout(coords, edges = edges, n = 20,
                        sample.size.stress = 200, stress.seed = 11,
                        edge.crossings = "never")
quality[, c("sampled.stress", "edge.length.cv")]
```

`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`.

Examples

```
cycle_edges <- edges.cycle(6)
cycle_lengths <- seq(0.8, 1.3, length.out = nrow(cycle_edges))
levels <- build.weighted.misf(
  edges = cycle_edges, n = 6, edge.weights = cycle_lengths,
  num.init = 3, num.nbrs = 2, seed = 1
)
```

`classical.mds` *Classical-MDS baseline layout for GMDS experiments*

Description

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

Usage

```
classical.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 classical MDS requires a graph distance matrix.
`edges`	Two-column integer edge matrix used when `prepared` is omitted. Supply either edges/edge.weights or adj.list/weight.list, not both. Raw graph inputs cannot be combined with a prepared object.
`n`	Finite positive integer vertex count. When supplied with a prepared object, it must match the stored graph size.
`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, classical.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.

Workflow guides

Start with vignette("function-guide", package = "grip") to choose a layout, diagnostic, or reference comparison. List installed guides with vignette(package = "grip").

Examples

```
cycle_edges <- edges.cycle(6)
prepared <- prepare.graph.geodesic.mds(cycle_edges, n = 6)
fit <- classical.mds(prepared = prepared, dim = 2)
```

---

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()`.

Details

Procrustes stability is the mean pairwise root-mean-square vertex distance after centering each layout, scaling it to unit maximum radius, and applying the optimal orthogonal rotation or reflection. The default composite score converts each available summary metric to average ranks on $[0, 1]$, reverses ranks for higher-is-better metrics, applies the documented weights, and renormalizes the weights after omitting unavailable metrics. Lower composite scores are better.

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 `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.

Examples

```
cavities <- mask.cube.asymmetric.cavities(
  side = 5, cavity.size = 1, pocket.size = 1
)
channels <- mask.cube.channel.network(
  side = 5, channel.width = 1, branch.offset = 2
)
tunnels <- mask.cube.periodic.tunnels(
  side = 5, tunnel.width = 1, tunnel.period = 2, tunnel.offset = 1
)
```

---

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.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

The coords.surface component contains 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

The coords_surface component is 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.

Examples

```
cylinder <- cylinder.surface.graph(4, 5, surface = "hourglass")
```

---

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

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

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("classical_mds", "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 integer edge matrix used when `prepared` is omitted. Supply either edges/edge.weights or adj.list/weight.list, not both. Raw graph inputs cannot be combined with a prepared object.

`n` Finite positive integer vertex count. When supplied with a prepared object, it must match the stored graph size.

`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. `"classical_mds"` (the default) uses `classical.mds()`. `"metric_mds"` uses stress-minimizing `metric.mds()` with its default native SGD backend. Both require 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. Former underscore argument names in this list are also accepted for the transition release; duplicate old/current spellings are rejected.

`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.

`...` Former underscore spellings of this function's dotted arguments, accepted for one transition release without deprecation warnings (for example, `adj_list`, `scale_mode`, and `max_iter`). Use the dotted names in new code. Supplying both spellings or unknown arguments errors.

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` is `"weighted_grip"` or `"random"`. Use `init = "weighted_grip"` for a scalable weighted-GRIP warm start followed by edge-KK polish. If `coords` is omitted and either MDS initializer is requested, 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"`.

Examples

```
cycle_edges <- edges.cycle(6)
theta <- seq(0, 2 * pi, length.out = 7)[-7]
initial <- cbind(cos(theta), sin(theta))
prepared <- prepare.edge.kk(cycle_edges, n = 6)
fit <- edge.kk(
  coords = initial, prepared = prepared,
  max.iter = 1, density.n = 32, engine = "R"
)
```

---

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.

Examples

```
stiffness <- edge.length.density.stiffness(
  c(0.8, 0.9, 1, 1.1, 1.2, 1.3), density.n = 32
)
```

---

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.

Examples

```
cycle_edges <- edges.cycle(6)
theta <- seq(0, 2 * pi, length.out = 7)[-7]
initial <- cbind(cos(theta), sin(theta))
fit <- edge.repulsive.stage(
  initial, cycle_edges, rep(1, nrow(cycle_edges)),
  lambda = 0.1, max.iter = 1, engine = "R"
)
```

---

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.

Examples

```
cycle_edges <- edges.cycle(6)
theta <- seq(0, 2 * pi, length.out = 7)[-7]
coords <- cbind(cos(theta), sin(theta))
state <- edge.repulsive.state(
  coords, cycle_edges, rep(1, nrow(cycle_edges)),
  lambda = 0.1, engine = "R"
)
```

`estimate.preparation` *Estimate preparation storage before allocating graph distances*

Description

Reports a lower bound for dense distance storage, not peak memory or runtime. This arithmetic-only helper does not construct a graph or allocate its cache.

Usage

```
estimate.preparation(
  n,
  n.edges = NA_integer_,
 pair.mode = c("all_pairs", "landmark_sparse", "distance_matrix_only", "edge_only"),
  local.nbrs = 20L,
  landmark.count = 8L
)
```

Arguments

`n`	Number of vertices.
`n.edges`	Number of undirected edges, or NA if unknown.
`pair.mode`	Preparation mode: full paths, landmark paths, distances only, or edges only.
`local.nbrs, landmark.count`	Counts used by landmark preparation.

Details

One dense double matrix needs `8 * n^2` bytes. Full and landmark preparations currently compute dense distances; landmark mode reduces retained pairs, not that matrix. Trees, routes, copies, R object overhead, and solver workspace require additional memory and are not estimated. Edge-only preparation has no dense distance matrix, but still stores the graph. Use `prepare.edge.kk()` when only edge constraints are needed, or ordinary `grip()` when a full geodesic cache is unnecessary.

Dense preparation warns before its searches/allocations when this bound exceeds `getOption("grip.preparation.warn.by` `512 * 1024^2)`. Set that option to another positive byte count, or `Inf` to acknowledge and disable the advisory warning. It is not a hard memory limit. Small calls are quiet. Native refinement checks for cancellation between rounds or optimizer evaluations; worker threads finish an evaluation before interruption is raised on the main thread. Cancellation does not return a partial fit.

Value

A one-row data frame containing graph counts, pair mode, a pair-count upper bound, and the dense-distance storage lower bound in bytes and GiB.

See Also

`prepare.graph.geodesic.mds()`, `prepare.landmark.geodesic.kk()`

Examples

```
estimate.preparation(10000, n.edges = 20000)
estimate.preparation(10000, n.edges = 20000, pair.mode = "edge_only")
```

`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.

Examples

```
cycle_edges <- edges.cycle(6)
theta <- seq(0, 2 * pi, length.out = 7)[-7]
initial <- cbind(cos(theta), sin(theta))
prepared <- prepare.geodesic.kk(cycle_edges, n = 6)
fit <- geodesic.kk(initial, prepared = prepared, max.iter = 1)
```

`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.

Examples

```
cycle_edges <- edges.cycle(6)
theta <- seq(0, 2 * pi, length.out = 7)[-7]
target <- cbind(cos(theta), sin(theta))
diagnostics <- geometry.diagnostics(
  target + 0.01, target, cycle_edges,
  sample.size.symmetry = 10, sample.size.wedges = 10
)
```

`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). Supply either edges/edge.weights or adj.list/weight.list, not both. Fractional, nonfinite, and out-of-range vertex ids are rejected before integer conversion.
n	Number of vertices, a finite positive integer.
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). Supply either edges/edge.weights or adj.list/weight.list, not both. Fractional, nonfinite, and out-of-range vertex ids are rejected before integer conversion.
n	Number of vertices, a finite positive integer.
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.

Examples

```
cycle_edges <- edges.cycle(6)
cycle_lengths <- seq(0.8, 1.3, length.out = nrow(cycle_edges))
layout <- globalrep.weighted.grip(
  edges = cycle_edges, n = 6, edge.weights = cycle_lengths,
  dim = 2, rounds = 1, final.rounds = 1,
  num.init = 3, num.nbrs = 2, seed = 1
)
```

`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.

Examples

```
cycle_edges <- edges.cycle(6)
theta <- seq(0, 2 * pi, length.out = 7)[-7]
coords <- cbind(cos(theta), sin(theta))
prepared <- prepare.graph.geodesic.mds(cycle_edges, n = 6)
star <- graph.riemannian.star.structure(X = coords, prepared = prepared)
```

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.cylinder(h, w = h)

edges.torus(h, w = h)

edges.cube(side = 2)

edges.kary.tree(k = 2, depth = 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.
`side`	Number of lattice points along each cube edge.
`k`	Branching factor.
`depth`	Number of levels below the root.
`level`	Recursion depth for the Sierpinski graph families. Must be at least 1.

