

Package: nat.fastcore (via r-universe)

October 8, 2026

Title Re-implementation of nat core functions in Rust

Version 0.13.0

Description A re-implementation of core 'nat' (Neuroanatomy Toolbox) functions in Rust for speed. Provides fast skeleton/tree (DAG) operations (geodesic distances, Strahler order, segment generation, twig pruning, synapse flow centrality, node classification, skeleton healing), mesh connected components, NBLAST/synNBLAST neuron similarity scoring, CMTK and Elastix spatial transforms (applying a registration to points without needing CMTK or Elastix installed), and landmark-based warps (thin-plate spline, moving least squares), exposed to R via extendr.

License GPL-3

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Config/rextendr/version 0.4.2

SystemRequirements Cargo (Rust's package manager), rustc >= 1.65.0, xz

Suggests testthat (>= 3.0.0)

Config/testthat/edition 3

Config/pak/sysreqs xz-utils libclang-dev

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

Date/Publication 2026-08-13 06:10:44 UTC

RemoteUrl <https://github.com/schlegelp/fastcore-rs>

RemoteRef HEAD

RemoteSha 5ada9d8347e1f72f0e415fef37ec914350a2f086

RemoteSubdir R/nat.fastcore

Contents

adjacency	4
all_dists_to_root	4
betweenness	5
boundary_halfedges	5
break_segments	6
bridges	6
child_to_parent_dists	7
classify_nodes	8
cmtk_properties	8
cmtk_read	9
cmtk_xform	10
cmtk_xform_inv	12
connected_components	14
connected_components_graph	14
contract_nodes	15
contract_vertices	15
descendant_counts	16
descendants	16
dist_to_root	17
downsample_skeleton	17
elastix_probe_invertible	18
elastix_properties	19
elastix_read	20
elastix_xform	21
elastix_xform_inv	22
exposed_halfedges	24
fast_hclust	24
generate_segments	25
geodesic_clusters	26
geodesic_distances	27
geodesic_farthest	28
geodesic_farthest_mesh	28
geodesic_matrix_graph	29
geodesic_matrix_mesh	30
geodesic_mst_graph	32
geodesic_mst_mesh	33
geodesic_nearest	34
geodesic_nearest_mesh	34
geodesic_pairs	35
geodesic_path	36
geodesic_predecessors	37
get_num_threads	38
has_cycles	38
heal_skeleton	39
leaf_order	40
level_set_components	41

longest_path	41
longest_paths	42
mesh_connected_components	43
minimum_spanning_tree	44
mls_affine	45
mls_transform	46
mls_xform	47
nblast	48
nblast_allbyall	49
nblast_dist	50
nblast_hclust	51
nblast_knn	53
nblast_pairs	55
node_indices	56
parents_from_edges	57
paths_to_root	58
prune_twigs	58
reroot	59
reroot_rewire	59
resample_skeleton	60
set_num_threads	61
simplify_mesh	61
simplify_mesh_lossless	63
simplify_rdp	64
simplify_skeleton	65
simplify_vw	65
smat_auto_limit	66
smooth_mesh	67
smooth_skeleton	69
smooth_skeleton_gaussian	70
stitch_fragments	71
strahler_index	72
subtree_height	73
symmetrize	73
synapse_flow_centrality	74
synblast	75
synblast_allbyall	76
tps_affine	77
tps_coefs	77
tps_transform	78
tps_xform	79
trace_loops	80
triangulate_rings	81
unique_edges	82

adjacency	<i>The skeleton's adjacency matrix, as the three arrays of a CSR matrix.</i>
-----------	--

Description

Handing back the raw arrays rather than a matrix object keeps this package free of a sparse-matrix dependency; feed them to `Matrix::sparseMatrix(p = indptr, i = indices, x = data, ...)` if you want one.

Usage

```
adjacency(parents, weights = NULL, directed = TRUE, transpose = FALSE)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
weights	Optional numeric vector of child-to-parent edge weights; NULL gives every edge weight 1.
directed	Logical; if TRUE (default) only child-to-parent edges are stored, if FALSE both directions are.
transpose	Logical; if TRUE store parent-to-child instead. Ignored when directed is FALSE, where the matrix is symmetric anyway.

Value

List with `indptr` (integer, $N + 1$ entries), `indices` (integer column index per non-zero) and `data` (numeric weight per non-zero). Rows and columns are in node-index order and column indices ascend within each row.

all_dists_to_root	<i>Compute all distances to root.</i>
-------------------	---------------------------------------

Description

Compute all distances to root.

Usage

```
all_dists_to_root(parents, sources, weights)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
sources	Optional integer vector of node indices to measure from; NULL uses every node.
weights	Optional numeric vector of child-to-parent edge weights; NULL counts edges (hop distance).

Value

Numeric vector of distances to the root for each requested node.

betweenness	<i>Betweenness centrality, in $O(N)$ rather than Brandes' $O(V \cdot E)$.</i>
-------------	---

Description

Shortest paths in a tree are unique, so the count through a node is a closed form: descendants times ancestors when directed, and a sum of products over the parts it separates when not. Pairs are only counted within a connected component.

Usage

```
betweenness(parents, directed = TRUE)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
directed	Logical; if TRUE only count paths running child-to-parent.

Details

Note this is *not* `navis::betweenness centrality(from_ = ...)`, which despite the name computes a descendant count — see `descendant_counts`.

Value

Numeric vector of path counts, aligned with parents. Returned as double rather than integer because an undirected 100k-node skeleton reaches $\sim 5e9$, well past R's 32-bit integer.

boundary_halfedges	<i>Every edge of a mesh that has only one face on it.</i>
--------------------	---

Description

An interior edge has two faces on it and a boundary edge has one, so this is a grouping of the 3F edges the faces name — and that grouping is the whole cost. Use `exposed_halfedges` instead where you already know which vertices are going away: that one never looks at the mesh as a whole.

Usage

```
boundary_halfedges(faces, threads = NULL)
```

Arguments

faces	Integer or numeric ($F, 3$) matrix of triangle vertex indices (0-based).
threads	Optional integer; number of threads. NULL uses all cores.

Details

The half-edges come back **directed**, wound the way the one face they have left winds them. That is what makes `trace_loops` a walk rather than a search, and what tells `triangulate_rings` which way round to wind the cap.

Value

Integer ($K, 2$) matrix of directed half-edges, in the order they appear in the 3F edge list.

break_segments	<i>Break the tree into its linear segments (one integer vector per segment).</i>
----------------	--

Description

Break the tree into its linear segments (one integer vector per segment).

Usage

```
break_segments(parents)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
---------	--

Value

List of integer vectors, one per linear segment.

bridges	<i>Which edges are bridges — the ones whose removal would disconnect their component.</i>
---------	---

Description

Tarjan's algorithm, one depth-first sweep. The counterpart to `minimum_spanning_tree` rather than a variant of it: the MST asks which edges to *keep* to stay connected, this asks which ones may not be *dropped*. That is the question behind "prune this graph but do not shatter it".

Usage

```
bridges(edges, n_nodes)
```

Arguments

edges	Integer or numeric (E, 2) matrix of edges (0-based node indices). Direction is ignored.
n_nodes	Integer; total number of nodes.

Details

Parallel edges are honoured — two nodes joined twice are joined by a cycle, so neither of those edges is a bridge. Self-loops are never bridges.

Value

Logical vector with one flag per input edge, TRUE for a bridge.

child_to_parent_dists	<i>Calculate child -> parent distances.</i>
-----------------------	--

Description

Calculate child -> parent distances.

Usage

```
child_to_parent_dists(parents, x, y, z)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
x, y, z	Numeric vectors of node coordinates, one entry per node.

Value

Numeric vector of Euclidean child-to-parent distances (0 for roots).

classify_nodes	<i>Classify nodes into roots (0), leaves (1), branch points (2) and slabs (3).</i>
----------------	--

Description

Classify nodes into roots (0), leaves (1), branch points (2) and slabs (3).

Usage

```
classify_nodes(parents)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
---------	---

Value

Integer vector: 0 root, 1 leaf, 2 branch point, 3 slab.

cmtk_properties	<i>Properties of a CMTK registration</i>
-----------------	--

Description

Properties of a CMTK registration

Usage

```
cmtk_affine(reg)
```

```
cmtk_domain(reg)
```

```
cmtk_dims(reg)
```

```
cmtk_spacing(reg)
```

```
cmtk_versions(reg)
```

Arguments

reg	A cmtk_registration, from cmtk_read() .
-----	---

Value

- `cmtk_affine()`: the 4x4 affine matrix of the first registration, or `NULL`.
- `cmtk_domain()`: a (k, 3) matrix, one row per spline warp. Points outside `[0, domain]` cannot be transformed – CMTK reports them as `FAILED`.
- `cmtk_dims() / cmtk_spacing()`: (k, 3) matrices of the control-point lattice dimensions and spacing.
- `cmtk_versions()`: the CMTK TypedStream version of each registration.

Examples

```
## Not run:
reg <- cmtk_read("JFRC2_FCWB.list")
cmtk_affine(reg)
cmtk_domain(reg)

## End(Not run)
```

cmtk_read

*Read a CMTK registration***Description**

Reads a CMTK registration (a 12-DOF affine plus, usually, a cubic B-spline warp) ready to be applied to points with `cmtk_xform()` and `cmtk_xform_inv()`.

Usage

```
cmtk_read(path)
```

Arguments

path Path to a CMTK .list registration directory, or to a registration file itself (plain or gzipped). Pass a character vector of several to build a chain, applied in order: `points -> path[1] -> path[2] -> ... -> output`.
Direction is **not** fixed here. It is chosen per call – see the `invert` argument of `cmtk_xform()`, and `cmtk_xform_inv()` – so one object serves every direction and the registration is parsed only once.

Details

Unlike `nat`'s CMTK support, this does **not** shell out to CMTK's `streamxform` binary: CMTK does not need to be installed. Results match `streamxform` to $\sim 5e-7$ – its own printing precision – including which points it declines to transform.

The registration is parsed once, here, and can then be applied as often as you like.

Value

An object of class `cmtk_registration`.