Details

The exported generators cover primitive paths, cycles, regular meshes, cylinders, tori, cubes, trees, and the standard Sierpinski families. Specialized weighted benchmark families are exposed through complete `*.surface.graph()` and `*.solid.graph()` bundles instead of separate edge-list and embedding functions. `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.cylinder()`: Cylindrical grid graph with `h` rows and wrapped width `w`.
- `edges.torus()`: Toroidal grid graph with wrapped height and width.
- `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.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

```
cube_edges <- edges.cube(3)
cycle_edges <- edges.cycle(6)
cylinder_edges <- edges.cylinder(3, 4)
tree_edges <- edges.kary.tree(k = 2, depth = 2)
mesh_edges <- edges.mesh(3, 4)
carpet_edges <- edges.sierpinski.carpet(level = 1)
tetrahedron_edges <- edges.sierpinski.tetrahedron(level = 1)
torus_edges <- edges.torus(3, 4)
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). Supply either edges/edge.weights or adj.list/weight.list, not both. Fractional, nonfinite, and out-of-range vertex ids are rejected before integer conversion.
n	Number of vertices, a finite positive integer.
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"`.

`...`
: Former underscore spellings of this function's dotted arguments, accepted for one transition release without deprecation warnings (for example, `adj_list`, `weight_list`, and `final_rounds`). Use the dotted names in new code. Supplying both spellings or unknown arguments errors.

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.

Workflow guides

Start with `vignette("function-guide", package = "grip")` to choose a layout, diagnostic, or reference comparison. List installed guides with `vignette(package = "grip")`.

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-0.2-migration` *Migrate code to the grip 0.2 API*

Description

Version 0.2.0 removes the long-form compatibility aliases that were deprecated in the 0.1 series and narrows the exported graph-family API. The canonical replacements are listed below.

Removed aliases

- `grip.layout()` -> `grip()`
- `grip.layout.globalrep()` -> `globalrep.grip()`
- `grip.layout.globalrep.weighted()` -> `globalrep.weighted.grip()`

- `grip.layout.legacy()` -> `legacy.grip()`
- `grip.layout.trace()` -> `trace.grip()`
- `grip.layout.trace.legacy()` -> `trace.legacy.grip()`
- `grip.layout.trace.weighted()` -> `trace.grip(metric = "edge_length")`
- `grip.layout.weighted()` -> `grip(metric = "edge_length")`
- `grip.layout.weighted.nd()` -> `weighted.grip.nd()`
- `grip.plot()` -> `plot.layout()`
- `grip.project.3d()` -> `project.3d()`
- `grip.metric.mds.layout()` -> `classical.mds()` (classical scaling)
- `grip.optimize.kernel.gram.gkk.layout()` -> `kernel.gram.gkk()`
- `grip.score.gmds.layout()` -> `score.gmds()`
- `grip.gmds.layout.result()` -> `gmds.result()` (now internal)
- `grip.edge.length.density.stiffness()` -> `edge.length.density.stiffness()`
- `grip.prepare.landmark.geodesic.kk()` -> `prepare.landmark.geodesic.kk()`
- `grip.prepare.geodesic.kk()` -> `prepare.geodesic.kk()`
- `grip.prepare.graph.geodesic.mds()` -> `prepare.graph.geodesic.mds()`
- `grip.prepare.misf.geodesic.kk()` -> `prepare.misf.geodesic.kk()`
- `grip.prepare.edge.kk()` -> `prepare.edge.kk()`
- `grip.score.landmark.geodesic.kk()` -> `score.landmark.geodesic.kk()`
- `grip.score.geodesic.kk()` -> `score.geodesic.kk()`
- `grip.score.misf.geodesic.kk()` -> `score.misf.geodesic.kk()`
- `grip.optimize.landmark.geodesic.kk()` -> `landmark.geodesic.kk()`
- `grip.optimize.geodesic.kk()` -> `geodesic.kk()`
- `grip.optimize.misf.geodesic.kk()` -> `misf.geodesic.kk()`
- `grip.optimize.edge.kk.layout()` -> `edge.kk()`
- `grip.optimize.edge.gkk.layout()` -> `edge.kk()`
- `grip.optimize.edge.isometric.layout()` -> `edge.kk()`
- `grip.optimize.edge.repulsive.stage()` -> `edge.repulsive.stage()`
- `grip.optimize.repulsive.stage()` -> `repulsive.stage()`
- `grip.edge.repulsive.state()` -> `edge.repulsive.state()`
- `grip.repulsive.state()` -> `repulsive.state()`
- `grip.build.misf.weighted()` -> `build.weighted.misf()`
- `grip.build.misf()` -> `build.misf()`
- `grip.geometry.diagnostics()` -> `geometry.diagnostics()`
- `grip.params.from.summary()` -> `params.from.summary()`
- `grip.score.layout()` -> `score.layout()`
- `grip.compare.layouts()` -> `compare.layouts()`

Graph-family helpers

Complete `*.surface.graph()` and `*.solid.graph()` constructors remain public and return reusable bundles. Use the bundle's `edges`, `coords.surface`, and `coords.param` components instead of the former specialized edge-list, standalone embedding, and parameter-coordinate exports. Primitive generators such as `edges.path()`, `edges.cycle()`, `edges.mesh()`, and `edges.cube()` remain public.

---

grip-mds-migration *Migrating classical and metric stress MDS workflows*

Description

In grip through version 0.2.0, `metric.mds()` used `stats::cmdscale()`. From version 0.2.0.9000, that algorithm is named `classical.mds()`, and `metric.mds()` minimizes raw distance stress, using native SGD by default. This is an intentional behavioral change rather than a deprecated alias.

Preserve an existing analysis

Replace `metric.mds(...)` with `classical.mds(...)`, and explicit `init = "metric_mds"` with `init = "classical_mds"`. Both `edge.kk()` and `kernel.gram.gkk()` default to classical initialization, preserving their previous default algorithm without requiring smacof. `add` and `eig` are classical-scaling arguments and are accepted only by `classical.mds()`. Existing saved coordinates, figure labels, and cache fields named `metric_mds` from older versions remain classical results. Renaming a call does not require recomputing those fits.

Request stress minimization

Use `metric.mds(...)` or `edge.kk(init = "metric_mds", ...)` to use the default native SGD backend. For control over multiple starts and budgets, call `metric.mds()` first and pass `coords = fit$coords` to `edge.kk()`. Check `metadata$starts`, `metadata$termination`, and the independently calculated stress diagnostics. Both backends are local optimizers; an iteration limit or a finite result does not certify an optimum.

Preserve a previous SMACOF analysis

Install the optional smacof package and explicitly request `metric.mds(backend = "smacof", ...)`. The `eps` tolerance belongs to that backend and must not be supplied to SGD. To retain a SMACOF initializer in a refinement workflow, compute it explicitly and pass its coordinates to `edge.kk()` or `kernel.gram.gkk()`. No automatic backend fallback is made.

Native SGD backend

`metric.mds(backend = "sgd", max.iter = 30, ...)` explicitly selects the default backend with a short trial budget, without requiring smacof. SGD uses the same uniform pair weights and shortest-path targets. Its controls remain provisional calibration choices, and `max.iter` counts complete pair passes. Schedule completion is reported as `iteration_limit`, not convergence. Inspect per-start records and `metadata$sgd` checkpoint histories. Selecting `init = "metric_mds"` in refinement methods follows the default SGD backend; their default `init = "classical_mds"` is unchanged. A different optimizer cannot repair inaccurate graph distances.

Reproducibility

Raw stress is measured against the original graph-distance units after restoring scale from the backend. Target-normalized raw stress and scale-profiled Stress-1 have the same optimal shapes when scale is free and pair weights match, but literal Stress-1 at the returned raw-stress scale is a different diagnostic value. Record the algorithm, objective, backend/package versions, graph identity, scale policy, and start settings in new result manifests. Recompute downstream refinement when changing its initializer, and use new cache identities rather than replacing historical classical results in place.

---

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.gc *HMP Illumina 16S Giant-Component Graph*

Description

A symmetric k-nearest-neighbor graph constructed from samples from the Ravel-led University of Maryland Baltimore Human Microbiome Project (UMB-HMP) longitudinal vaginal cohort, generated by Illumina 16S rRNA amplicon sequencing. Rows without an explicit HMP project label, rows from the related U01 cohort, and legacy 454 runs are excluded before preprocessing.

Usage

```
hmp.gc
```

Format

A named list with components:

adj_list Adjacency list with 1-based integer neighbor indices.

weight_list Parallel list of Euclidean edge lengths in the ten-component PCA representation.

vertex_data Data frame with one row per HMP sample and columns for sample identifier, HMP project, sequencing platform and phase, CST and subCST annotations, and retained-read diagnostics.

graph_info Named list recording the cohort filter, representation, graph constructor, selected k, and graph dimensions.

Details

Taxonomic count features detected in at least one percent of the retained HMP samples are converted to relative abundance and represented by ten principal components. A symmetric k = 3 nearest-neighbor graph is built in that representation, and the largest connected component is retained.

Source

Derived from Illumina 16S rRNA amplicon profiles from the Ravel-led UMB-HMP longitudinal vaginal cohort. Exact graph edges, sample metadata, feature screening results, and provenance are distributed under `inst/extdata/hmp_gc/`.

Examples

```
data(hmp.gc)
length(hmp.gc$adj_list)
hmp.gc$graph_info[c("assay", "graph_constructor", "selected_k")]
```

---

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% relative-abundance plus PCA branch 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.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

The coords_surface component is 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.

Examples

```
annulus <- irregular.annulus.surface.graph(rings = 3, outer.count = 12)
```

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.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

The `coords_surface` component is 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.

Examples

```
ball <- irregular.ball.solid.graph(
  base = "tetrahedron", level = 0, layers = 2
)
```

---

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.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

The coords_surface component is 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.

Examples

```
double_torus <- irregular.double.torus.surface.graph(
  slices = 7, tube.count = 8
)
```

---

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.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

The coords_surface component is 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.

Examples

```
pair_of_pants <- irregular.pair.of.pants.surface.graph(
  slices = 5, outer.count = 12
)
```

`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.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`.
`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"`.
`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]`.

`connectivity`
: Mesh neighborhood rule passed to `edges.mesh()`.

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

Details

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

Value

The `coords_param` component is an `n x 2` numeric matrix with columns `u` and `v`.

The `coords_surface` component is 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.

Examples

```
rectangle <- irregular.rectangle.surface.graph(4, 5, surface = "ripple")
```

`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.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

The `coords_surface` component is 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"`.

Examples

```
shell <- irregular.shell.solid.graph(
  base = "tetrahedron", level = 0, layers = 2
)
```

---

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.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

The coords_surface component is 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.

Examples

```
sphere <- irregular.sphere.surface.graph(bands = 3, equator.count = 12)
```

---

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.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

The coords_surface component is 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.

Examples

```
torus <- irregular.torus.surface.graph(major.rings = 4, tube.count = 8)
```

---

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.

Examples

```
tree <- kary.tree.weighted.graph(k = 2, depth = 2)
```

`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:

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

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("classical_mds", "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 integer edge matrix used when `prepared` is omitted. Supply either edges/edge.weights or adj.list/weight.list, not both. Raw graph inputs cannot be combined with a prepared object.
n	Finite positive integer vertex count. When supplied with a prepared object, it must match the stored graph size.
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. `"classical_mds"` uses classical scaling (the default), `"metric_mds"` uses stress MDS through the default native SGD backend, both from an all-pairs prepared object; `"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"`.

Examples

```
cycle_edges <- edges.cycle(6)
theta <- seq(0, 2 * pi, length.out = 7)[-7]
initial <- cbind(cos(theta), sin(theta))
prepared <- prepare.graph.geodesic.mds(cycle_edges, n = 6)
fit <- kernel.gram.gkk(
  coords = initial, prepared = prepared, X = initial,
  max.iter = 1, density.n = 32, engine = "R"
)
```

`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.

Examples