See Also

`cmtk_xform()`, `cmtk_xform_inv()`

Examples

```
## Not run:
reg <- cmtk_read("JFRC2_FCWB.list")
reg

# a chain: A -> B -> C, where the second registration is stored as C->B
chain <- cmtk_read(c("A_B.list", "C_B.list"))
cmtk_xform(chain, pts, invert = c(FALSE, TRUE))

## End(Not run)
```

cmtk_xform

Apply a CMTK registration to points

Description

Apply a CMTK registration to points

Usage

```
cmtk_xform(
  reg,
  xyz,
  transform = c("warp", "affine"),
  allow_extrapolation = FALSE,
  fallback_to_affine = FALSE,
  invert = FALSE,
  n_cores = NULL,
  progress = FALSE
)
```

Arguments

<code>reg</code>	A <code>cmtk_registration</code> , from <code>cmtk_read()</code> .
<code>xyz</code>	An (N, 3) matrix of coordinates. A data frame is also accepted – if it has x/y/z columns (in any case) those are used, whatever else it carries, otherwise the first three columns – as is a bare length-3 vector. <code>nat::xyzmatrix()</code> gives you this for a neuron.

transform	"warp" (default) applies the full transformation; "affine" applies only the affine component. A registration with no spline warp uses its affine either way.
allow_extrapolation	Evaluate points outside the registration's domain by clamping to the outermost control points, instead of failing them. Defaults to FALSE, which is what CMTK does: streamxform reports a point outside the domain as FAILED, and we return NaN. Setting this to TRUE gives every point <i>an</i> answer, but that answer extrapolates a warp which was never fitted there, and it will silently disagree with every other CMTK-based tool.
fallback_to_affine	Replace failed rows with the affine result rather than NaN. One of FALSE (the default), TRUE/"chain", or "hop". TRUE (or "chain") re-runs the whole chain affine-only, starting again from the <i>original</i> point. This is what nat/navis do – they hand the failed rows back to streamxform --affine-only over the same registration list – so it is the default whenever a fallback is asked for. "hop" instead swaps the affine in for only the hop that failed, keeping the warps of the hops that succeeded. Arguably the better answer, but a silent departure from every other CMTK-based tool, so you have to ask for it. On a single registration the two are identical; on a chain they differ by a median of ~6 world units. Works in both directions: a hop travelled backwards falls back to the <i>inverse</i> affine, so the rescued point still lands in the space you asked for.
invert	Logical, length 1 or one flag per registration. TRUE traverses that registration backwards – what you need when routing through a bridging graph, where an edge may be walked in either direction, and a chain may need some hops forwards and others backwards. This is not the same as <code>cmtk_xform_inv()</code>, which inverts the whole composition (reversing the order <i>and</i> flipping every hop). For a single registration the two agree; for a chain they do not, and only invert can express a mixed-direction traversal.
n_cores	Number of threads. NULL (default) uses all cores.
progress	Show a progress bar.

Value

An (N, 3) matrix. Rows that could not be transformed are NaN.

See Also

`cmtk_read()`, `cmtk_xform_inv()`

Examples

```
## Not run:
reg <- cmtk_read("JFRC2_FCWB.list")
pts <- rbind(c(50, 50, 50), c(100, 100, 20))
```

```

cmtk_xform(reg, pts)

# transform a neuron
n <- nat::Cell07PNs[[1]]
nat::xyzmatrix(n) <- cmtk_xform(reg, nat::xyzmatrix(n))

## End(Not run)

```

cmtk_xform_inv

Apply a CMTK registration to points, backwards

Description

The affine part is inverted exactly. The spline warp has no closed-form inverse and is solved per point by damped Gauss-Newton against the analytic Jacobian.

Usage

```

cmtk_xform_inv(
  reg,
  xyz,
  transform = c("warp", "affine"),
  initial_guess = NULL,
  max_iter = 50L,
  tolerance = 1e-09,
  accuracy = 0.001,
  clamp_to_domain = TRUE,
  fallback_to_affine = FALSE,
  invert = FALSE,
  n_cores = NULL,
  progress = FALSE
)

```

Arguments

reg	A cmtk_registration, from <code>cmtk_read()</code> .
xyz	An (N, 3) matrix of coordinates. A data frame is also accepted – if it has x/y/z columns (in any case) those are used, whatever else it carries, otherwise the first three columns – as is a bare length-3 vector. <code>nat::xyzmatrix()</code> gives you this for a neuron.
transform	"warp" (default) applies the full transformation; "affine" applies only the affine component. A registration with no spline warp uses its affine either way.
initial_guess	An (N, 3) matrix of starting points for the solver. Defaults to xyz itself, which is a good guess for any well-behaved registration. In a chain, this seeds only the first solve.
max_iter	Maximum Gauss-Newton iterations per point.

tolerance	Stop once the step falls below this.
accuracy	Accept a solution only if its residual is within this of the target; otherwise the row is NaN.
clamp_to_domain	Confine the iterate to the registration's domain box. This is what makes the result agree with CMTK. Turning it off finds preimages that lie outside the image domain, where <code>streamxform</code> reports failure – so you will get finite numbers where CMTK gives you none.
fallback_to_affine	Replace rows the solver could not land with the <i>inverse</i> affine result rather than NaN – the mirror of the same argument on <code>cmtk_xform()</code> , with the same "chain" (default) and "hop" semantics.
invert	The same per-hop flags as on <code>cmtk_xform()</code> , composed with this whole-chain inversion: hop <i>i</i> runs <i>forwards</i> here exactly when <code>invert[i]</code> is TRUE.
n_cores	Number of threads. NULL (default) uses all cores.
progress	Show a progress bar.

Details

Points whose inverse does not converge come back as NaN. This is deliberate and matches CMTK, which reports such points as FAILED – a registration is only defined over a finite domain, and some points simply have no preimage inside it.

Value

An (N, 3) matrix. Rows that did not converge are NaN.

See Also

`cmtk_read()`, `cmtk_xform()`

Examples

```
## Not run:
reg <- cmtk_read("JFRC2_FCWB.list")
pts <- rbind(c(50, 50, 50), c(100, 100, 20))
back <- cmtk_xform_inv(reg, cmtk_xform(reg, pts))
stopifnot(all.equal(back, pts, tolerance = 1e-4))

## End(Not run)
```

connected_components *Connected components.*

Description

Connected components.

Usage

connected_components(parents)

Arguments

parents Integer vector of 0-based parent indices (roots are < 0).

Value

Integer vector assigning each node a component id.

connected_components_graph
 Connected components of a graph given as an edge list.

Description

The edge-list counterpart of mesh_connected_components, using the same Union-Find: one integer array, no adjacency list.

Usage

connected_components_graph(edges, n_nodes)

Arguments

edges Integer or numeric ($E, 2$) matrix of edges (0-based node indices). Direction is ignored; self-loops and parallel edges are harmless.

n_nodes Integer; total number of nodes. Nodes named by no edge form components of size one.

Value

Integer vector giving, per node, the smallest node index in its component.

contract_nodes	<i>Collapse groups of nodes onto a representative and rewire what is left.</i>
----------------	--

Description

Edges internal to a group are dropped rather than turned into self-loops.

Usage

```
contract_nodes(parents, mapping)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
mapping	Integer vector, one entry per node, giving the 0-based index of the node it collapses onto. A node mapped to itself survives.

Value

List with nodes (0-based indices of the surviving nodes, in their original relative order) and parents (their new 0-based parent indices, -1 for roots, indexing *into* nodes).

contract_vertices	<i>Contract nodes onto new ids, returning the simplified edge list.</i>
-------------------	---

Description

Both endpoints of every edge are pushed through mapping; edges that end up with both ends on the same new node are dropped, and the rest deduplicated. This is `igraph::contract_vertices()` followed by `simplify()`, fused — and, unlike that pair, it does not rewrite a graph object in place, so contracting costs no copy.

Usage

```
contract_vertices(edges, mapping, threads = NULL)
```

Arguments

edges	Integer or numeric ($E, 2$) matrix of edges (0-based node indices).
mapping	Integer vector giving the new 0-based id of each old node. Ids need not be contiguous, but the output is only as compact as the ids you supply.
threads	Optional integer; number of threads. NULL uses all cores.

Value

Integer ($n_{\text{unique}}, 2$) matrix of the surviving edges as $[\text{min}, \text{max}]$ rows.

descendant_counts	<i>How many nodes lie strictly below each node.</i>
-------------------	---

Description

This is what `navis::betweenness_centrality(from_ = ...)` actually computes, and what `navis::find_main_branchpoint = "betweenness")` — its one caller — wants.

Usage

```
descendant_counts(parents, targets = NULL)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
targets	Optional integer vector of 0-based node indices; NULL counts every node, otherwise only these are counted.

Value

Numeric vector of counts, aligned with parents. A node is never its own descendant, so a leaf scores 0 even when it is itself a target. Double rather than integer for the same reason as `betweenness`.

descendants	<i>Every node in the sub-tree below each source.</i>
-------------	--

Description

The replacement for `igraph::subcomponent(mode = "in")`, and what makes "cut the skeleton here" a masking operation rather than a graph rebuild.

Usage

```
descendants(parents, sources)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
sources	Integer vector of 0-based node indices to walk down from.

Value

List of integer vectors, one per source in `sources` order. Each starts with the source itself and is in depth-first pre-order, so a node always precedes its own descendants. An out-of-range source gives an empty vector.

dist_to_root	<i>Return path length from a single node to the root.</i>
--------------	---

Description

Return path length from a single node to the root.

Usage

```
dist_to_root(parents, node)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
node	Integer index of the node to measure from.

Value

Numeric path length (edge count) from node to its root.

downsample_skeleton	<i>Keep every factor-th node of every segment, dropping the rest.</i>
---------------------	---

Description

The plain "make this skeleton smaller" operation: it pays no attention to geometry, so reach for it when the skeleton is already evenly sampled and you just want fewer nodes. Roots, branch points and leafs always survive, so the result is still the same neuron — only its unbranched stretches are sampled factor times more coarsely. See `simplify_rdp()` and `simplify_vw()` for the geometry-aware alternatives.

Usage

```
downsample_skeleton(parents, factor, preserve = NULL, weights = NULL)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
factor	Integer; keep one node in every factor, counting from each segment's distal end. 1 keeps everything; the useful range starts at 2.
preserve	Optional logical vector, one entry per node, marking extra nodes that must survive; NULL for none.
weights	Optional numeric vector of child-to-parent edge weights; NULL returns no weights.

Value

List with nodes (0-based indices of the surviving nodes, in their original relative order), parents (their new 0-based parent indices, -1 for roots, indexing *into* nodes), weights (length of each node's edge to its new parent, i.e. the summed length of the chain it replaces; NULL exactly when weights was NULL) and node_map (one entry per *input* node, giving the 0-based index *into* nodes of the survivor its data belongs to now: itself if it survived, otherwise whichever end of its chain is nearer, measured in weights, ties going towards the root). Total cable length is preserved.

elastix_probe_invertible

Can an Elastix transform be inverted?

Description

Answered **without reading the transform's coefficients**, and so cheaply enough to ask about many files at once.

Usage

elastix_probe_invertible(path)

Arguments

path	Path to a TransformParameters.*.txt file. Its initial-transform chain is followed, however deep.
------	--

Details

A transform cannot be inverted exactly when some step in its chain combines via Add: $T(x) = T_{\text{initial}}(x) + T_{\text{this}}(x) - x$ does not decompose into invertible hops. That fact lives in one short key – but the key sits *after* a coefficient array that runs to tens of megabytes, so answering it honestly used to mean parsing the whole file.

This walks the same chain and applies the same validation, skipping only the numbers: roughly **20x** faster than `elastix_read()` across real registrations, and ~200x on the largest.

Use it when you must label *many* files up front and cannot afford to parse them – building a bridging graph, say, where each edge needs to know whether it can be walked backwards.

Value

TRUE if `elastix_xform_inv()` would work on it.

Errors if the file would not load at all (missing, not an Elastix transform, an unsupported transform kind, binary parameters, a circular chain) – so TRUE is a promise, not a guess.

See Also

`elastix_read()`, `elastix_xform_inv()`

Examples

```
## Not run:
elastix_probe_invertible("TransformParameters.FixedFANC.txt")

## End(Not run)
```

elastix_properties	<i>Properties of an Elastix transform</i>
--------------------	---

Description

Properties of an Elastix transform

Usage

```
elastix_affine(xf)

elastix_kinds(xf)

elastix_grid_size(xf)

elastix_grid_spacing(xf)

elastix_grid_origin(xf)
```

Arguments

xf An elastix_transform, from [elastix_read\(\)](#).

Value

- elastix_affine: the 4 x 4 matrix of the first linear step, or NULL if there is none.
- elastix_kinds: one string per transform in the chain, giving its resolved steps initial-first (e.g. "linear+bspline").
- elastix_grid_size / elastix_grid_spacing / elastix_grid_origin: a (k, 3) matrix, one row per B-spline in the chain, or NULL if the chain is purely linear.

elastix_read

Read an Elastix transform

Description

Reads an Elastix TransformParameters file, ready to apply to points. **Elastix itself does not need to be installed** – there is no call out to transformix, no subprocess and no temporary directory.

Usage

```
elastix_read(path)
```

Arguments

path	Path to a TransformParameters.*.txt file. Pass several to build a chain, applied in order: xyz -> path[1] -> path[2] -> ... -> output. Direction is not fixed here. It is chosen per call – see the invert argument of elastix_xform() , and elastix_xform_inv() – so one object serves every direction and the file is parsed only once. That is worth caring about: real warps run to tens of megabytes.
------	--

Details

A TransformParameters file is already a chain: its InitialTransformParametersFileName is followed recursively, resolved relative to *that file's own directory*. So a four-deep affine -> affine -> B-spline -> B-spline stack loads from its outermost file alone.

The transform is parsed once and can then be applied any number of times, which is what you want when transforming every neuron in a dataset.

Value

An object of class elastix_transform.

See Also

[elastix_xform\(\)](#), [elastix_xform_inv\(\)](#)

Examples

```
## Not run:
xf <- elastix_read("TransformParameters.FixedFANC.txt")
xyzmatrix(n) <- elastix_xform(xf, xyzmatrix(n))

## End(Not run)
```

elastix_xform	<i>Apply an Elastix transform to points</i>
---------------	---

Description

Apply an Elastix transform to points

Usage

```
elastix_xform(
  xf,
  xyz,
  out_of_bounds = c("identity", "nan"),
  invert = FALSE,
  n_cores = NULL,
  progress = FALSE
)
```

Arguments

xf	An elastix_transform, from elastix_read() .
xyz	An (N, 3) matrix of coordinates. A data frame is also accepted – if it has x/y/z columns (in any case) those are used, whatever else it carries, otherwise the first three columns – as is a bare length-3 vector.
out_of_bounds	What to do with points outside a B-spline's control-point grid. "identity" (the default) returns them unchanged , which is exactly what transformix does. "nan" returns NaN instead. The default is silent by nature: a neuron straddling the grid edge comes back partly transformed and looks perfectly fine. Use "nan" when you would rather see the boundary than trust it.
invert	Logical, length 1 or one flag per transform. TRUE traverses that transform backwards – what you need when routing through a bridging graph, where an edge may be walked in either direction. This is not the same as elastix_xform_inv() , which inverts the whole composition (reversing the order <i>and</i> flipping every hop). For a single transform the two agree; for a chain they do not, and only invert can express a mixed-direction traversal.
n_cores	Cap the thread pool. NULL uses all cores.
progress	Show a progress bar.

Value

An (N, 3) matrix of transformed coordinates.

See Also

[elastix_xform_inv\(\)](#)

Examples

```
## Not run:
xf <- elastix_read("TransformParameters.FixedFANC.txt")
elastix_xform(xf, cbind(50, 50, 50))

## End(Not run)
```

elastix_xform_inv	<i>Invert an Elastix transform on points</i>
-------------------	--

Description

Elastix itself cannot do this: transformix only goes forwards, which is why packages that use Elastix ship two separate registration files per brain pair.

Usage

```
elastix_xform_inv(
  xf,
  xyz,
  out_of_bounds = c("identity", "nan"),
  initial_guess = NULL,
  max_iter = 50L,
  seed_iter = 8L,
  tolerance = 1e-09,
  accuracy = 0.001,
  lattice_points = 16000L,
  invert = FALSE,
  n_cores = NULL,
  progress = FALSE
)
```

Arguments

xf	An elastix_transform, from elastix_read() .
xyz	An (N, 3) matrix of coordinates. A data frame is also accepted – if it has x/y/z columns (in any case) those are used, whatever else it carries, otherwise the first three columns – as is a bare length-3 vector.
out_of_bounds	What to do with points outside a B-spline's control-point grid. "identity" (the default) returns them unchanged , which is exactly what transformix does. "nan" returns NaN instead.

	The default is silent by nature: a neuron straddling the grid edge comes back partly transformed and looks perfectly fine. Use "nan" when you would rather see the boundary than trust it.
initial_guess	An (N, 3) matrix of starting points for the solver, in the <i>source</i> space. Rarely needed – the solver seeds itself – but it is what breaks the tie where a target has more than one preimage.
max_iter	Solver budget per point.
seed_iter	Rounds of the fixed-point pre-seed. Zero starts the solver at the target, which loses points wherever the deformation is large.
tolerance	Step-size convergence threshold.
accuracy	Accept a solution only if its residual is within this of the target, in world units. Otherwise the row is NaN.
lattice_points	Size of the global seed lattice – the last-resort start for the few points the cheap seeds fail on. Built once per call and consulted only by points that have already failed. Set to 0 to disable.
invert	The same per-hop flags as on elastix_xform() , composed with this whole-chain inversion: hop <i>i</i> runs <i>forwards</i> here exactly when <code>invert[i]</code> is TRUE.
n_cores	Cap the thread pool. NULL uses all cores.
progress	Show a progress bar.

Details

Linear steps are inverted exactly. Each B-spline warp has no closed-form inverse and is solved per point by damped Gauss-Newton against the analytic Jacobian.

What is guaranteed is **forward-consistency**: `elastix_xform(xf, elastix_xform_inv(xf, y))` returns *y*, to within accuracy. What is *not* guaranteed is that inverting a forward transform returns the point you started from, because a B-spline warp need not be injective – a strongly folded registration maps several points to the same place, and no inverse can recover which one you meant. Points with no preimage at all come back as NaN.

Value

An (N, 3) matrix of coordinates in the source space. Rows with no preimage are NaN.

See Also

[elastix_xform\(\)](#)

exposed_halfedges	<i>The edges a subset is about to expose. Takes the faces before subsetting.</i>
-------------------	--

Description

A face survives only if all three of its corners do, so an edge ends up on a new boundary exactly when it loses a face to the cut but keeps one. Both halves of that test are local to the cut, so — unlike `boundary_halfedges` — this never groups the edges of the whole mesh.

Usage

```
exposed_halfedges(faces, dropped, threads = NULL)
```

Arguments

faces	Integer or numeric (F, 3) matrix of faces <i>before</i> subsetting (0-based vertex indices).
dropped	Logical vector of length V: for each vertex, whether the subset drops it.
threads	Optional integer; number of threads. NULL uses all cores.

Details

Edges that were boundary *already* are left out: they belong to openings the mesh came with, and sealing those is `boundary_halfedges`' job. One consequence is worth knowing: where a cut runs into an opening the mesh came with, what it exposes is an open chain rather than a ring, and `trace_loops` abandons it.

Value

Integer (K, 2) matrix of directed half-edges, with indices into the *original* vertices. Remap them onto the surviving vertices before capping.

fast_hclust	<i>Hierarchical clustering without the 65536 observation limit</i>
-------------	--

Description

A drop-in replacement for `stats::hclust()` that does not stop at 65536 observations, and clusters at a fraction of the memory.

Usage

```
fast_hclust(
  d,
  method = c("ward", "single", "complete", "average", "weighted", "centroid", "median"),
  labels = NULL
)
```

Arguments

d	A dist object, or a bare numeric vector of length $n * (n - 1) / 2$ holding the lower triangle by column.
method	Linkage method: one of "ward", "single", "complete", "average", "weighted", "centroid" or "median". Names follow SciPy, so "ward" is R's "ward.D2" and "weighted" is R's "mcquitty". Note that "centroid" and "median" take plain distances here, whereas <code>stats::hclust()</code> expects squared ones for those two; the heights returned here equal <code>sqrt()</code> of what <code>hclust(d^2, ...)</code> gives.
labels	Optional character vector of n names, or NULL to use the <code>Labels</code> attribute of d when it has one.

Details

Clustering consumes its input as scratch, and R's value semantics forbid writing to the caller's vector, so d is copied once. Clustering straight from a score matrix with `nblast_hclust()` avoids that copy entirely, and is the better route when you have the matrix to hand.

Value

An object of class `hclust`.

Examples

```
d <- dist(matrix(rnorm(40), 20, 2))
h <- fast_hclust(d, method = "average")
# Same tree as stats::hclust for the equivalent method.
all.equal(h$height, hclust(d, method = "average")$height)
```

generate_segments	<i>Generate linear segments while maximising segment lengths.</i>
-------------------	---

Description

Returns a list with segments (a list of integer vectors, one per segment) and lengths (per-segment lengths, or NULL if no weights were supplied).

Usage

```
generate_segments(parents, weights)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
weights	Optional numeric vector of edge weights; NULL returns no lengths.

Value

List with segments (list of integer node-index vectors) and lengths (numeric per-segment lengths, or NULL).

geodesic_clusters	<i>Greedily partition nodes into connected clusters of bounded radius.</i>
-------------------	--

Description

Repeatedly takes an unassigned node as a seed and grows a cluster outwards from it, absorbing any node reachable within `max_dist` that no earlier cluster has already claimed. Collapsing each cluster to its centroid gives a coarser graph whose nodes are spaced by roughly `max_dist`, which is what makes this useful as mesh or skeleton downsampling.

Usage

```
geodesic_clusters(
  edges,
  n_nodes,
  max_dist,
  weights = NULL,
  seeds = NULL,
  precision = 32
)
```

Arguments

edges	Integer or numeric ($E, 2$) matrix of edges (0-based node indices). Treated as undirected.
n_nodes	Integer; total number of nodes. Isolated nodes each become their own cluster.
max_dist	Numeric; maximum distance from a cluster's seed. Must be finite and non-negative.
weights	Optional numeric vector, one length per edge; NULL makes <code>max_dist</code> a hop count.
seeds	Optional integer vector of 0-based nodes to try as seeds, in order of preference. Any node left unassigned afterwards becomes a seed in ascending index order; NULL seeds in ascending index order throughout.
precision	Integer; 32 or 64, the width the search accumulates at. Nothing here returns a distance, so this changes only which answer close ties resolve to.

Details

The radius is the **true geodesic distance from the seed**, not the length of the walk that happened to reach it — so a node close to a seed is never excluded merely because a traversal arrived at it the long way round.

Value

List with labels (integer cluster index per node, contiguous in $[0, n_clusters)$ and numbered in the order the clusters were grown; every node is labelled) and `n_clusters` (integer).

geodesic_distances	<i>Geodesic distances between nodes.</i>
--------------------	--

Description

Geodesic distances between nodes.