```
cycle_edges <- edges.cycle(6)
theta <- seq(0, 2 * pi, length.out = 7)[-7]
initial <- cbind(cos(theta), sin(theta))
prepared <- prepare.landmark.geodesic.kk(
  cycle_edges, n = 6, local.nbrs = 2, landmark.count = 2
)
fit <- landmark.geodesic.kk(initial, prepared = prepared, max.iter = 1)
```

---

landmark.mds.constraints

Construct sparse MDS targets from landmark distances

Description

Converts supplied distances into soft pairwise fitting targets for [metric.mds()]. No distances are recomputed and no graph paths are inferred. With ‘weighting = "uniform"‘, every retained unordered pair has endpoint multiplicities one, giving inverse-squared distance stress on those pairs. With ‘weighting = "region"‘, samples belong to their nearest landmark (ties use landmark row order). The update at a non-landmark endpoint represents the number of samples in the landmark’s region at most half the target distance from that landmark. Opposite landmark directions are combined. Optional local pairs replace landmark contributions for that pair with multiplicities one at each endpoint. Actual fitting weights divide these multiplicities by squared target distances. Region weighting can give asymmetric endpoint updates; it is not ordinary symmetric weighted stress.

Usage

```
landmark.mds.constraints(
  distance.matrix,
  landmarks,
  local.pairs = NULL,
  local.distances = NULL,
  weighting = c("uniform", "region")
)
```

Arguments

distance.matrix	Numeric landmark-by-sample distance matrix, finite and nonnegative. Each row has zero at its landmark and strictly positive values elsewhere. Consolidate zero-distance duplicates before use.
landmarks	Distinct integer sample indices identifying matrix rows, in row order; sample indices correspond to matrix columns.
local.pairs	Optional two-column integer matrix of sample indices.
local.distances	Positive finite distances for ‘local.pairs‘, from the same metric as ‘distance.matrix‘. Repeated pairs and overlapping landmark targets must agree to relative tolerance 1e-10.
weighting	Endpoint weighting scheme: ‘"uniform"‘ or ‘"region"‘.

Value

List with ‘pairs‘, ‘targets‘, ‘count_i‘, and ‘count_j‘, suitable for ‘metric.mds(n = ncol(distance.matrix), constraints = ..., approximation = "sparse")‘. Distances are retained in their supplied units.

Examples

```
X <- matrix(sin(seq_len(24)), 8, 3)
D <- as.matrix(dist(X))
C <- landmark.mds.constraints(D[c(1, 5), ], c(1, 5))
fit <- metric.mds(n = 8, constraints = C, approximation = "sparse",
                  dim = 3, max.iter = 5)
```

---

layout.coords *Extract coordinates from a layout result*

Description

Use one extraction step for ordinary layouts, traces, MDS fits, and supported refinement results. Existing return types and direct field access are unchanged.

Usage

```
layout.coords(x)
```

Arguments

x	A numeric coordinate matrix, a grip_layout_trace, a grip_gmds_layout, or a refinement list with the documented coords/trace/frames/prepared/score or coords/state/trace/frames structure.

Details

For traces the final frame is selected. For MDS and refinement fits the fitted coords are selected. A plain list containing only coords is not a recognized result; pass that matrix explicitly instead.

Keep graph vertex order when reusing coordinates. MDS result fields method, coords, diagnostics, and metadata remain directly available. Stopping information in metadata is method-dependent. Trace $final and refinement $coords remain supported. Prepared caches and internal state fields are version-dependent: rebuild them when the graph or package contract changes.

Value

The coordinate matrix, unchanged, including vertex order, dimensions, dimnames, and attributes. No projection, alignment, scaling, or scoring is performed. Malformed, ambiguous, or unsupported objects produce an error.

See Also

grip(), trace.grip(), classical.mds(), geodesic.kk()

Examples

```
edges <- edges.path(6)
fits <- list(grip(edges, n = 6, dim = 2, seed = 1),
             trace.grip(edges, n = 6, dim = 2, seed = 1),
             classical.mds(edges = edges, n = 6))
first <- layout.coords(fits[[1]])
coords <- lapply(fits, layout.coords)
plot.layout(coords[[1]], edges = edges)
vapply(coords, function(z) score.layout(z, edges = edges, n = 6,
    sample.size.stress = 15, stress.seed = 1,
    edge.crossings = "never")$sampled.stress, numeric(1))
```

`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). Supply either `edges`/`edge.weights` or `adj.list`/`weight.list`, not both. Fractional, nonfinite, and out-of-range vertex ids are rejected before integer conversion.
`n`	Number of vertices, a finite positive integer.
`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 `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.

Examples

```
asymmetric <- mask.asymmetric.holes(k = 5, hole.size = 1)
border <- mask.border(k = 5, thickness = 1)
corner <- mask.corner(k = 5, width = 2, corner = "top_left")
cross <- mask.cross(k = 5, arm.width = 1)
```

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.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

The `coords_surface` component is 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.

Examples

```
sponge <- menger.sponge.surface.graph(level = 1)
```

`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.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`	Mesh neighborhood rule. `"orthogonal"` keeps the four-neighbor grid; `"diagonal"` also adds both diagonals of every unit square.
`normalize`	Normalization applied to the induced edge lengths. One of `"median"`, `"mean"`, or `"none"`.

Details

The coords.surface component contains 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

The coords_surface component is 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.

Examples

```
mesh <- mesh.surface.graph(4, 5, surface = "saddle")
```

---

metric.mds *Metric stress MDS using stochastic gradient descent or SMACOF*

Description

metric.mds() fits Euclidean coordinates to graph distances. The default approximation = "full" minimizes raw stress over all shortest-path distances, using native stochastic gradient descent (SGD, the default) or smacof::mds(type = "ratio"). With approximation = "sparse", pivots and sparse SGD approximate long-range interactions without constructing an all-pairs distance matrix. Only the SMACOF backend requires the optional **smacof** package. Before version 0.2.0.9000, this name performed classical scaling; use classical.mds() to retain that behavior. The add and eig arguments belong to classical.mds() only.

Usage