Usage

`geodesic_distances(parents, sources, targets, weights, directed)`

Arguments

- | | |
|----------|---|
| parents | Integer vector of 0-based parent indices (roots are < 0). |
| sources | Optional integer vector of source node indices; NULL uses every node. |
| targets | Optional integer vector of target node indices; NULL uses every node. |
| weights | Optional numeric vector of edge weights; NULL counts edges. |
| directed | Logical; if TRUE only traverse edges child-to-parent. |

Value

Numeric matrix of geodesic distances (sources in rows, targets in columns).

geodesic_farthest	<i>Distance to the farthest target for each source.</i>
-------------------	---

Description

The mirror image of `geodesic_nearest`: same linear-time algorithm, but it keeps the farthest rather than the nearest target and never materialises the full distance matrix. Returns a list with distances (distance to the farthest target) and `farthest` (index of that target); sources without a reachable target get `-1`. A source that is itself a target is matched to the farthest *other* target, never to itself.

Usage

```
geodesic_farthest(parents, sources, targets, weights, directed)
```

Arguments

<code>parents</code>	Integer vector of 0-based parent indices (roots are <code>< 0</code>).
<code>sources</code>	Optional integer vector of source node indices; <code>NULL</code> uses every node.
<code>targets</code>	Optional integer vector of target node indices; <code>NULL</code> uses every node.
<code>weights</code>	Optional numeric vector of edge weights; <code>NULL</code> counts edges.
<code>directed</code>	Logical; if <code>TRUE</code> only traverse edges child-to-parent. With non-negative weights the farthest such target is the target ancestor closest to the root.

Value

List with distances (numeric, distance to the farthest target) and `farthest` (integer target index, `-1` when unreachable).

geodesic_farthest_mesh	<i>Distance to the farthest target vertex, for each source vertex of a mesh.</i>
------------------------	--

Description

The mirror image of `geodesic_nearest_mesh`, with the same $O(\text{sources})$ memory footprint. Unlike `nearest`, this cannot stop early — it has to settle every target — but the farthest one then comes for free, because the search settles vertices in increasing order of distance.

Usage

```
geodesic_farthest_mesh(
  faces,
  n_vertices,
  vertices,
  sources,
  targets,
  limit,
  threads,
  precision = 32
)
```

Arguments

faces	Integer or numeric (F, 3) matrix of triangle vertex indices.
n_vertices	Integer; total number of vertices in the mesh.
vertices	Optional numeric (V, 3) matrix of vertex coordinates; NULL counts edges.
sources	Optional integer vector of source vertex indices; NULL uses every vertex.
targets	Optional integer vector of target vertex indices; NULL uses every vertex.
limit	Optional numeric; ignore targets further away than this.
threads	Optional integer; number of threads. NULL uses all cores.
precision	Integer; 32 or 64, the width distances are accumulated at. The result is numeric either way — R has no float32 — so this buys accuracy.

Details

Returns a list with distances and farthest. Sources with no reachable target get -1. A source that is itself a target is matched to a *distinct* target.

Value

A list with distances and farthest.

geodesic_matrix_graph *Geodesic distances over an arbitrary graph given as an edge list.*

Description

The general form of geodesic_matrix_mesh. Unlike geodesic_distances, this makes no tree assumption — cycles are fine.

Usage

```
geodesic_matrix_graph(
  edges,
  n_nodes,
  weights,
  directed,
  sources,
  targets,
  limit,
  threads,
  precision = 32
)
```

Arguments

edges	Integer or numeric ($E, 2$) matrix of edges (0-based node indices).
n_nodes	Integer; total number of nodes.
weights	Optional numeric vector with one length per edge; NULL counts edges. Must be finite and non-negative. Parallel edges collapse to the shortest.
directed	Logical; if TRUE an edge (u, v) may only be traversed from u to v .
sources	Optional integer vector of source node indices; NULL uses every node.
targets	Optional integer vector of target node indices; NULL uses every node.
limit	Optional numeric; ignore nodes further away than this.
threads	Optional integer; number of threads. NULL uses all cores.
precision	Integer; 32 or 64, the width distances are accumulated at. The result is numeric either way — R has no float32 — so this buys accuracy.

Value

Numeric matrix of geodesic distances (sources in rows, targets in columns). Unreachable pairs are -1.

geodesic_matrix_mesh *Geodesic ("along-the-mesh-edge") distances on a triangle mesh.*

Description

The mesh counterpart to `geodesic_distances`, which works on skeletons. A skeleton is a tree, so distances there come from walking to the lowest common ancestor; a mesh is a general cyclic graph, so this runs one Dijkstra per source (or a BFS when unweighted), in parallel.

Usage

```
geodesic_matrix_mesh(
    faces,
    n_vertices,
    vertices,
    sources,
    targets,
    limit,
    threads,
    precision = 32
)
```

Arguments

<code>faces</code>	Integer or numeric ($F, 3$) matrix of triangle vertex indices (0-based).
<code>n_vertices</code>	Integer; total number of vertices in the mesh.
<code>vertices</code>	Optional numeric ($V, 3$) matrix of vertex coordinates. If given, edges are weighted by their euclidean length; if <code>NULL</code> , every edge has weight 1 and the result is a hop count.
<code>sources</code>	Optional integer vector of source vertex indices; <code>NULL</code> uses every vertex.
<code>targets</code>	Optional integer vector of target vertex indices; <code>NULL</code> uses every vertex.
<code>limit</code>	Optional numeric; ignore vertices further away than this.
<code>threads</code>	Optional integer; number of threads. <code>NULL</code> uses all cores.
<code>precision</code>	Integer; 32 or 64, the width distances are accumulated at. The result is numeric either way — R has no float32 — so this buys accuracy.

Details

Note this is the distance *along mesh edges*, not the exact surface geodesic: paths are constrained to run along edges, so on a coarse mesh they overshoot.

Beware the size of the output: a full $V \times V$ matrix is ~107 GB at $V = 164k$. Use `sources` and/or `targets` — unlike `scipy`'s Dijkstra, passing `targets` here means only those columns are ever allocated.

Value

Numeric matrix of geodesic distances (sources in rows, targets in columns). Unreachable pairs are -1.

geodesic_mst_graph	<i>Minimum spanning tree over a subset of graph nodes, by geodesic distance.</i>
--------------------	--

Description

The edge-list form of `geodesic_mst_mesh`, which explains why this never builds the $k \times k$ distance matrix the question seems to call for. Always undirected — a minimum spanning tree of a directed graph is a different problem (an arborescence) with a different algorithm.

Usage

```
geodesic_mst_graph(
    edges,
    n_nodes,
    nodes,
    weights = NULL,
    limit = NULL,
    threads = NULL,
    precision = 32
)
```

Arguments

<code>edges</code>	Integer or numeric ($E, 2$) matrix of edges (0-based node indices).
<code>n_nodes</code>	Integer; total number of nodes.
<code>nodes</code>	Integer vector of 0-based nodes to span. Must be distinct.
<code>weights</code>	Optional numeric vector, one length per edge; NULL counts edges.
<code>limit</code>	Optional numeric; do not join nodes further apart than this.
<code>threads</code>	Optional integer; number of threads. NULL uses all cores.
<code>precision</code>	Integer; 32 or 64, the width distances are accumulated at. The result is numeric either way — R has no float32 — so this buys accuracy.

Value

List with `edges` (integer ($M, 2$) matrix of 0-based positions in nodes, ascending by weight) and `weights` (numeric geodesic distance across each).

geodesic_mst_mesh	<i>Minimum spanning tree over a subset of mesh vertices, by geodesic distance.</i>
-------------------	--

Description

The tree that reconnects a scatter of surviving vertices through the mesh they were carved out of — the last step of a skeletonization, where the mesh has been thinned to a few thousand vertices that must be rejoined along the surface rather than through space.

Usage

```
geodesic_mst_mesh(
    faces,
    n_vertices,
    nodes,
    vertices = NULL,
    limit = NULL,
    threads = NULL,
    precision = 32
)
```

Arguments

faces	Integer or numeric (F, 3) matrix of triangle vertex indices (0-based).
n_vertices	Integer; total number of vertices.
nodes	Integer vector of 0-based vertices to span. Must be distinct.
vertices	Optional numeric (V, 3) matrix of vertex coordinates. If given, edges are weighted by their euclidean length; if NULL, distances are hop counts.
limit	Optional numeric; do not join vertices further apart than this. The result is then a <i>forest</i> when that disconnects the subset. Unlike a matrix route's limit this also prunes the sweep, so it buys time rather than merely discarding results.
threads	Optional integer; number of threads. NULL uses all cores.
precision	Integer; 32 or 64, the width distances are accumulated at. The result is numeric either way — R has no float32 — so this buys accuracy.

Details

The obvious route is to ask for the $k \times k$ geodesic matrix and hand it to a matrix MST. That materialises k^2 distances to use $k - 1$ of them — 400 MB at $k = 10000$, before the $O(k^2)$ MST itself — and needs k separate searches to fill. This never forms the matrix: following Mehlhorn's distance-network construction, one multi-source search partitions every vertex by which of nodes is nearest, and each mesh edge whose endpoints fall in different cells offers one candidate. An MST over those is an MST of the full distance network, so one sweep and one Kruskal replace k searches and a dense matrix.

Value

List with edges (integer (M, 2) matrix of **0-based positions in** nodes, not vertex indices — so nodes[edges + 1] maps back — ascending by weight) and weights (numeric geodesic distance across each). The returned weights are exactly the geodesic distances between the pairs they join, so they are usable as lengths and not merely as an ordering.

geodesic_nearest	<i>Distance to the nearest target for each source.</i>
------------------	--

Description

Memory-efficient companion to geodesic_distances that never materialises the full distance matrix. Returns a list with distances (distance to the nearest target) and nearest (index of that target); sources without a reachable target get -1.

Usage

```
geodesic_nearest(parents, sources, targets, weights, directed)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
sources	Optional integer vector of source node indices; NULL uses every node.
targets	Optional integer vector of target node indices; NULL uses every node.
weights	Optional numeric vector of edge weights; NULL counts edges.
directed	Logical; if TRUE only traverse edges child-to-parent.

Value

List with distances (numeric, distance to the nearest target) and nearest (integer target index, -1 when unreachable).

geodesic_nearest_mesh	<i>Distance to the nearest target vertex, for each source vertex of a mesh.</i>
-----------------------	---

Description

A memory-efficient alternative to geodesic_matrix_mesh: it keeps only the nearest target and the distance to it, so the output is O(sources) rather than O(sources * targets). It is also faster, because the search stops at the first target it settles rather than exploring the whole component.

Usage

```
geodesic_nearest_mesh(
  faces,
  n_vertices,
  vertices,
  sources,
  targets,
  limit,
  threads,
  precision = 32
)
```

Arguments

<code>faces</code>	Integer or numeric ($F, 3$) matrix of triangle vertex indices.
<code>n_vertices</code>	Integer; total number of vertices in the mesh.
<code>vertices</code>	Optional numeric ($V, 3$) matrix of vertex coordinates; NULL counts edges.
<code>sources</code>	Optional integer vector of source vertex indices; NULL uses every vertex.
<code>targets</code>	Optional integer vector of target vertex indices; NULL uses every vertex.
<code>limit</code>	Optional numeric; ignore targets further away than this.
<code>threads</code>	Optional integer; number of threads. NULL uses all cores.
<code>precision</code>	Integer; 32 or 64, the width distances are accumulated at. The result is numeric either way — R has no float32 — so this buys accuracy.

Details

Returns a list with distances and nearest (the vertex index of that nearest target). Sources with no reachable target get -1. A source that is itself a target is matched to its nearest *other* target, never to itself.

Value

A list with distances and nearest.

<code>geodesic_pairs</code>	<i>Geodesic distances for explicit pairs of nodes.</i>
-----------------------------	--

Description

`sources` and `targets` are parallel arrays of node indices; the returned vector holds the distance between each (source, target) pair.

Usage

```
geodesic_pairs(parents, sources, targets, weights, directed, threads = NULL)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
sources	Integer vector of source node indices.
targets	Integer vector of target node indices (same length as sources).
weights	Optional numeric vector of edge weights; NULL counts edges.
directed	Logical; if TRUE only traverse edges child-to-parent.
threads	Optional integer; number of threads. NULL uses all cores.

Value

Numeric vector with the distance of each (source, target) pair.

geodesic_path	<i>The shortest route from one source to each target, as node sequences.</i>
---------------	--

Description

The shortest route from one source to each target, as node sequences.

Usage

```
geodesic_path(
  edges,
  n_nodes,
  source,
  targets,
  weights = NULL,
  directed = FALSE,
  precision = 32
)
```

Arguments

edges	Integer or numeric (E, 2) matrix of edges (0-based node indices).
n_nodes	Integer; total number of nodes.
source	Integer; the 0-based node to start from.
targets	Integer vector of 0-based nodes to reach.
weights	Optional numeric vector, one length per edge; NULL counts edges.
directed	Logical; if TRUE an edge (u, v) may only be traversed from u to v.
precision	Integer; 32 or 64, the width the search accumulates at. Nothing here returns a distance, so this changes only which answer close ties resolve to.

Value

List of integer vectors, one per target in targets order, each running source-first to target-last. An unreachable target gives an empty vector.

geodesic_predecessors *Shortest path trees over a graph — distances and the route to each node.*

Description

The predecessor-returning counterpart to `geodesic_matrix_graph`. Use this when you need the path itself; use `geodesic_matrix_graph` when the distance is enough, and `geodesic_path` when you want the node sequences rather than the raw chains.

Usage

```
geodesic_predecessors(
    edges,
    n_nodes,
    weights = NULL,
    directed = FALSE,
    sources = NULL,
    limit = NULL,
    threads = NULL,
    precision = 32
)
```

Arguments

<code>edges</code>	Integer or numeric ($E, 2$) matrix of edges (0-based node indices).
<code>n_nodes</code>	Integer; total number of nodes.
<code>weights</code>	Optional numeric vector, one length per edge; <code>NULL</code> counts edges. Must be finite and non-negative. Zero weights are explicitly allowed — they are how a penalised-path search makes an already-extracted route free to re-traverse.
<code>directed</code>	Logical; if <code>TRUE</code> an edge (u, v) may only be traversed from u to v .
<code>sources</code>	Optional integer vector of 0-based source nodes, one tree each; <code>NULL</code> uses every node.
<code>limit</code>	Optional numeric; ignore nodes further away than this.
<code>threads</code>	Optional integer; number of threads. <code>NULL</code> uses all cores.
<code>precision</code>	Integer; 32 or 64, the width distances are accumulated at. The result is numeric either way — R has no float32 — so this buys accuracy.

Details

Because this takes a bare edge list there is no index to build or invalidate between calls, which is what algorithms that re-weight the graph every iteration need — TEASAR zeroes the edges along each path it extracts, then searches again.

Value

List with distances (numeric (n_sources, n_nodes) matrix, -1 where unreachable) and predecessors (integer matrix of the same shape giving, per node, the node before it on the shortest path back to that row's source; -1 for the source itself and for unreachable nodes, so one ≥ 0 test both walks the path and terminates it).

get_num_threads	<i>Number of threads available for parallel work in this session.</i>
-----------------	---

Description

Note that asking builds the thread pool if it does not exist yet — which is then exactly what makes a subsequent set_num_threads() fail. Set first, ask second.

Usage

```
get_num_threads()
```

Value

Integer; the number of threads.

has_cycles	<i>Check whether the tree contains cycles.</i>
------------	--

Description

Check whether the tree contains cycles.

Usage

```
has_cycles(parents)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
---------	--

Value

Logical; TRUE if the parent structure contains a cycle.

heal_skeleton	<i>Heal a fragmented skeleton by reconnecting its fragments.</i>
---------------	--

Description

Convenience wrapper that finds the minimal-length set of new edges between the skeleton's connected components (see `stitch_fragments()`) and regenerates a single rooted tree from them (see `reroot_rewire()`).

Usage

```
heal_skeleton(
  parents,
  x,
  y,
  z,
  method,
  max_dist,
  min_size,
  mask,
  radius,
  use_radius,
  threads = NULL
)
```

Arguments

<code>parents</code>	Integer vector of 0-based parent indices (roots are < 0), e.g. from <code>node_indices()</code> .
<code>x, y, z</code>	Numeric vectors of node coordinates, one entry per node.
<code>method</code>	Character; "ALL" lets any node form a new edge, "LEAFS" restricts new edges to leaf and root nodes (faster, occasionally suboptimal attachment points).
<code>max_dist</code>	Optional numeric maximum length for any single new edge; gaps larger than this are left unhealed, so the result may stay fragmented. NULL means no limit.
<code>min_size</code>	Optional integer; fragments with fewer than this many nodes are excluded from healing and stay disconnected. NULL heals every fragment.
<code>mask</code>	Optional logical vector restricting which nodes may be used as endpoints for a new edge; combined with <code>method</code> . NULL allows every node.
<code>radius</code>	Optional numeric vector of node radii, one entry per node. Only required when <code>use_radius</code> is set.
<code>use_radius</code>	TRUE/FALSE, a number, or NULL. If set, node radii are taken into account when measuring distances, which prioritises connecting fragments of similar calibre. A number weights the effect: higher values give radius more influence (TRUE means 1). To keep this robust we use the mean radius of the segment a node belongs to, not the node's own radius. Note that <code>max_dist</code> is then measured in this augmented space too.

threads Optional integer; number of threads. NULL uses all cores. Worth capping when healing many skeletons across several processes — see `set_num_threads()`, which is cheaper than passing this on every call.

Value

Integer vector of new 0-based parent indices (roots are -1). If the skeleton could be fully healed this is a single tree with one root.

leaf_order	<i>Dendrogram leaf order</i>
------------	------------------------------

Description

The order to place the leaves in so a dendrogram draws without crossing branches – what `stats::hclust()` returns in its order element, computed from a merge matrix alone.

Usage

```
leaf_order(x)
```

Arguments

x An `stats::hclust()` object, or its (n - 1, 2) merge matrix: negative entries are observations (-j is observation j), positive entries are the cluster formed at that earlier step.

Details

You do not need this for an untouched `hclust`: it already carries the same ordering. It is for a merge matrix you built or edited yourself, where order is missing or no longer matches – a dendrogram drawn with an ordering from a *different* child order is the crossing-branches mess this exists to avoid.

The walk is iterative, so a 200k-observation chain does not need 200k nested calls.

Value

An integer vector of length n holding each observation's position in the drawing order, left to right – a permutation of 1:n, in the same form as `hclust$order`.

Examples

```
set.seed(1)
h <- fast_hclust(dist(matrix(rnorm(20), 10, 2)), method = "average")
identical(leaf_order(h), h$order)
```

level_set_components	<i>Connected components of every level set at once.</i>
----------------------	---

Description

Given a label per node, finds the connected components of each subgraph induced by the nodes sharing a label — all labels in a single pass, by unioning an edge only when its two endpoints agree. This is the inner loop of wavefront-style mesh skeletonization, where the label is a binned geodesic distance and each component is one ring around the structure.

Usage

```
level_set_components(edges, n_nodes, labels)
```

Arguments

edges	Integer or numeric (E, 2) matrix of edges (0-based node indices).
n_nodes	Integer; total number of nodes.
labels	Integer vector with one label per node. Negative labels mark excluded nodes: they join no component and come back as -1. That is what lets you feed in the output of a search that could not reach everything — <code>geodesic_matrix_*</code> returns -1 for unreachable — rather than lumping every unreachable node into one bogus level.

Details

Done conventionally that loop costs one subgraph construction plus one component search per distinct label; here it is one sweep over the edges.

Value

List with `ids` (integer component index per node in $[0, n_components)$, or -1 for excluded nodes; assigned in order of first appearance scanning nodes low to high) and `n_components` (integer).

longest_path	<i>The longest path from any node to its root.</i>
--------------	--

Description

Not the NP-hard general problem: in a rooted forest every maximal path is fixed by its start node, so this is a distances-to-root question.

Usage

```
longest_path(parents, weights = NULL)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
weights	Optional numeric vector of child-to-parent edge weights; NULL counts edges.

Value

Integer vector of 0-based node indices along the path, **distal first** — so the first element is the far end and the last is a root. Ties break towards the lowest node index.

longest_paths	<i>The n longest paths, each peeled off before the next is sought.</i>
---------------	--

Description

The n longest paths, each peeled off before the next is sought.

Usage

```
longest_paths(parents, n, weights = NULL, min_length = NULL)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
n	Integer; how many paths to take.
weights	Optional numeric vector of child-to-parent edge weights; NULL counts edges.
min_length	Optional numeric; stop once the next path is no longer than this. Note it measures the path's whole <i>catchment</i> — every edge whose parent lies on the path, so each twig hanging off it contributes its first edge too — and that hitting it stops the search rather than skipping one path. Both are inherited from <code>navis::split_into_fragments</code> , where they are load-bearing.

Value

List of up to n integer vectors of 0-based node indices, longest first, each distal-first.

 mesh_connected_components

Connected components of a triangle mesh

Description

Three readings of "connected", chosen with connectivity, each strictly finer than the one before it:

"vertex" (**the default**) Joins two vertices whenever a face names them both, so a face joins its three corners. One label per *vertex*.

"face" Joins two faces wherever they share an **edge**, not merely a corner. One label per *face*.

"manifold" Joins two faces only across an edge carrying **exactly two** of them. Likewise one label per *face*.

Usage

```
mesh_connected_components(  
    faces,  
    n_vertices = NULL,  
    connectivity = c("vertex", "face", "manifold"),  
    threads = NULL  
)
```

Arguments

faces	Integer or numeric (F, 3) matrix of triangle vertex indices (0-based).
n_vertices	Integer; total number of vertices in the mesh. Belongs to connectivity = "vertex", where it is required: it sets the length of the result, and vertices in no face are components of size one.
connectivity	One of "vertex" (default), "face" or "manifold"; what two faces must share to count as connected - a corner, any edge, or an edge carrying exactly two faces.
threads	Integer cap on the thread count for this call, or NULL for the process-wide pool. Belongs to connectivity = "face" and "manifold".

Details

Each step drops a kind of junction. Going from "vertex" to "face" drops the pinch points: two triangles meeting at a single corner are one component under the first and two under the second, because there is no edge to step across. Going from "face" to "manifold" drops the seams: three sheets meeting along one edge are one component under "face" and three under "manifold", because an edge that deep belongs to no single surface.

So pick by what the components are *for*. "vertex" answers "can these vertices reach each other along mesh edges" - the question [geodesic_matrix_mesh\(\)](#) answers with distances. "face" splits a mesh into the pieces you could walk across. "manifold" splits it into pieces that are *surfaces*,

each with a well-defined inside, which is what you want before asking a piece for its volume or its winding; it reproduces trimesh's `face_adjacency` exactly. There is no per-vertex form of the face answers, and that is not a gap in the interface: a pinch vertex belongs to several face components at once.

All three are Union-Find with path-halving. "vertex" builds no adjacency at all, one serial sweep over a single integer vector; the other two group the 3F edges the faces name first, which is a parallel sort and the only reason they take threads. They then differ by one test on how many faces each edge came back with, so "manifold" costs no more than "face".

An edge with a single face on it - the boundary of an open mesh - joins nothing under any reading; there is no second face to join it to. Self-loop edges from degenerate faces are kept throughout, as in `unique_edges()`.

Value

Integer vector: one entry per vertex giving the smallest vertex index in its component ("vertex"), or one entry per face giving the smallest face index in its component ("face" and "manifold").

See Also

`connected_components_graph()`, the same question from an edge list.

Examples