```
metric.mds(
  prepared = NULL,
  edges = NULL,
  n = NULL,
  adj.list = NULL,
  weight.list = NULL,
  edge.weights = NULL,
  dim = 2L,
  init = c("classical", "random"),
  n.init = 1L,
  max.iter = 1000L,
  eps = 1e-08,
  seed = 1L,
  diagnostics = TRUE,
  scale.mode = c("profiled", "identity"),
  distance.floor = 1e-08,
  edge.length.epsilon = 1e-08,
  band.quantiles = c(1/3, 2/3),
  backend = c("sgd", "smacof"),
  sgd.control = list(),
  pair.weights = c("uniform", "inverse_squared"),
  approximation = c("full", "sparse"),
  sparse.control = list(),
  constraints = NULL,
  distance.matrix = NULL,
  resume = NULL
)
```

Arguments

prepared	An all-pairs prepared graph object containing distance.matrix, for full MDS only. Sparse MDS requires raw graph inputs. A prepare.sparse.mds() result is accepted with sparse approximation. Other edge-only preparations are not supported as input.
edges	Two-column integer edge matrix used when prepared is omitted. Supply either edges/edge.weights or adj.list/weight.list, not both. Raw graph inputs cannot be combined with a prepared object.
n	Finite positive integer vertex count. When supplied with a prepared object, it must match the stored graph size.
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	Embedding dimension. Full MDS supports integer dimensions of at least two and smaller than the vertex count. Sparse SGD supports integer dimensions of at least two. Both support 10D fits for subsequent PCA projection or initialization of a 3D fit.

`init` "classical" (full default), "random" (sparse default), or a finite numeric matrix with n rows and dim columns in input-distance units. Sparse MDS supports only random or supplied starts; random starts use a uniform unit cube in normalized-distance units.

`n.init` Positive integer number of starts, including the first start. Sparse MDS currently requires one start.

`max.iter` Positive integer iteration limit per start. For SGD, the number of complete passes over full or retained sparse pairs. Defaults to 1000 for full MDS and 30 for sparse MDS. The sparse default is a provisional calibration choice, not a convergence criterion. With resume, this is the additional epoch budget (default 1000).

`eps` Positive SMACOF tolerance for the change in normalized stress. Do not supply this argument for SGD, which runs its prescribed schedule.

`seed` Integer random seed, or NULL to use the current RNG stream. With a non-NULL seed, random starts do not change the caller's RNG state.

`diagnostics` Attach the common GMDS diagnostic panel. With FALSE and raw graph inputs in full mode, prepare only the distance matrix, without path caches. Defaults to TRUE for full MDS and FALSE for sparse MDS. Sparse MDS rejects TRUE and explicitly supplied diagnostic controls (scale.mode, distance.floor, edge.length.epsilon, band.quantiles).

`scale.mode` Diagnostic scale policy: "profiled" fits a separate scalar for each diagnostic family, and "identity" uses scale one. To evaluate user-specified scales, call score.gmds() on the returned coordinates separately. This argument never changes the fitted coordinates.

`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.

`backend` "sgd" (default) or "smacof". No automatic fallback occurs.

`sgd.control` Named list used only with backend = "sgd": scheduler ("hybrid", "exponential" or "zheng"), learning.rate (0.5), final.rate (0.01), switch.ratio (0.4), schedule.epsilon (0.1), checkpoint.every (1), schedule.epochs (initial max.iter), keep.state (FALSE), stopping (NULL, disabled), trace (NULL, disabled), and max.workspace.bytes (256 MiB for full MDS, 512 MiB for sparse MDS). The schedule defaults to "zheng" for prepare.sparse.mds() inputs and "hybrid" otherwise. See Learning-rate schedules and evidence below. The hybrid/exponential tuning defaults are provisional calibration choices. Their rates must be positive with final.rate <= learning.rate; switch.ratio is between zero and one inclusive. The memory allowance covers native workspace, not the R input matrices, retained continuation state, serialization copies or total process memory. See Stopping, histories and continuation below.

`pair.weights` "uniform" (full default) or "inverse_squared" (sparse default and the only supported sparse choice). The latter applies the paper's inverse-squared shortest-path-distance weights in either backend. This is separate from the graph's edge.weights.

`approximation` "full" (default) or "sparse". This selects the distance model separately from the optimizer selected by backend. Sparse mode currently supports only SGD, in 2D or 3D. The default dim = 2 is shared by both modes. Mode-specific defaults apply only to omitted arguments; explicitly incompatible choices produce errors.

`sparse.control` Named list used only with approximation = "sparse": n.pivots (200, capped at n) and optional distinct vertex ids pivots in selection order. Explicit pivots override the default count; supplying an inconsistent count is an error. The default count is a provisional calibration choice; increasing it increases memory and work per epoch.

`constraints` Optional list for sparse SGD with explicit distance targets: pairs is a two-column matrix of distinct unordered vertex pairs; targets contains positive finite distances. Optional count_i and count_j contain nonnegative endpoint multiplicities (both default to one), divided by squared target distance during fitting. Each pair must have positive total multiplicity. Supply n; the constraint network must be connected. Cannot accompany graph inputs, prepared objects or pivot controls. Targets are used unchanged, with no shortest-path calculation. Symmetric multiplicities give ordinary sparse inverse-squared stress; asymmetric multiplicities use the existing sparse endpoint updates. Zero distances are unsupported: consolidate duplicate observations before constructing inverse-squared constraints. No triangle-inequality validation is performed.

`distance.matrix` Optional square numeric matrix of supplied distances for full MDS. Values are used directly, without graph shortest paths. Must be finite, symmetric and nonnegative with zero diagonal. Cannot accompany graph inputs, prepared objects or constraints. Graph diagnostics are not available: diagnostics defaults to FALSE and TRUE is rejected.

`resume` A prior SGD result created with keep.state = TRUE. When supplied, only resume and max.iter may be specified; targets, weights, preparation, initialization, seeds and controls are frozen in saved state. Requires n.init = 1 in the original fit. Historical fits without retained state cannot be resumed from their returned coordinates.

Details

With approximation = "full", the objective is

$$S(Z) = \sum_{i<j} w_{ij}(\|z_i - z_j\|_2 - \delta_{ij})^2.$$

By default, all pair weights are one. With pair.weights = "inverse_squared", $w_{ij} = 1/\delta_{ij}^2$, as in Zheng et al. (2018); stress is then the sum of squared relative distance errors. Edge weights define graph distances; pair.weights separately specifies their importance in the stress objective. Both backends, coordinate rescaling, checkpoint/start selection, and stress metadata use the selected weights. The common score.gmds() diagnostic panel retains its own definitions and does not become a weighted objective report. SMACOF normalizes targets internally. Returned coordinates are rescaled to minimize raw stress against the original input distances. scale.mode controls the optional diagnostic panel only, not the optimization objective.

Target-normalized raw stress and scale-profiled Stress-1 select the same shapes when global scale is free and pair weights agree. Their values at a fixed coordinate scale need not agree. Both the literal and target-profiled Stress-1 values are independently calculated in `metadata`; the backend's own reported stress is retained separately in the start summaries.

The first run uses `init`; subsequent runs use random configurations. The smallest achieved raw stress selects the result. A backend result that increases stress beyond numerical tolerance is rejected in favor of its scaled start and marked accordingly. Failed starts are recorded, and all failing starts cause an error. Iteration limits are not convergence or global optimality certificates. Collinear or planar starts may remain in their initial span; use multiple starts to investigate this sensitivity.

Newly prepared graph objects supply symmetric strict shortest distances, separately from retained-route lengths used in path diagnostics. Rebuild older cached preparations if their near-tie distance matrix is asymmetric.

Targets must be finite, symmetric, and nonnegative, with zero diagonal and at least one positive distance. Missing and infinite distances are rejected. Inverse-squared weighting requires strictly positive off-diagonal targets; zero distances are rejected, not floored. Uniform weighting still allows zero distances. Extreme ratios that overflow or underflow the normalized weights are rejected. Supplied starts may have coincident points but must not be wholly collapsed. Full MDS uses dense all-pairs matrices; use `approximation = "sparse"` when that preparation is too large. Sparse behavior is described below.

Value

A `"grip_gmds_layout"` object with method `"metric_mds"`. `metadata$engine` and `metadata$approximation` identify both choices. In full mode, `metadata` records the objective, backend/version, achieved raw stress, weighted target-normalized RMSE, both weighted Stress-1 conventions, selected start, coordinate scale multiplier, and per-start losses and stopping information. SGD adds per-start elapsed seconds (native fitting including scoring), native seeds, pair-update counts, best epochs, and checkpoint histories in `metadata$sgd`. Histories use RMS-normalized target-distance units and include both raw and scale-profiled stress. Internally, inverse-squared weights are `1 / (distance / input_rms_distance)^2`; the corresponding stress is already dimensionless. Uniform raw stress is converted to squared input-distance units for public metadata. Weighted target-normalized RMSE takes the square root of the ratio of stress to the sum of weighted squared target distances. SGD reports `iteration_limit` and `converged = FALSE` when its schedule finishes, including when an earlier checkpoint is returned.

Full SGD behavior

Every pass visits all unordered pairs in shuffled order. The pair step is clipped at `min(rate * normalized_pair_weight, 1)`. Rates use RMS-normalized distance units for both weighting choices; changing input distance units therefore preserves the optimization trajectory up to scale. The default schedule is shared across weighting choices, not a reproduction of the graph-dependent annealing schedule used in Zheng et al. (2018). Symmetric clipped updates follow Zheng et al. (2018); the hybrid exponential/harmonic schedule follows Hangan et al. (2026). The terminal epoch is always scored. The best independently scored checkpoint, including initialization, is retained using scale-profiled stress; checkpoint scoring does not rescale the ongoing search. The R wrapper independently rescales and recomputes its returned stress. The legacy user-controlled exponential schedule's final rate is its boundary value after the planned passes, not its last applied rate. The hybrid schedule switches after `floor(switch.ratio * schedule.epochs)` passes, at one

tenth the initial rate (or the initial rate if the switch is immediate). Its harmonic segment approaches `final.rate` at the boundary only when the switch rate exceeds `final.rate`; otherwise that segment stays constant at the switch rate. With `switch.ratio = 1`, no harmonic segment occurs and `final.rate` is unused. Coincident points with positive target distance are separated along a seeded direction; zero targets are not floored. Without such collisions, updates preserve a deficient initial affine span. Use full-dimensional random starts to examine this sensitivity. This backend retains quadratic pair storage and work per pass. It does not validate the supplied graph distances.

Sparse approximation

Sparse SGD implements the asymmetric updates of Zheng et al. (2018), Algorithm 2, based on the model of Ortmann et al. (2017). The graph must have positive finite edge lengths and be connected. Disconnected graphs are rejected without repair. Pivots are chosen with probability proportional to distance to the nearest selected pivot, starting from a uniform random vertex. Graph lengths are normalized by their maximum before shortest-path calculations. Exact distance ties in region assignment go to the earliest selected pivot. Near floating-point boundaries, general weighted graphs can still be sensitive to arithmetic. Each nonadjacent vertex receives an influence from a pivot weighted by the number of vertices in that pivot's region within half their separation, divided by squared separation. The reverse influence can be zero or different. Edges have inverse-squared weights at both endpoints.

Edge targets are the supplied edge lengths, as in the upstream sparse implementation. For a weighted edge with a shorter alternative route, that target differs from the shortest-path target of full MDS. Exact full-pivot update parity requires every edge to be a shortest path, as with unit lengths or Euclidean chord lengths, and matched starts, rates and order.

Targets are normalized by RMS over the retained unordered pairs; endpoint weights use the same units. Legacy graph calls default to grip's hybrid schedule; `prepare.sparse.mds()` inputs default to the published fixed-budget `"zheng"` schedule. Select the schedule explicitly when comparing methods. The terminal iterate is returned, centered and converted back to input units, with no fitted scale. A checkpoint proxy averages the two endpoint weights for each pair. Asymmetric updates are not claimed to follow the gradient of that proxy, or to decrease it monotonically. This diagnostic is not full stress or equation (19) of Zheng et al. Use independent distance evaluation to compare sparse and full layouts. No dense diagnostics or classical initialization run implicitly.

Sparse results share the `metric_mds` method name. `coords` is the terminal layout. `prepared` contains an edge-only graph, without a distance matrix. `metadata$sparse` records retained pairs, original-unit targets, endpoint multiplicities, selected pivots and region owners. Metadata also records seeds, controls, preparation/fitting seconds, native workspace estimates (separately checked for preparation and fitting), `sparse_proxy_stress`, its normalized root error, and stopping status. By default `termination = "iteration_limit"` and `converged = FALSE`. `trace` reports checkpoint proxy stress, epochs, rates and pair updates.

Recording individual SGD updates

Set `sgd.control = list(trace = list(every = 100L))` to record coordinates after every 100 examined pairs in each epoch, plus the initial state and each epoch endpoint. Set `every = 1L` to record every pair, including no-op pairs. Tracing requires `n.init = 1`. It is independent of `checkpoint.every`. The native solver dispatches once to a compile-time specialization: when tracing is disabled, no tracing checks, copies or buffers occur in the update loop. When enabled, recording adds time and memory but never changes solver arithmetic, the random stream, stopping or the selected result.

`fit$update_trace$coordinates` is a list of matrices in input-distance units; `fit$update_trace$steps` gives the epoch, examined pairs within that epoch, and cumulative `pair_updates`. These are raw iterates, without Procrustes alignment, recentering or the final best-checkpoint scale adjustment; the last frame can therefore differ from `fit$coords`. Stress is not recomputed at each frame. The existing `fit$trace` remains the epoch-level diagnostic history. A resumed fit records only its new call, starting at the saved terminal state, with absolute epoch/update counts. Keep earlier fits to retain their frames.

`trace$max.bytes` defaults to 64 MiB. Before fitting, the solver checks an upper-bound allocation for the requested epoch budget (even with early stopping), including coordinate payloads and a conservative per-frame allowance. This allocation also counts against `max.workspace.bytes`. R result conversion, older retained traces and serialization need additional memory. Increase `every` to reduce the number of frames. Completing the epoch budget is not a convergence certificate.

Stopping, histories and continuation

Default SGD calls still exhaust the prescribed epoch budget. This is reported as `metadata$stopping$reason = "epoch_budget"`, separately from whether the planned schedule has completed or a stopping rule was assessed. SGD always reports `converged = FALSE` and `stopping$certified = FALSE`: neither budget exhaustion nor practical stabilization certifies convergence or optimality. Full SGD returns its best scale-profiled checkpoint; sparse SGD returns its terminal iterate. Stopping uses current checkpoint diagnostics, not the running best. Metadata identifies the returned iterate and reports both best and terminal diagnostics in normalized target units.

Checkpoint history is preserved in both `trace` and `metadata$history`; full multi-start runs also retain every start's trace in `metadata$sgd`. History includes global epoch, diagnostic, learning rate, pair updates, checkpoint type, relative diagnostic change, centered motion, update residual and consecutive successful checks. Unassessed quantities are `NA`.

Opt in with `stopping = list()` inside `sgd.control`, or supply any of `relative.tol` (1e-6), `motion.tol` (1e-5), `residual.tol` (1e-5), `patience` (5), and `min.epochs` (10). At scheduled checkpoints after at least `min.epochs`, all three diagnostics must pass for `patience` consecutive checks. Relative change is the absolute change in current diagnostic divided by `max(abs(previous), 1e-12)`. Motion is RMS point displacement after removing translation, without rotation alignment. The residual is the largest norm of each vertex's endpoint-weighted mean distance-error vector. Motion and residual use RMS-normalized target units. The residual is independent of learning rate, preventing a tiny rate alone from passing the rule. Positive-target collisions fail the residual check. For asymmetric sparse updates it is an update-field diagnostic, not a gradient certificate; the sparse stress proxy need not decrease. Defaults are practical safeguards, not universally calibrated scientific thresholds.

With `keep.state = TRUE`, the result retains terminal coordinates, native RNG, normalized targets and endpoint weights, schedule, checkpoint/stopping state, best checkpoint and preparation provenance. `saveRDS()`/`readRDS()` preserve this state. Resume with `metric.mds(resume = fit, max.iter = extra)`. State version, platform and an integrity fingerprint are checked; the fingerprint detects accidental edits and is not a security mechanism. Reproducibility is supported on the same platform with a compatible state version. Already stabilized runs cannot resume under altered controls.

`schedule.epochs` fixes the initial schedule horizon, independently of call size. Pausing and resuming continues the same RNG and rate sequence without restarting or stretching its prefix. Beyond this horizon, rate at extra epoch `j` is `last.rate / (1 + j / schedule.epochs)`. This exten-

sion is a practical harmonic tail, not the published unlimited-iteration schedule or a convergence guarantee. A larger fresh horizon is a different experiment. Checkpoints occur at global multiples of `checkpoint.every` and at the schedule horizon. Other call boundaries are recorded with `scheduled = 0`; they do not affect stopping or checkpoint selection in a continued run. Full fits may return such a boundary when it improves the best diagnostic for that call, but continuation always uses terminal solver state. With the same horizon and final budget, split and uninterrupted runs match; split histories additionally preserve intermediate call boundaries.

Learning-rate schedules and evidence

The original full-SGD backend adapted exponential and hybrid schedules from the implementation accompanying Hangan et al. (2026). The graph sparse-SGD extension retained those controls. The hybrid default is preserved for existing callers; it is not a demonstrated improvement over Zheng et al.'s weight-dependent schedule. The original package design records its settings as provisional calibration choices.

`scheduler = "hybrid"` starts at `learning.rate` (0.5), uses exponential decay during the first `floor(switch.ratio * schedule.epochs)` epochs (default fraction 0.4), then a harmonic tail toward `final.rate` (0.01). The switch and rate endpoints are user controls, not derived from endpoint weight extrema. `scheduler = "exponential"` instead uses the controlled rates in a single exponential phase. For these legacy schedules, the final rate is a planned boundary value at epoch `schedule.epochs`, not necessarily the last applied rate. These schedules are distinct from `"zheng"`.

Hangan et al. motivate a fixed switch for the usual uniform-weight MDS setting, where the weight ratio does not distinguish pairs. That rationale does not establish better performance for inverse-squared, region-weighted sparse stress. Package backend comparisons and the earlier sparse-MDS comparison used hybrid controls; they did not isolate schedule effects. The October 2026 observation-method validation used `"zheng"` throughout. These experiments do not establish that either schedule outperforms the other. A schedule comparison must hold targets, pivots, endpoint weights, starting coordinates, update-order seeds and budgets fixed, and report independently evaluated error and runtime.

Set `sgd.control = list(scheduler = "zheng", schedule.epsilon = 0.1)` to use Zheng et al.'s Section II-A1 fixed-budget exponential schedule. Its rates run from `1 / min(positive endpoint weights)` to `schedule.epsilon / max(positive endpoint weights)`, including both endpoints; one epoch uses the first rate. Weights are computed after grip's target normalization. `schedule.epsilon` defaults to 0.1; `learning.rate`, `final.rate` and `switch.ratio` must be omitted with this schedule. This is the default for `prepare.sparse.mds()` inputs, including uniform, Euclidean regional and geodesic regional preparations. Other inputs retain the hybrid default, including raw graphs and explicit constraints. Fits from observation preparations retain constraint pairs and connectivity metadata but omit duplicate constraint adjacency lists. The local-neighbor graph remains available in the original preparation's `graph` field.

Choosing `"zheng"` reproduces that schedule's rule, not every published solver setting or trajectory: epoch budget, initialization and scale, random-number generator and update order can differ. It does not implement the paper's separate Section II-A2 unlimited-iteration schedule or certify convergence. Prefer the named published schedule when the purpose is sparse-stress analysis following Zheng et al.; label hybrid runs as an alternative configuration rather than assuming they are better.

Workflow guides

Start with `vignette("function-guide", package = "grip")` to choose a layout, diagnostic, or reference comparison. List installed guides with `vignette(package = "grip")`.

References

Ortmann, M., Klimenta, M. and Brandes, U. (2017). A Sparse Stress Model. doi:10.7155/jgaa.00440. Zheng, J. X., Pawar, S. and Goodman, D. F. (2018). Graph Drawing by Stochastic Gradient Descent. doi:10.1109/TVCG.2018.2859997. Hangan, D., Kobourov, S. and Miller, J. (2026). Bridging Graph Drawing and Dimensionality Reduction with Stochastic Stress Optimization. https://arxiv.org/abs/2605.00641.

See Also

`prepare.sparse.mds()`, `classical.mds()`, `edge.kk()`, `smacof::mds()`

Examples