```
# Two triangles pinched together at vertex 2 and joined nowhere else.
faces <- matrix(c(0, 1, 2, 2, 3, 4), ncol = 3, byrow = TRUE)

# The vertex graph walks straight through the pinch...
mesh_connected_components(faces, n_vertices = 5)

# ...the faces cannot step across it.
mesh_connected_components(faces, connectivity = "face")

# Three fins meeting along the spine (1, 2): one component across any shared edge,
# three once the spine is too deep to belong to a single surface.
fins <- matrix(c(1, 2, 3, 1, 2, 4, 1, 2, 5), ncol = 3, byrow = TRUE)
mesh_connected_components(fins, connectivity = "face")
mesh_connected_components(fins, connectivity = "manifold")
```

`minimum_spanning_tree` *Minimum (or maximum) spanning forest of an undirected graph.*

Description

Kruskal's algorithm on the same Union-Find as the component search: sort the edges by weight, keep the ones that join two different components. Disconnected input is fine — each component contributes its own tree.

Usage

```

minimum_spanning_tree(
  edges,
  n_nodes,
  weights = NULL,
  maximize = FALSE,
  threads = NULL,
  precision = 32
)

```

Arguments

edges	Integer or numeric (E, 2) matrix of edges (0-based node indices).
n_nodes	Integer; total number of nodes.
weights	Optional numeric vector, one weight per edge; NULL treats every edge as equal. Must be finite; negative weights are allowed.
maximize	Logical; return the <i>maximum</i> spanning forest instead. This exists so you do not have to pass 1 / weights to invert the ordering — a transform that both loses precision and blows up on the zero weights that legitimately occur.
threads	Optional integer; number of threads. NULL uses all cores.
precision	Integer; 32 or 64, the width the search accumulates at. Nothing here returns a distance, so this changes only which answer close ties resolve to.

Value

Integer vector of 0-based **row indices into** edges, ordered by weight — not the edges themselves, so you can index whatever per-edge data you hold with the same vector. Remember to add 1 before using it to subset an R matrix.

mls_affine	<i>The global affine of a moving least squares transform</i>
------------	--

Description

Moving least squares is *locally* weighted – every point effectively gets its own affine – so no single matrix describes it. This returns the *global* affine: the least-squares fit of the source landmarks onto the target landmarks, which is what the warp converges to far from them, where the distance weights even out.

Usage

```
mls_affine(reg)
```

Arguments

reg	An mls_transform, from mls_transform() .
-----	--

Value

A (4, 4) homogeneous transformation matrix, with last row `c(0, 0, 0, 1)`.

See Also

`mls_transform()`, `tps_affine()`

Examples

```
src <- rbind(c(0, 0, 0), c(1, 0, 0), c(0, 1, 0), c(0, 0, 1), c(1, 1, 1))
trg <- src * 2 + matrix(c(3, 4, 5), nrow(src), 3, byrow = TRUE)
mls_affine(mls_transform(src, trg))
```

mls_transform

Build a moving least squares transform from landmark pairs

Description

The affine flavour of the algorithm published in Schaefer et al. (2006). Unlike a thin-plate spline there is no fit: every point gets its *own* affine, solved on the fly from all landmarks weighted by inverse squared distance. Construction is therefore free and `mls_xform()` is the entire cost.

Usage

```
mls_transform(source, target, direction = c("forward", "inverse"))
```

Arguments

source	An (M, 3) matrix of source landmarks. A data frame is also accepted – if it has x/y/z columns (in any case) those are used, whatever else it carries, otherwise the first three columns.
target	An (M, 3) matrix of target landmarks, one per source landmark.
direction	"forward" (default), or "inverse" to treat the target as the source and vice versa. Note this fits the warp in the opposite direction; it is not an exact inverse, which moving least squares does not have. It is free, though – no refit.

Details

Results are similar to but not identical with `tps_transform()`; which suits a given pair of spaces is worth checking empirically.

Value

An object of class `mls_transform`.

References

Schaefer S, McPhail T, Warren J (2006). "Image deformation using moving least squares." *ACM Transactions on Graphics* 25(3):533-540. doi:[10.1145/1141911.1141920](https://doi.org/10.1145/1141911.1141920)

See Also

[mls_xform\(\)](#), [tps_transform\(\)](#)

Examples

```
src <- rbind(c(0, 0, 0), c(10, 10, 10), c(100, 100, 100), c(80, 10, 30))
trg <- rbind(c(1, 15, 5), c(9, 18, 21), c(80, 99, 120), c(5, 10, 80))
mls <- mls_transform(src, trg)
mls

# a point sitting on a landmark maps to its partner
stopifnot(all.equal(mls_xform(mls, src), trg, tolerance = 1e-6))

# and the inverse direction maps them back
back <- mls_transform(src, trg, direction = "inverse")
stopifnot(all.equal(mls_xform(back, trg), src, tolerance = 1e-6))
```

mls_xform	<i>Apply a moving least squares transform to points</i>
-----------	---

Description

Apply a moving least squares transform to points

Usage

```
mls_xform(reg, xyz, n_cores = NULL)
```

Arguments

reg	An <code>mls_transform</code> , from mls_transform() .
xyz	An (N, 3) matrix of coordinates. A data frame is also accepted – if it has x/y/z columns (in any case) those are used, whatever else it carries, otherwise the first three columns – as is a bare length-3 vector. <code>nat::xyzmatrix()</code> gives you this for a neuron.
n_cores	Number of threads. NULL (default) uses all cores.

Value

An (N, 3) matrix of transformed coordinates.

See Also

[mls_transform\(\)](#), [mls_affine\(\)](#)

Examples

```
src <- rbind(c(0, 0, 0), c(10, 10, 10), c(100, 100, 100), c(80, 10, 30))
trg <- rbind(c(1, 15, 5), c(9, 18, 21), c(80, 99, 120), c(5, 10, 80))
mls <- mls_transform(src, trg)
mls_xform(mls, c(50, 50, 50))
```

nblast

Forward NBLAST of every query neuron against every target neuron.

Description

Returns an (n_query, n_target) score matrix.

Usage

```
nblast(
  q_points,
  q_vects,
  t_points,
  t_vects,
  q_alphas,
  t_alphas,
  smat_values,
  dist_edges,
  dot_edges,
  normalize,
  limit_dist,
  n_cores,
  precision,
  progress
)
```

Arguments

q_points	List of (N, 3) numeric matrices of query point coordinates.
q_vects	List of (N, 3) numeric matrices of query tangent vectors.
t_points	List of (N, 3) numeric matrices of target point coordinates.
t_vects	List of (N, 3) numeric matrices of target tangent vectors.
q_alphas	Optional list of per-point alpha vectors for the queries; NULL disables alpha weighting.
t_alphas	Optional list of per-point alpha vectors for the targets.

smat_values	Numeric scoring matrix, or NULL for the built-in FCWB matrix.
dist_edges	Numeric vector of distance bin edges for smat_values.
dot_edges	Numeric vector of dot-product bin edges for smat_values.
normalize	Logical; normalise each score by the query self-match score.
limit_dist	Optional numeric distance cut-off; NULL disables it.
n_cores	Optional integer thread count; NULL or ≤ 0 uses all cores.
precision	Integer; compute in 32- or 64-bit floats.
progress	Logical; display a progress bar.

Value

Numeric (n_query, n_target) score matrix.

nblast_allbyall	<i>All-by-all forward NBLAST.</i>
-----------------	-----------------------------------

Description

points/vects are lists of (N, 3) matrices (one per neuron). Returns an (n, n) score matrix; cell (i, j) is query i against target j.

Usage

```
nblast_allbyall(
  points,
  vects,
  alphas,
  smat_values,
  dist_edges,
  dot_edges,
  normalize,
  limit_dist,
  n_cores,
  precision,
  progress
)
```

Arguments

points	List of (N, 3) numeric matrices of neuron point coordinates.
vects	List of (N, 3) numeric matrices of unit tangent vectors, one per neuron and aligned with points.
alphas	Optional list of per-point alpha (anisotropy) vectors; NULL disables alpha weighting.

smat_values	Numeric scoring matrix, or NULL for the built-in FCWB matrix.
dist_edges	Numeric vector of distance bin edges for smat_values.
dot_edges	Numeric vector of dot-product bin edges for smat_values.
normalize	Logical; normalise each score by the query self-match score.
limit_dist	Optional numeric distance cut-off; NULL disables it.
n_cores	Optional integer thread count; NULL or ≤ 0 uses all cores.
precision	Integer; compute in 32- or 64-bit floats.
progress	Logical; display a progress bar.

Value

Numeric (n, n) score matrix; cell (i, j) is query i vs target j.

nblast_dist	<i>Condensed distances from a score matrix</i>
-------------	--

Description

Symmetrises a score matrix, converts it to distances and condenses it to the upper triangle in a single fused pass, returning a `stats::dist()` object. The only allocation is the result itself, where `as.dist(1 - (m + t(m)) / 2)` builds several full n x n copies on the way.

Usage

```
nblast_dist(
  scores,
  symmetry = c("mean", "min", "max", "none"),
  transform = c("one_minus", "none"),
  labels = NULL,
  n_cores = NULL
)
```

Arguments

scores	Square (n, n) numeric score matrix. The diagonal is not read.
symmetry	How to combine cell (i, j) with (j, i): "mean" (default), "min", "max", or "none".
transform	"one_minus" (default) for 1 - score, or "none" if the values are already distances.
labels	Optional character vector of n names, or NULL. Falls back to the matrix's row names.
n_cores	Optional integer thread count; NULL uses all cores.

Details

R stores a dist as the lower triangle by column, which for a symmetric matrix is element-for-element the same sequence as the upper triangle by row, so no rearranging is needed.

Value

An object of class `dist` with $n * (n - 1) / 2$ entries.

See Also

[nblast_hclust\(\)](#) to go straight to a dendrogram without materialising the distances at all.

Examples

```
m <- matrix(c(1, 0.75, 0.25, 1), 2, 2)
nblast_dist(m)
```

nblast_hclust

Hierarchical clustering of a score matrix

Description

Clusters a square score matrix – typically NBLAST output from [nblast_allbyall\(\)](#) – and returns a standard [stats::hclust\(\)](#) object.

Usage

```
nblast_hclust(
  scores,
  method = c("ward", "single", "complete", "average", "weighted", "centroid", "median"),
  symmetry = c("mean", "min", "max", "none"),
  transform = c("one_minus", "none"),
  labels = NULL,
  n_cores = NULL
)
```

Arguments

scores	Square (n, n) numeric score matrix. The diagonal is never read, so self-scores may be left on it.
method	Linkage method: one of "ward", "single", "complete", "average", "weighted", "centroid" or "median". Names follow SciPy, so "ward" is R's "ward.D2" and "weighted" is R's "mcquitty". Note that "centroid" and "median" take plain distances here, whereas stats::hclust() expects squared ones for those two; the heights returned here equal <code>sqrt()</code> of what <code>hclust(d^2, ...)</code> gives.

symmetry	How to combine cell (i, j) with (j, i), since NBLAST is not symmetric: "mean" (the default), "min", "max", or "none" to take the upper triangle as given. Matches the symmetry argument of <code>nblast_allbyall()</code> ; use "none" if the matrix is already symmetric, which is also the fastest path.
transform	"one_minus" (the default) turns a similarity into a distance via $1 - \text{score}$; "none" treats the values as distances already.
labels	Optional character vector of n names for the observations, or NULL. Falls back to the matrix's row names.
n_cores	Optional integer thread count for the condensing pass; NULL uses all cores.

Details

Unlike `stats::hclust()`, which refuses more than 65536 observations, this has no size ceiling beyond available memory. That limit is a hard blocker for whole-brain connectome work, where 100k-200k neurons is routine.

Symmetrising, the score-to-distance transform and condensing to the upper triangle are **fused into a single pass**, and the resulting buffer is then clustered in place. The idiomatic R spelling, `hclust(as.dist(1 - (m + t(m)) / 2))`, materialises three further $n \times n$ matrices before clustering even begins, which at these sizes is where the memory goes.

Value

An object of class `hclust`, so `stats::cutree()`, `plot()` and `stats::as.dendrogram()` all work on it directly.

See Also

`nblast_dist()` for the condensed distances on their own, `fast_hclust()` to cluster an existing `dist`.

Examples

```
# A score matrix with two obvious pairs.
m <- matrix(c(
  1.00, 0.90, 0.20, 0.10,
  0.70, 1.00, 0.30, 0.15,
  0.25, 0.35, 1.00, 0.80,
  0.05, 0.10, 0.60, 1.00
), 4, 4, byrow = TRUE)

h <- nblast_hclust(m, method = "average")
cutree(h, 2)
```

nblast_knn	<i>k nearest neighbours under NBLAST, without building the score matrix</i>
------------	---

Description

At connectomics scale an all-by-all is the wrong shape for a k-NN question: 164k neurons is 2.7e10 pairs and a 107 GB matrix, even though the answer wanted from it is a small k-NN graph (typically to feed a UMAP embedding). This computes that graph directly, in three stages: each neuron becomes a coarse voxel-occupancy signature; the `n_candidates` most similar neurons per row are shortlisted from those signatures; and the **exact** NBLAST score is then computed for the shortlisted pairs only.

Usage

```
nblast_knn(
  points,
  vects,
  target = NULL,
  target_vects = NULL,
  k = 20L,
  symmetry = c("mean", "forward", "min", "max"),
  n_candidates = 200L,
  alphas = NULL,
  target_alphas = NULL,
  smat = NULL,
  dist_edges = NULL,
  dot_edges = NULL,
  normalize = TRUE,
  limit_dist = NULL,
  voxel = 20,
  n_dirs = 3L,
  splat = TRUE,
  n_cores = NULL,
  precision = 32L,
  progress = FALSE
)
```

Arguments

<code>points</code>	List of (N, 3) numeric matrices of query point coordinates, one per neuron.
<code>vects</code>	List of (N, 3) numeric matrices of unit tangent vectors, aligned with points.
<code>target</code>	Optional list of (N, 3) target point matrices. When given, neighbours are searched among the targets and the returned indices address <i>them</i> ; this is the k-NN counterpart of <code>nblast()</code> . NULL (the default) runs the all-by-all form over points, excluding self-matches.
<code>target_vects</code>	Tangent vectors for target; required when target is given.

k	Integer; neighbours to return per neuron.
symmetry	How the two directions of a pair are combined <i>before</i> the top-k cut. This matters more than for a full matrix: with a matrix you can symmetrise afterwards against the transpose, but once only k neighbours per row are kept the transpose is gone. The asymmetry is real – a small neuron contained in a large one scores high one way and low the other – so "mean" is the default. "forward" keeps each row's own forward score.
n_candidates	Integer; shortlist size per neuron, and the one recall/cost knob. Measured recall@20 on 163,976 real neurons: 0.91 at 50, 0.97 at 100, 0.99 at 200, 0.996 at 400. The budget needed to hold a given recall grows only about logarithmically with the number of neurons.
alphas, target_alphas	Optional lists of per-point alpha (anisotropy) vectors; NULL disables alpha weighting.
smat	Optional scoring matrix, or NULL for the built-in FCWB matrix.
dist_edges, dot_edges	Bin edges for smat; required when it is given.
normalize	Logical; divide each score by the query self-match score, so a perfect self-match is 1.
limit_dist	Optional numeric distance cut-off; NULL disables it and "auto" takes it from the scoring matrix.
voxel	Numeric; signature voxel edge, in the units of points (um for the FCWB matrix). 10-20 measured equivalently.
n_dirs	Integer; tangent-direction bins for the signature (1 disables them). 3 is the cheap "which axis dominates" case; larger values use a Fibonacci half-sphere.
splat	Logical; trilinearly spread each point over its 8 surrounding voxels. Worth about 0.05 recall@20.
n_cores	Optional integer thread count; NULL or <= 0 uses all cores.
precision	Integer; 32 or 64, the width scores are returned at. The scoring maths always runs in double precision.
progress	Logical; display a progress bar.

Details

The pre-filter is sound because the FCWB scoring matrix has finite support – beyond its last distance bin (40 um) every cell is about -10, so neurons that do not overlap in space score at that floor whatever their shape. Only *which* neurons make the shortlist is approximate; every returned score is an exact NBLAST value, because a neuron belonging in the true top-k outranks the global k-th and so cannot be dropped by the rerank once shortlisted.

Measured on 163,976 zebrafish neurons: recall@20 = 0.99 at the default n_candidates = 200, scoring 0.16% of the pairs and taking about 5 minutes against an estimated 35 hours for the exact all-by-all.

Value

A list with `idx`, an integer (`n_query`, `k`) matrix of **1-based** neighbour indices in descending score order, and `scores`, the matching numeric matrix. Rows with fewer than `k` available neighbours are padded with NA in both.

See Also

`nblast_allbyall()` for the full matrix, `nblast_hclust()` to cluster one.

Examples

```
set.seed(1)
pts <- lapply(1:6, function(i) {
  base <- matrix(rnorm(60), ncol = 3)
  base + matrix(rep(c(i, 0, 0), each = 20), ncol = 3)
})
vects <- lapply(pts, function(p) {
  v <- matrix(0, nrow(p), 3)
  v[, 1] <- 1
  v
})
nn <- nblast_knn(pts, vects, k = 2)
nn$idx
nn$scores
```

nblast_pairs

Forward NBLAST for a set of (query, target) index pairs.

Description

`q_idx/t_idx` are 0-based indices into the query/target lists; element `k` of the result is query `q_idx[k]` against target `t_idx[k]`.

Usage

```
nblast_pairs(
  q_points,
  q_vects,
  t_points,
  t_vects,
  q_idx,
  t_idx,
  q_alphas,
  t_alphas,
  smat_values,
  dist_edges,
  dot_edges,
  normalize,
```

```

    limit_dist,
    n_cores,
    precision,
    progress
)

```

Arguments

q_points	List of (N, 3) numeric matrices of query point coordinates.
q_vects	List of (N, 3) numeric matrices of query tangent vectors.
t_points	List of (N, 3) numeric matrices of target point coordinates.
t_vects	List of (N, 3) numeric matrices of target tangent vectors.
q_idx	Integer vector of 0-based query indices, one per pair.
t_idx	Integer vector of 0-based target indices (same length as q_idx).
q_alphas	Optional list of per-point alpha vectors for the queries; NULL disables alpha weighting.
t_alphas	Optional list of per-point alpha vectors for the targets.
smat_values	Numeric scoring matrix, or NULL for the built-in FCWB matrix.
dist_edges	Numeric vector of distance bin edges for smat_values.
dot_edges	Numeric vector of dot-product bin edges for smat_values.
normalize	Logical; normalise each score by the query self-match score.
limit_dist	Optional numeric distance cut-off; NULL disables it.
n_cores	Optional integer thread count; NULL or <= 0 uses all cores.
precision	Integer; compute in 32- or 64-bit floats.
progress	Logical; display a progress bar.

Value

Numeric vector of scores, one per (query, target) pair.

node_indices	<i>For each node ID in parents find its index in nodes.</i>
--------------	---

Description

Importantly this is 0-indexed to match indexing in Rust. Roots will have parent index -1.

Usage

```
node_indices(nodes, parents)
```

Arguments

nodes	Integer vector of node IDs.
parents	Integer vector of parent IDs, one per node; roots use their own ID or a negative value.

Value

Integer vector of 0-based parent indices (-1 for roots).

parents_from_edges	<i>Orient a graph into a rooted spanning forest — one parent per node.</i>
--------------------	--

Description

`minimum_spanning_tree` picks *which* edges survive; this picks which way they point, which is what turns a bag of undirected edges into something you can walk, root, or write out as SWC. Cycles are fine — each component contributes a spanning tree of itself, so this doubles as a cycle-breaker.

Usage

```
parents_from_edges(
  edges,
  n_nodes,
  weights = NULL,
  roots = NULL,
  precision = 32
)
```

Arguments

edges	Integer or numeric (E, 2) matrix of edges (0-based node indices). Direction is ignored.
n_nodes	Integer; total number of nodes. Nodes named by no edge are isolated roots.
weights	Optional numeric vector, one length per edge. NULL gives the breadth-first tree; weights give the shortest-path tree, which is a different (and generally deeper) spanning tree. Neither is the minimum spanning tree — for that, run <code>minimum_spanning_tree</code> first and orient the edges it keeps.
roots	Optional integer vector of 0-based nodes to root at; NULL roots each component at its lowest node index. Components holding none of roots fall back to that, so the result is always a complete forest. Two roots in the <i>same</i> component split it into two trees, each node going to whichever root is nearer.
precision	Integer; 32 or 64, the width the search accumulates at. Nothing here returns a distance, so this changes only which answer close ties resolve to.

Details

One search covers the whole graph. The obvious construction — a shortest-path tree per component — costs $O(\text{components} * n_{\text{nodes}})$ in output alone, which on a mesh that shatters into thousands of specks is gigabytes to answer a question whose answer is one vector.

Value

List with parents (integer 0-based parent index per node, -1 at a root) and order (integer 0-based node indices in the order they settled — a node always follows its parent, so relabelling by it guarantees parents get lower indices than their children, which is exactly the SWC requirement).

paths_to_root	<i>The path from each source up to its root.</i>
---------------	--

Description

The other direction of the same walk as descendants.

Usage

```
paths_to_root(parents, sources)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
sources	Integer vector of 0-based node indices to walk up from.