```
prepared <- prepare.graph.geodesic.mds(edges.cycle(6), n = 6)
fit <- suppressWarnings(metric.mds(prepared = prepared, dim = 2, max.iter = 30))
fit$metadata$raw_stress
fit$metadata$termination # Schedule completion is not a convergence certificate.
# Schedule completion is reported explicitly, not as convergence.
fit <- suppressWarnings(metric.mds(edges = edges.cycle(8), n = 8,
  dim = 3, init = "random", backend = "sgd", max.iter = 30,
  diagnostics = FALSE))
fit$metadata$engine
fit$metadata$target_normalized_rmse
sparse <- metric.mds(edges = edges.path(30), dim = 3,
  approximation = "sparse", sparse.control = list(n.pivots = 5))
sparse$metadata$sparse_proxy_stress
# Pause after 5 epochs of a 20-epoch schedule, then continue the same run.
first <- suppressWarnings(metric.mds(edges = edges.cycle(8), n = 8,
  diagnostics = FALSE, max.iter = 5,
  sgd.control = list(keep.state = TRUE, schedule.epochs = 20)))
continued <- metric.mds(resume = first, max.iter = 15)
continued$metadata$stopping
tail(continued$metadata$history)
```

`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.

Examples

```
mesh_edges <- edges.mesh(4, 4)
fit <- misf.geodesic.kk(
  edges = mesh_edges, n = 16, num.init = 4, dim = 2,
  top.level.restarts = 2, top.level.max.iter = 3,
  insertion.mode = "geodesic", insertion.max.iter = 6,
  refinement.pair.mode = "auto", refinement.full.limit = 4,
  refinement.max.iter = 3, final.pair.mode = "landmark",
  final.full.limit = 4, final.max.iter = 3,
  return.trace = TRUE, seed = 1
)
```

---

occupied_mesh_surface_helpers

Occupied-mesh and perforated-grid helpers

Description

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

Usage

```
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

The coords_surface component is 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.

Examples

```
keep <- matrix(TRUE, nrow = 4, ncol = 5)
keep[2, 3] <- FALSE
occupied <- occupied.mesh.surface.graph(keep, surface = "paraboloid")
```

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 `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.

Examples

```
notches <- keep.asymmetric.notches(7, 8, notch.depth = 2, notch.width = 2)
holes <- keep.periodic.holes(7, 8, hole.period = 3)
slits <- keep.slit.channels(7, 8, slit.period = 3)
windows <- keep.staggered.windows(7, 8, row.period = 3, col.period = 4)
```

---

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.
...	For static plots, base graphics options override defaults, including asp, axis labels, limits, pch, cex, and vertex col/bg. In an orthographic plot, symbol, size, color, line width and type options are passed to points(); other options configure the plot region. Edges use edge.col. For interactive plots, options go to rgl::plot3d() and follow its conventions (for example size, not base-graphics cex).
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.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.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.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 graph bundle's coords_surface component is 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.

Examples

```
asymmetric <- cube.asymmetric.cavities.surface.graph(
  level = 1, side = 5, cavity.size = 1, pocket.size = 1
)
channels <- cube.channel.network.surface.graph(
  level = 1, side = 5, channel.width = 1, branch.offset = 2
)
tunnels <- cube.periodic.tunnels.surface.graph(
  level = 1, side = 5, tunnel.width = 1,
  tunnel.period = 2, tunnel.offset = 1
)
```

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.

Examples

```
cycle_edges <- edges.cycle(6)
prepared <- prepare.edge.kk(cycle_edges, n = 6)
```

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"`.

Examples

```
cycle_edges <- edges.cycle(6)
prepared <- prepare.geodesic.kk(cycle_edges, n = 6)
```

`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.

The all-pairs `distance.matrix` contains symmetric strict graph distances. Retained-route lengths in `pair_graph_distance` follow the deterministic near-tie path convention and can differ slightly from these targets. Route selection and path diagnostics are not changed by this distinction.

Value

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

Examples

```
cycle_edges <- edges.cycle(6)
prepared <- prepare.graph.geodesic.mds(cycle_edges, n = 6)
```

`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".

Examples

```
cycle_edges <- edges.cycle(6)
prepared <- prepare.landmark.geodesic.kk(
  cycle_edges, n = 6, local.nbrs = 2, landmark.count = 2
)
```

---

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
```

`prepare.sparse.mds` *Prepare sparse metric MDS directly from observations*

Description

Selects pivots and computes fitting weights internally, without constructing a full distance matrix. Pass the result to [metric.mds()] with ‘approximation = "sparse"‘. Preparation can be reused across dimensions and fit seeds. The default fit schedule for this preparation is ‘"zheng"‘.

Usage

```
prepare.sparse.mds(
  X,
  method = c("euclidean", "uniform", "geodesic", "linf"),
  n.pivots = 200L,
  pivot.selection = c("randomized", "farthest"),
  q = 10L,
  neighbor.variance.target = 0.9,
  seed = 1L,
  max.workspace.bytes = 512 * 1024^2
)
```

Arguments

`X`	Finite numeric observation matrix with distinct rows.
`method`	‘"euclidean"‘ uses regional Euclidean sparse stress; ‘"uniform"‘ uses uniformly selected landmarks without regional multipliers or local edges; ‘"geodesic"‘ uses graph shortest-path targets and regions; ‘"linf"‘ uses regional L-infinity (maximum absolute difference) targets.
`n.pivots`	Positive pivot count, capped at ‘nrow(X) - 1‘ to retain a strictly sparse preparation. All-pivot reduction is tested internally.
`pivot.selection`	Distance-proportional ‘"randomized"‘ selection or ‘"farthest"‘ (Euclidean and L-infinity regional methods only). Uniform mode selects uniformly without replacement regardless of the omitted default.
`q`	Number of non-self neighbors for the symmetric union graph.
`neighbor.variance.target`	Fraction of PCA variance retained for neighbor search, default 0.9; ‘NULL‘ searches original coordinates. Scores are not whitened. Only search changes: edge lengths and bridges use original ‘X‘. A target of one retains all numerically resolved components. For ‘"linf"‘, the omitted default becomes ‘NULL‘; an explicit non-NULL target is rejected because PCA rotation does not preserve L-infinity distances.

`seed` Nonnegative integer preparation seed. The caller's RNG is untouched.

`max.workspace.bytes`
Conservative preparation allocation allowance; this is not a total process memory cap.

Details

The geodesic method uses exact component-MST repair in 'dgraphs'. Its streaming implementation can evaluate quadratically many distances but never retains their full matrix. A recent local 'dgraphs' implementation supporting 'search.data' and streaming component repair is required. Uniform and Euclidean modes compute one Euclidean distance row per pivot. L-infinity mode performs exact neighbor search in the supplied coordinates, breaking equal-distance ties by row index, and computes L-infinity pivot distances. Search streams candidates, with quadratic worst-case work but no full distance matrix, and does not require 'dgraphs'. No normalization is applied internally; normalize observations before preparation if needed. All modes use inverse-squared target weights. Regional counts are asymmetric endpoint multipliers, not ordinary symmetric stress weights.

Value

A 'grip_sparse_mds_preparation' list containing constraints, pivot distance rows, pivots, regions, graph (when used), PCA information and metadata. For '"linf"', 'graph' is a list with an 'edges' data frame ('from', 'to', 'length') and search 'metadata'; Euclidean and geodesic methods return a 'dgraphs' graph. Input coordinates and settings are retained for cache validation.

See Also

[metric.mds()], [landmark.mds.constraints()]

Examples

```
X <- cbind(seq_len(12), sin(seq_len(12)))
prep <- prepare.sparse.mds(X, method = "uniform", n.pivots = 3)
fit <- metric.mds(prepared = prep, approximation = "sparse", max.iter = 5)
fit$metadata$data_preparation$method
```

---

print.grip_gmds_layout

Print a geodesic-MDS layout summary

Description

Reports the fitted objective, available stopping information, coordinate dimensions, and separately labeled graph diagnostics. Missing convergence metadata is reported as such; finite coordinates do not imply convergence. Printing never recomputes diagnostics. Raw metric-MDS stress has squared input-distance units. The displayed graph RMSE values are divided by the root mean squared target distance and are dimensionless. Extract `x$coords` to plot or score the coordinates.

Usage

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

Arguments

x	A grip_gmds_layout object.
...	Unused additional arguments.

Value

The input object, invisibly.

See Also

classical.mds, metric.mds

---

print.grip_misf *Print a multiscale hierarchy summary*

Description

Summarizes the vertices and levels of a hop-distance MISF hierarchy.

Usage

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

Arguments

x	A grip_misf object from build.misf.
...	Unused additional arguments.

Value

The input object, invisibly.

See Also

build.misf

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.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

The coords_surface component is 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.

Examples

```
mask <- mask.cube.channel.network(
  side = 5, channel.width = 1, branch.offset = 2
)
cube <- recursive.cube.mask.surface.graph(mask, level = 1)
```

---

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.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.

The coords.surface component contains the 3D coordinates of the occupied cells in the same vertex order as edges. 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

The coords_surface component is 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.

Examples

```
grid <- recursive.mask.grid.surface.graph(mask.cross(3), level = 1)
```

`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.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

The `coords_surface` component is 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.

Examples

```
tetrahedron <- recursive.tetrahedron.mask.surface.graph(level = 1)
```

---

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.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

The coords_surface component is 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.

Examples

```
triangle <- recursive.triangle.mask.surface.graph(level = 1)
```

`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.

Examples

```
theta <- seq(0, 2 * pi, length.out = 7)[-7]
initial <- cbind(cos(theta), sin(theta))
fit <- repulsive.stage(
  initial, lambda = 0.1, max.iter = 1, engine = "R"
)
```

---

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.

Examples

```
theta <- seq(0, 2 * pi, length.out = 7)[-7]
coords <- cbind(cos(theta), sin(theta))
state <- repulsive.state(coords, lambda = 0.1, engine = "R")
```

---

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

The coords.param component contains the sampled planar coordinates. The coords.surface component contains 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

The coords_param component is an n x 2 numeric matrix with columns u and v.

The coords_surface component is 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.

Examples

```
rectangle <- sampled.rectangle.surface.graph(n = 20, k = 4, seed = 1)
sequence <- sampled.rectangle.surface.graphs(
  n = 20, k = c(3, 4), seed = 1
)
```

---

score.coordinates *Score coordinates against corresponding reference observations*

Description

Measures recovery of known coordinates, not preservation of graph distances. Existing layout stability and Figure 7 diagnostics are not changed.

Usage

```
score.coordinates(
  coords,
  reference,
  alignment = c("rigid", "none", "similarity"),
  allow.reflection = TRUE
)
```

Arguments

coords, reference
: Numeric matrices of identical dimensions, with rows in the same observation order. Row names do not trigger reordering.

alignment
: One of "rigid" (translation and orthogonal transformation), "none", or "similarity" (also fit a nonnegative uniform scale).

allow.reflection
: Allow a reflection in the fitted orthogonal matrix.

Details

A rigid fit preserves all distances. A similarity fit removes uniform scale as well. No vertex correspondence is estimated. A collapsed source is allowed for rigid alignment but has no identifiable similarity scale and is rejected for similarity alignment. Nonunique rotations in rank-deficient configurations can have the same minimum error.

Value

A list with rmse, relative_rmse, coords (aligned coordinates), rotation, translation, scale, alignment, and reference_radius. RMSE is the square root of the mean squared Euclidean vertex displacement, not the mean over individual coordinate entries. Relative RMSE divides it by the reference RMS radius about its centroid; it is NA for zero radius. The transformation is scale * coords %*% rotation + translation.

Examples

```
x <- rbind(c(0, 0), c(1, 0), c(0, 1))
score.coordinates(x + 2, x)$rmse
score.coordinates(2 * x, x, alignment = "similarity")$scale
```

---

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.

Examples

```
cycle_edges <- edges.cycle(6)
theta <- seq(0, 2 * pi, length.out = 7)[-7]
coords <- cbind(cos(theta), sin(theta))
prepared <- prepare.geodesic.kk(cycle_edges, n = 6)
score <- score.geodesic.kk(coords, prepared = prepared)
```

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 integer edge matrix used when prepared is omitted. Supply either edges/edge.weights or adj.list/weight.list, not both. Raw graph inputs cannot be combined with a prepared object.
n	Finite positive integer vertex count. When supplied with a prepared object, it must match the stored graph size.
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. Lower residual errors indicate closer agreement with the stated targets; no single column certifies geometric recovery.

Diagnostic columns

A chord is the straight-line distance between two coordinates. A retained path adds embedded edge lengths along a route stored in the preparation. These can disagree on a folded graph. The diagnostic targets use the stored route convention and can differ slightly from strict shortest distances near ties. Compare candidates with the same graph, preparation, and scale policy.

- `n.vertices`, `n.edges`, `n.pairs`: graph size and numbers of stored edges and retained pairs (before exclusion of invalid or near-zero targets).
- `scale.mode`: the requested scale policy. `edge.scale`, `gmds.scale`, and `metric.chord.scale` multiply edge, path, and chord targets separately.
- `edge.rel.rmse`, `gmds.stress`, `metric.chord.stress`: root sum squared residual divided by root sum squared scaled target, a dimensionless error. This is not the mean of squared pairwise relative errors and is not the raw stress minimized by `metric.mds()`.
- `edge.rmse`, `gmds.rmse`, `metric.chord.rmse`: root mean squared residual in drawn-coordinate units against the scaled targets.
- `edge.mean.abs.error`, `gmds.mean.abs.error`: mean absolute residual in drawn-coordinate units. The corresponding `mean.rel.abs.error` columns (also available for `metric.chord`) average absolute residual/target ratios.
- `edge.signed.bias`, `gmds.signed.bias`: mean residual/target ratio; positive values indicate lengths greater than their scaled targets.
- `gmds.short.stress`, `gmds.mid.stress`, `gmds.long.stress` and their `signed.bias` counterparts: the path statistics restricted to bands defined by `band.quantiles`; ties at a boundary belong to the lower band.
- `shortcut.fraction`: fraction of evaluated retained paths more than 5% shorter than their scaled target, between zero and one.
- `spread.score`: RMS chord length divided by RMS unscaled graph target; it changes with drawing scale and is not a universal quality grade.

Residual summaries exclude nonfinite values, nonpositive weights, and targets no larger than `distance.floor`. Unavailable summaries are `NA`, including path/chord diagnostics for edge-only preparations. No reference alignment is performed. Use `score.coordinates()` or `score.surface()` for reference agreement, keeping their alignment and units explicit.

Workflow guides

Start with `vignette("function-guide", package = "grip")` to choose a layout, diagnostic, or reference comparison. List installed guides with `vignette(package = "grip")`.

Examples

```
cycle_edges <- edges.cycle(6)
theta <- seq(0, 2 * pi, length.out = 7)[-7]
coords <- cbind(cos(theta), sin(theta))
prepared <- prepare.graph.geodesic.mds(cycle_edges, n = 6)
score <- score.gmds(coords, prepared = prepared)
score[, c("scale.mode", "edge.rel.rmse", "gmds.stress", "metric.chord.stress")]
# These dimensionless errors assess edges, retained paths, and chords;
# they do not measure agreement with reference coordinates.
```

---

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.

Examples

```
cycle_edges <- edges.cycle(6)
theta <- seq(0, 2 * pi, length.out = 7)[-7]
coords <- cbind(cos(theta), sin(theta))
prepared <- prepare.landmark.geodesic.kk(
  cycle_edges, n = 6, local.nbrs = 2, landmark.count = 2
)
score <- score.landmark.geodesic.kk(coords, prepared = prepared)
```

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"`.

Details

The sampled stress is scale invariant. For sampled pairs p, let d_p be graph distance and e_p be Euclidean layout distance. The fitted scale is $a = \sum_p e_p d_p / \sum_p d_p^2$, and the reported value is $\sqrt{\sum_p (e_p - a d_p)^2 / \sum_p e_p^2}$. Edge-length CV is the standard deviation divided by the mean of embedded edge lengths. The sampled non-edge separation ratio is the minimum sampled non-edge distance divided by the median embedded edge length. Cluster separation is the mean pairwise distance between group centroids divided by the mean within-group radius.

Value

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

Workflow guides

Start with `vignette("function-guide", package = "grip")` to choose a layout, diagnostic, or reference comparison. List installed guides with `vignette(package = "grip")`.

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.

Examples

```
mesh_edges <- edges.mesh(4, 4)
prepared <- prepare.misf.geodesic.kk(
  mesh_edges, n = 16, num.init = 4,
  top.level.mode = "skip", seed = 1
```

```
)
coords <- matrix(seq_len(32), nrow = 16, ncol = 2)
score <- score.misf.geodesic.kk(coords = coords, prepared = prepared)
```

`score.surface` *Estimate symmetric area-weighted distance between triangular surfaces*

Description

Compares already aligned surfaces using closest points on triangles, not nearest vertices. No registration, rescaling, or triangulation is performed.

Usage

```
score.surface(
  coords,
  triangles,
  reference.coords,
  reference.triangles,
  sample.size = 5000L,
  seed = 1L
)
```

Arguments

`coords`, `reference.coords`
: Numeric three-column vertex matrices.

`triangles`, `reference.triangles`
: Three-column matrices of one-based triangle indices into the corresponding vertex matrix. Open surfaces and different triangulations are supported. Duplicate faces are rejected.

`sample.size`
: Number of independent area-uniform samples per surface, at least two. Increase this to assess Monte Carlo convergence.

`seed`
: Nonnegative integer seed, at most 2147483646. Separate fixed streams are used for the two directions; the caller's RNG state is restored.

Details

For surfaces A and B, squared symmetric RMS is one half of the sum of the area-normalized integrals of squared closest-point distance in the two directions. Each direction has equal weight regardless of total area. The returned score has coordinate-distance units. The Monte Carlo standard error uses a delta-method approximation; it does not quantify mesh discretization error, alignment uncertainty, or between-cloud variability. Zero observed error gives a standard error of zero, not a proof of identity. Samples are uniform over triangle area, so overlapping faces count with multiplicity. Self-intersections are not repaired or detected. Zero-area faces have no sampling mass but remain distance targets. A surface with no positive-area faces is rejected. No Hausdorff maximum is estimated.

Value

A list with rms, forward_rms, reverse_rms, forward_mean, reverse_mean, rms_mc_se, surface areas, zero-area face counts, sample_size, and seed. Forward means coords to reference.

Examples

```
x <- rbind(c(0, 0, 0), c(1, 0, 0), c(0, 1, 0))
f <- matrix(1:3, nrow = 1)
score.surface(x, f, x, f, sample.size = 100)$rms
```

---

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.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

The coords.surface component contains 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

The coords_surface component is 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.

Examples

```
carpet <- sierpinski.carpet.surface.graph(level = 1)
```

---

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.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

The coords_surface component is 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.

Examples

```
tetrahedron <- sierpinski.tetrahedron.surface.graph(level = 1)
```

---

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.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

The coords_surface component is 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.

Examples

```
triangle <- sierpinski.triangle.surface.graph(level = 1)
```

---

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.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

The coords.surface component contains the 3D coordinates of the sampled surface in the same vertex order as edges: 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

The coords_surface component is 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.

Examples

```
sphere <- sphere.surface.graph(4, 6, surface = "ellipsoid")
```

`suitesparse.graph` *Bundled SuiteSparse Graphs*

Description

Load a real graph offline, or browse the frozen package catalogue. No network, Python, or external viewer is needed. The catalogue includes the curated comparison gallery and all six examples used by Zheng, Pawar and Goodman. Gallery membership and original-six membership are separate columns.

Usage

```
suitesparse.graph(name)

suitesparse.graphs()
```

Arguments

`name`	Full graph ID (for example ‘"HB/494_bus"‘) or a unique short name (‘"494_bus"‘). Names are case-sensitive. Component-derived IDs include the ‘__LCC_‘ suffix and never resolve as the original full graph.

Details

Square matrices use the union of off-diagonal numerical support with its transpose, after summing duplicates and removing numerical zeros. Rectangular matrices use distinct row and column vertices. Matrix signs, magnitudes and physical units are not traversal lengths: every edge has length one. Some full graphs are disconnected. Largest-component graphs are explicitly identified and retain their original vertex mapping.

The installed selection is fixed; changing viewer favorites does not change package data. See ‘extdata/suitesparse/selection-30_sept_2026.json‘ and ‘extdata/suitesparse/PROVENANCE.html‘ in the installed package.

Value

'suitesparse.graphs()' returns a data frame with 'id', 'name', 'group', 'vertices', 'edges', 'conversion_type', 'components', 'original_vertices', 'retained_fraction', 'gallery', and 'original_six'.

'suitesparse.graph()' returns a list with 'n' (including isolated vertices), 'edges' (a two-column integer matrix of unique, lexicographically sorted, one-based undirected edges), 'edge_weights' (unit traversal lengths), 'vertex_data', and 'provenance'. Vertex data contain consecutive 'vertex', one-based 'source_index', 'label', 'source_partition' ('row' or 'column'), 'original_vertex' (one-based index before component extraction), and one-based 'component', 'vertex_id' (saved graph identity), and 'source_vertex_id' (identity before extraction). In bipartite graphs, source indices refer separately to matrix rows and columns. Les Miserables retains character names.

Provenance includes source and download URLs, source matrix headers with original credits, CC BY 4.0 licensing, SHA-256 checksums, conversion recipe, original graph identity, and component retention information. The original six also retain their earlier metadata under 'original_example'.

Source

SuiteSparse Matrix Collection, https://sparse.tamu.edu/.

References

Davis, T. A. and Hu, Y. (2011). The University of Florida Sparse Matrix Collection. doi:10.1145/2049662.2049663.

Zheng, J. X., Pawar, S. and Goodman, D. F. M. (2018). Graph Drawing by Stochastic Gradient Descent. doi:10.1109/TVCG.2018.2859997.

Examples

```
catalogue <- suitesparse.graphs()
head(catalogue)
g <- suitesparse.graph("494_bus")
identical(g, suitesparse.graph("HB/494_bus"))
c(vertices = g$n, edges = nrow(g$edges))
head(suitesparse.graph("lesmis")$vertex_data)
```

---

tetrahedron_mask_helpers

Tetrahedron mask helpers for recursive gasket families

Description

Convenience constructors for the four-corner tetrahedron masks used by 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.

Examples

```
classic <- mask.tetrahedron.classic()
missing_apex <- mask.tetrahedron.corner.missing("apex")
```

---

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.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

The coords.surface component contains 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

The coords_surface component is 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.

Examples

```
torus <- torus.surface.graph(4, 6, surface = "pinched")
```

`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). Supply either edges/edge.weights or adj.list/weight.list, not both. Fractional, nonfinite, and out-of-range vertex ids are rejected before integer conversion.
n	Number of vertices, a finite positive integer.
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.

Workflow guides

Start with `vignette("function-guide", package = "grip")` to choose a layout, diagnostic, or reference comparison. List installed guides with `vignette(package = "grip")`.

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). Supply either edges/edge.weights or adj.list/weight.list, not both. Fractional, nonfinite, and out-of-range vertex ids are rejected before integer conversion.
n	Number of vertices, a finite positive integer.
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.every`th 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 `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.

Examples

```
classic <- mask.triangle.classic()
bridge <- mask.triangle.bridge("top")
```

`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.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

The `coords.surface` component is the 3D coordinates of the clipped lattice vertices in the same vertex order as `edges`. `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

The coords_surface component is 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.

Examples

```
annulus <- triangulated.annulus.surface.graph(resolution = 6)
```

---

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.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

The coords.surface component is the 3D coordinates of the clipped lattice vertices in the same vertex order as edges. triangulated.pair.of.pants.surface.graph() 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

The coords_surface component is 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.

Examples

```
pair_of_pants <- triangulated.pair.of.pants.surface.graph(resolution = 8)
```

`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.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

The `coords_surface` component is 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.

Examples

```
tetrahedron <- triangulated.polyhedron.surface.graph(
  base = "tetrahedron", level = 1
)
```

---

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.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

The coords.surface component contains the 3D coordinates of the occupied cells in the same vertex order as edges. vicsek.surface.graph() returns a reusable weighted-graph bundle for the named Vicsek family.

Value

The coords_surface component is 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.

Examples

```
vicsek <- vicsek.surface.graph(level = 1)
```

---

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.

Examples

```
cycle_edges <- edges.cycle(6)
cycle_lengths <- seq(0.8, 1.3, length.out = nrow(cycle_edges))
layout <- weighted.grip.nd(
  edges = cycle_edges, n = 6, edge.weights = cycle_lengths,
  dim = 4, rounds = 1, final.rounds = 1,
  num.init = 5, num.nbrs = 2, seed = 1
)
```

Index

∗ **datasets**
    hmp.gc, 53
    hmp.u01.gc.coarse, 54

build.misf, 6, *110*
build.misf(), *45*
build.weighted.misf, 8
build.weighted.misf(), *45*

classical.mds, 9, *110*
classical.mds(), *17*, *45*, *46*, *76*, *82*, *90*
compare.layouts, 10, *96*, *132*
compare.layouts(), *45*
cube.asymmetric.cavities.surface.graph *(*porous_cube_surface_helpers*)*, 99
cube.channel.network.surface.graph *(*porous_cube_surface_helpers*)*, 99
cube.periodic.tunnels.surface.graph *(*porous_cube_surface_helpers*)*, 99
cube_mask_pattern_helpers, 13
cylinder.surface.graph *(*cylinder_surface_helpers*)*, 14
cylinder_surface_helpers, 14

edge.kk, 15, *102*
edge.kk(), *45*, *46*, *90*
edge.length.density.stiffness, 18
edge.length.density.stiffness(), *17*, *45*, *71*
edge.repulsive.stage, 19
edge.repulsive.stage(), *45*
edge.repulsive.state, 21
edge.repulsive.state(), *45*, *119*
edges.cube *(*graph_generators*)*, 37
edges.cube(), *46*
edges.cycle *(*graph_generators*)*, 37
edges.cycle(), *46*
edges.cylinder *(*graph_generators*)*, 37
edges.kary.tree, *68*
edges.kary.tree *(*graph_generators*)*, 37
edges.mesh *(*graph_generators*)*, 37
edges.mesh(), *46*
edges.path *(*graph_generators*)*, 37
edges.path(), *46*
edges.sierpinski.carpet, *38*
edges.sierpinski.carpet *(*graph_generators*)*, 37
edges.sierpinski.tetrahedron, *38*
edges.sierpinski.tetrahedron *(*graph_generators*)*, 37
edges.sierpinski.triangle, *38*
edges.sierpinski.triangle *(*graph_generators*)*, 37
edges.torus *(*graph_generators*)*, 37
estimate.preparation, 23

geodesic.kk, 24
geodesic.kk(), *45*, *76*
geometry.diagnostics, 26
geometry.diagnostics(), *45*
globalrep.grip, 27
globalrep.grip(), *44*
globalrep.weighted.grip, 31
globalrep.weighted.grip(), *44*
graph.riemannian.star.structure, 35
graph.riemannian.star.structure(), *70*, *71*
graph_generators, 37
grip, *4*, *11*, *12*, *27*, *31*, *37*, 39, *77*, *96*, *102*, *145*, *149*
grip(), *17*, *23*, *44*, *76*
grip-0.2-migration, 44
grip-mds-migration, 46
grip-package, 4
grip.build.misf *(*grip-0.2-migration*)*, 44
grip.compare.layouts *(*grip-0.2-migration*)*, 44

grip.edge.length.density.stiffness
(grip-0.2-migration), 44
grip.edge.repulsive.state
(grip-0.2-migration), 44
grip.geometry.diagnostics
(grip-0.2-migration), 44
grip.gmds.layout.result
(grip-0.2-migration), 44
grip.layout (grip-0.2-migration), 44
grip.metric.mds.layout
(grip-0.2-migration), 44
grip.optimize.edge.gkk.layout
(grip-0.2-migration), 44
grip.optimize.edge.isometric.layout
(grip-0.2-migration), 44
grip.optimize.edge.kk.layout
(grip-0.2-migration), 44
grip.optimize.edge.repulsive.stage
(grip-0.2-migration), 44
grip.optimize.geodesic.kk
(grip-0.2-migration), 44
grip.optimize.kernel.gram.gkk.layout
(grip-0.2-migration), 44
grip.optimize.landmark.geodesic.kk
(grip-0.2-migration), 44
grip.optimize.misf.geodesic.kk
(grip-0.2-migration), 44
grip.optimize.repulsive.stage
(grip-0.2-migration), 44
grip.params.from.summary
(grip-0.2-migration), 44
grip.plot (grip-0.2-migration), 44
grip.prepare.edge.kk
(grip-0.2-migration), 44
grip.prepare.geodesic.kk
(grip-0.2-migration), 44
grip.prepare.graph.geodesic.mds
(grip-0.2-migration), 44
grip.prepare.landmark.geodesic.kk
(grip-0.2-migration), 44
grip.prepare.misf.geodesic.kk
(grip-0.2-migration), 44
grip.project.3d (grip-0.2-migration), 44
grip.repulsive.state
(grip-0.2-migration), 44
grip.score.geodesic.kk
(grip-0.2-migration), 44
grip.score.gmds.layout
(grip-0.2-migration), 44
grip.score.landmark.geodesic.kk
(grip-0.2-migration), 44
grip.score.layout (grip-0.2-migration),
44
grip.score.misf.geodesic.kk
(grip-0.2-migration), 44
gripui.app, 47
gripui.family.app, 48
gripui.graph.family.catalog, 48
gripui.project, 49
gripui.project.from.compare, 50
gripui.project.from.dir, 51
gripui.validate.project, 52

hmp.gc, 53
hmp.u01.gc.coarse, 54

irregular.annulus.surface.graph
(irregular_annulus_surface_helpers),
55
irregular.ball.solid.graph
(irregular_ball_solid_helpers),
57
irregular.double.torus.surface.graph
(irregular_double_torus_surface_helpers),
58
irregular.pair.of.pants.surface.graph
(irregular_pair_of_pants_surface_helpers),
60
irregular.rectangle.surface.graph
(irregular_rectangle_surface_helpers),
62
irregular.shell.solid.graph
(irregular_shell_solid_helpers),
64
irregular.sphere.surface.graph
(irregular_sphere_surface_helpers),
65
irregular.torus.surface.graph
(irregular_torus_surface_helpers),
66
irregular_annulus_surface_helpers, 55
irregular_ball_solid_helpers, 57
irregular_double_torus_surface_helpers,
58
irregular_pair_of_pants_surface_helpers,
60

irregular_rectangle_surface_helpers, 62
irregular_shell_solid_helpers, 64
irregular_sphere_surface_helpers, 65
irregular_torus_surface_helpers, 66

kary.tree.weighted.graph (kary_tree_weighted_graph_helpers), 68
kary_tree_weighted_graph_helpers, 68
keep.asymmetric.notches (perforated_grid_helpers), 96
keep.periodic.holes (perforated_grid_helpers), 96
keep.slit.channels (perforated_grid_helpers), 96
keep.staggered.windows (perforated_grid_helpers), 96
kernel.gram.gkk, 70
kernel.gram.gkk(), *45, 46*

landmark.geodesic.kk, 72
landmark.geodesic.kk(), *45*
landmark.mds.constraints, 74
layout.coords, 75
legacy.grip, *29*, *33*, 77
legacy.grip(), *45*

mask.asymmetric.holes (mask_pattern_helpers), 79
mask.border (mask_pattern_helpers), 79
mask.corner (mask_pattern_helpers), 79
mask.cross (mask_pattern_helpers), 79
mask.cube.asymmetric.cavities (cube_mask_pattern_helpers), 13
mask.cube.channel.network (cube_mask_pattern_helpers), 13
mask.cube.periodic.tunnels (cube_mask_pattern_helpers), 13
mask.tetrahedron.classic (tetrahedron_mask_helpers), 142
mask.tetrahedron.corner.missing (tetrahedron_mask_helpers), 142
mask.triangle.bridge (triangle_mask_helpers), 152
mask.triangle.classic (triangle_mask_helpers), 152
mask_pattern_helpers, 79

menger.sponge.surface.graph (menger_sponge_surface_helpers), 80
menger_sponge_surface_helpers, 80
mesh.surface.graph (mesh_surface_helpers), 81
mesh_surface_helpers, 81
metric.mds, 82, *110*
metric.mds(), *17*, *46*, *129*
misf.geodesic.kk, 90
misf.geodesic.kk(), *45*, *134*

occupied.mesh.surface.graph, *96*
occupied.mesh.surface.graph (occupied_mesh_surface_helpers), 94
occupied_mesh_surface_helpers, 94

params.from.summary, 96
params.from.summary(), *45*
perforated_grid_helpers, 96
plot.layout, 98, *111*
plot.layout(), *45*
porous_cube_surface_helpers, 99
prepare.edge.kk, 102
prepare.edge.kk(), *9*, *16*, *18*, *23*, *45*, *71*, *128*
prepare.geodesic.kk, *24*, 103, *127*
prepare.geodesic.kk(), *9*, *16*, *18*, *45*, *71*, *92*, *107*, *128*
prepare.graph.geodesic.mds, *102*, 104
prepare.graph.geodesic.mds(), *9*, *16*, *18*, *23*, *45*, *71*, *128*
prepare.landmark.geodesic.kk, *73*, *103*, 105, *131*
prepare.landmark.geodesic.kk(), *23*, *45*
prepare.misf.geodesic.kk, 106
prepare.misf.geodesic.kk(), *45*
prepare.sparse.mds, 108
prepare.sparse.mds(), *83, 84*, *87*, *89, 90*
print.grip_gmds_layout, 109
print.grip_misf, 110
project.3d, *98*, 111
project.3d(), *45*

recursive.cube.mask.surface.graph, *13*
recursive.cube.mask.surface.graph (recursive_cube_mask_surface_helpers), 111
recursive.mask.grid.surface.graph, *79*

recursive.mask.grid.surface.graph
(recursive_mask_grid_surface_helpers), 113
recursive.tetrahedron.mask.surface.graph, *142*
recursive.tetrahedron.mask.surface.graph
(recursive_tetrahedron_mask_surface_helpers), 115
recursive.triangle.mask.surface.graph, *152*
recursive.triangle.mask.surface.graph
(recursive_triangle_mask_surface_helpers), 116
recursive_cube_mask_surface_helpers, 111
recursive_mask_grid_surface_helpers, 113
recursive_tetrahedron_mask_surface_helpers, 115
recursive_triangle_mask_surface_helpers, 116
repulsive.stage, 118
repulsive.stage(), *45*
repulsive.state, 119
repulsive.state(), *45*, *118*
run.gripui, 120
run.gripui.family, 121

sampled.rectangle.surface.graph
(sampled_rectangle_surface_helpers), 122
sampled.rectangle.surface.graphs
(sampled_rectangle_surface_helpers), 122
sampled_rectangle_surface_helpers, 122
score.coordinates, 125
score.coordinates(), *129*
score.geodesic.kk, 126
score.geodesic.kk(), *45*
score.gmds, *102*, 128
score.gmds(), *45*, *84*
score.landmark.geodesic.kk, 130
score.landmark.geodesic.kk(), *45*
score.layout, *4*, *10*, *26*, 132
score.layout(), *45*
score.misf.geodesic.kk, 133
score.misf.geodesic.kk(), *45*
score.surface, 135
score.surface(), *129*
sierpinski.carpet.surface.graph
(sierpinski_carpet_surface_helpers), 136
sierpinski.tetrahedron.surface.graph
(sierpinski_tetrahedron_surface_helpers), 137
sierpinski.triangle.surface.graph
(sierpinski_triangle_surface_helpers), 138
sierpinski_carpet_surface_helpers, 136
sierpinski_tetrahedron_surface_helpers, 137
sierpinski_triangle_surface_helpers, 138
smacof::mds(), *90*
sphere.surface.graph
(sphere_surface_helpers), 139
sphere_surface_helpers, 139
suitesparse.graph, 141
suitesparse.graphs (suitesparse.graph), 141

tetrahedron_mask_helpers, 142
torus.surface.graph
(torus_surface_helpers), 143
torus_surface_helpers, 143
trace.grip, 145
trace.grip(), *45*, *76*
trace.legacy.grip, 150
trace.legacy.grip(), *45*
triangle_mask_helpers, 152
triangulated.annulus.surface.graph
(triangulated_annulus_surface_helpers), 153
triangulated.pair.of.pants.surface.graph
(triangulated_pair_of_pants_surface_helpers), 154
triangulated.polyhedron.surface.graph
(triangulated_polyhedron_surface_helpers), 156
triangulated_annulus_surface_helpers, 153
triangulated_pair_of_pants_surface_helpers, 154
triangulated_polyhedron_surface_helpers, 156

vicsek.surface.graph
(vicsek_surface_helpers), 157

vicsek_surface_helpers, 157

weighted.grip.nd, 158
weighted.grip.nd(), *45*