Value

List of integer vectors, one per source in sources order, ordered source-first / root-last. A source that is itself a root gives a single-element vector; an out-of-range source gives an empty one.

prune_twigs	<i>Prune twigs below given threshold.</i>
-------------	---

Description

Returns indices of nodes to keep.

Usage

```
prune_twigs(parents, threshold, weights)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
threshold	Numeric length threshold; twigs shorter than this are pruned.
weights	Optional numeric vector of edge weights; NULL counts edges.

Value

Integer vector of node indices to keep.

reroot	<i>Re-orient a forest so that each of new_roots becomes its component's root.</i>
--------	---

Description

Only the edges between each new root and its component's old root are reversed; components nobody names come back byte-identical. The general form of reroot_rewire, which takes one preferred root plus a set of new edges.

Usage

```
reroot(parents, new_roots)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
new_roots	Integer vector of 0-based node indices to root at. Two roots in the same component: the last one named wins.

Value

Integer vector of new 0-based parent indices, aligned with parents. Roots are -1.

reroot_rewire	<i>Regenerate a parent vector after adding a set of undirected edges.</i>
---------------	---

Description

Turns an edited edge set back into a valid rooted tree: the undirected adjacency is built from the original child -> parent edges plus the new from/to edges, then oriented away from root by breadth-first search.

Usage

```
reroot_rewire(parents, from, to, root)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
from, to	Integer vectors of 0-based node indices giving the undirected edges to add, e.g. the from/to returned by <code>stitch_fragments()</code> .
root	Integer 0-based index of the preferred root; its whole component is rooted there. Use a negative value to auto-pick (lowest index per component).

Value

Integer vector of new 0-based parent indices (roots are -1). Any component not reachable from root is rooted at its lowest-index node, so the result is valid even when the skeleton could not be fully healed.

resample_skeleton	<i>Place nodes at a fixed spacing along every neurite.</i>
-------------------	--

Description

The inverse problem to `downsample_skeleton()`: rather than thinning what is there, this re-samples each segment from scratch, so a skeleton whose node density varies tenfold between neurites comes out evenly sampled throughout. Each segment is divided into `round(length / spacing)` equal parts (at least one), so both of its endpoints land exactly and no runt edge is left over; a segment shorter than `spacing / 2` collapses to a single straight edge.

Usage

```
resample_skeleton(parents, x, y, z, spacing, threads = NULL)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
x, y, z	Numeric vectors of node coordinates, one entry per node.
spacing	Numeric; target distance between adjacent nodes.
threads	Optional integer; number of threads. NULL uses all cores.

Value

List with parents (0-based parent index per output node, -1 for roots), x, y, z (their coordinates), source_from and source_to (the 0-based *input* node indices of the edge each output node sits on, child then parent) and alpha (how far along that edge it lies, from the child end). The input's roots, branch points and leafs come first, in input order and unmoved; they carry their own index in both source_ columns and an alpha of 0, so `attr[source_from + 1] * (1 - alpha) + attr[source_to + 1] * alpha` interpolates any per-node quantity over the whole output. node_map points the other way: one entry per *input* node, giving the 0-based index of the output node nearest it along the neurite (ties going towards the root). Use source_/alpha to carry a per-node column forward and node_map to re-home whatever is *attached* to a node.

set_num_threads	<i>Set the number of threads used for parallel work in this session.</i>
-----------------	--

Description

By default `nat.fastcore` uses every core it can see, which is the right answer for a single call and the wrong one when the *caller* is already spreading work over processes (`parallel::mclapply()`, `future::plan(multisession)`, a cluster job): each worker would claim every core, and the resulting oversubscription can make the whole thing slower than running it on one core. Nothing tells a worker process that it is one of twenty, so it has to be told.

Usage

```
set_num_threads(n)
```

Arguments

n	Integer; number of threads. Must be ≥ 1 .
---	--

Details

Call this once, before any other `nat.fastcore` function. The pool is built at most once per session, by whichever comes first: an earlier `set_num_threads()`, the `RAYON_NUM_THREADS` environment variable, a call to `get_num_threads()`, or simply the first parallel call. Calling it again with the same `n` is a no-op; calling it with a different `n` is an error, as the pool cannot be resized.

Value

NULL, invisibly. Called for its side effect.

simplify_mesh	<i>Simplify a triangle mesh, tracking where every vertex went.</i>
---------------	--

Description

Iteratively contracts the edge whose collapse costs least under the Garland-Heckbert quadric error. Unlike other implementations of this algorithm it also reports, for every vertex of the original mesh, which vertex of the simplified mesh it ended up in - so per-vertex data survives the simplification.

Usage

```
simplify_mesh(
    faces,
    vertices,
    ratio = NULL,
    n_faces = NULL,
    aggressiveness = 7,
    preserve_border = FALSE,
    lock = NULL
)
```

Arguments

<code>faces</code>	Integer or numeric (F , 3) matrix of triangle vertex indices (0-based).
<code>vertices</code>	Numeric (V , 3) matrix of vertex coordinates. Must be finite.
<code>ratio</code>	Numeric fraction of the faces to keep, in $(0, 1]$.
<code>n_faces</code>	Integer number of faces to keep. Give exactly one of <code>ratio</code> or <code>n_faces</code> .
<code>aggressiveness</code>	Numeric exponent of the error-threshold sweep. Higher reaches the target in fewer, coarser passes. Default 7.
<code>preserve_border</code>	Logical; freeze every vertex on a mesh boundary.
<code>lock</code>	Optional logical vector, one entry per vertex. A locked vertex is never merged into another and never moved, so it keeps its exact coordinates.

Details

Non-manifold input is fine: no manifoldness is assumed or checked, and each collapse guard skips what it cannot handle rather than failing.

Value

List with `vertices` (numeric (V' , 3) matrix), `faces` (integer (F' , 3) matrix, 0-based) and `vertex_map` (integer, one entry per *original* vertex giving its 0-based index in `vertices`, or -1 if it did not survive).

Being *merged* is not a -1: a collapsed vertex carries the index of whatever it merged into, which is the point of the map. An entry is -1 exactly when the vertex it ended up in is referenced by no surviving face, which takes one of four forms: it was in no face to begin with; it was only ever in zero-area faces, which are dropped on the way in and so reduce to the first case; the whole piece it belonged to was consumed, since nothing is reserved per connected component and a small fragment goes once the target is tight enough; or the input was degenerate throughout and the output mesh is empty. Mask with `vertex_map >= 0` before aggregating.

simplify_mesh_lossless

Simplify a triangle mesh without changing its shape.

Description

Collapses only edges whose quadric error is below epsilon and repeats until a whole pass changes nothing. There is no face budget: this sheds over-tessellation

- coplanar fans, duplicate vertices, degenerate faces - rather than hitting a target. Use [simplify_mesh\(\)](#) for that.

Usage

```
simplify_mesh_lossless(
    faces,
    vertices,
    epsilon = 0.001,
    max_iterations = 9999,
    preserve_border = FALSE,
    lock = NULL
)
```

Arguments

faces	Integer or numeric (F, 3) matrix of triangle vertex indices (0-based).
vertices	Numeric (V, 3) matrix of vertex coordinates. Must be finite.
epsilon	Numeric quadric error below which an edge may collapse. An <i>absolute</i> error with units of squared distance, so it scales with your coordinates. Default 1e-3.
max_iterations	Integer cap on the number of passes. Default 9999.
preserve_border	Logical; freeze every vertex on a mesh boundary.
lock	Optional logical vector, one entry per vertex; as simplify_mesh() .

Details

Note that "lossless" is a claim about the surface, not the outline: a quadric measures distance to the planes of the incident faces, and the plane of a flat patch says nothing about where that patch ends. Pass `preserve_border = TRUE` on open meshes.

Value

List with vertices, faces and vertex_map, as [simplify_mesh\(\)](#), including when an entry of vertex_map is -1. The "whole piece consumed" case arrives by a different route here: there is no face budget, but epsilon is an *absolute* error, so a component small enough that all of its edges fall under it collapses away entirely.

simplify_rdp

Drop the nodes that do not bend a neurite, by Ramer-Douglas-Peucker.

Description

Where `downsample_skeleton()` thins by counting, this thins by *shape*: a node survives only if removing it would move the traced path by more than `epsilon`. Long straight stretches collapse to their two ends while a tight curve keeps every node it needs, so the same tolerance buys a much better skeleton per node than a fixed factor does.

Usage

```
simplify_rdp(
  parents,
  x,
  y,
  z,
  epsilon,
  preserve = NULL,
  weights = NULL,
  threads = NULL
)
```

Arguments

<code>parents</code>	Integer vector of 0-based parent indices (roots are < 0).
<code>x, y, z</code>	Numeric vectors of node coordinates, one entry per node.
<code>epsilon</code>	Numeric; how far the simplified path may stray from the original, in the units of the coordinates. 0 still drops nodes that are <i>exactly</i> collinear, and nothing else.
<code>preserve</code>	Optional logical vector, one entry per node, marking extra nodes that must survive; NULL for none.
<code>weights</code>	Optional numeric vector of child-to-parent edge weights; NULL returns no weights.
<code>threads</code>	Optional integer; number of threads. NULL uses all cores.

Details

Each replacement edge carries the length of the chain it stands in for, so geodesic distances stay right even where the geometry has been cut across.

Value

List with `nodes`, `parents`, `weights` and `node_map`, as `downsample_skeleton()`.

simplify_skeleton	<i>Keep only roots, leafs and branch points, preserving cable length.</i>
-------------------	---

Description

Each replacement edge carries the summed length of the chain it stands in for, so total cable length is unchanged.

Usage

```
simplify_skeleton(parents, weights = NULL)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
weights	Optional numeric vector of child-to-parent edge weights; NULL returns no weights.

Value

List with nodes (0-based indices of the surviving nodes, in their original relative order), parents (their new 0-based parent indices, -1 for roots, indexing *into* nodes), weights (length of each node's edge to its new parent, roots 0; NULL exactly when weights was NULL) and node_map (one entry per *input* node, giving the 0-based index *into* nodes of the survivor its data belongs to now: itself if it survived, otherwise whichever end of its chain is nearer, measured in weights, ties going towards the root).

simplify_vw	<i>Drop the nodes that contribute least area, by Visvalingam-Whyatt.</i>
-------------	--

Description

The other geometry-aware thinning. Where `simplify_rdp()` asks how far the path *moves*, this asks how much area each node adds to it and repeatedly removes whichever node adds least. The difference shows under aggressive simplification: RDP will happily keep one spike and flatten everything around it, while Visvalingam-Whyatt sheds detail evenly and so keeps a neurite looking like itself.

Usage

```
simplify_vw(
  parents,
  x,
  y,
  z,
  min_area,
```

```

    preserve = NULL,
    weights = NULL,
    threads = NULL
)

```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
x, y, z	Numeric vectors of node coordinates, one entry per node.
min_area	Numeric; remove a node while the triangle it forms with its two surviving neighbours is smaller than this, in the <i>squared</i> units of the coordinates. 0 or less is a no-op.
preserve	Optional logical vector, one entry per node, marking extra nodes that must survive; NULL for none.
weights	Optional numeric vector of child-to-parent edge weights; NULL returns no weights.
threads	Optional integer; number of threads. NULL uses all cores.

Value

List with nodes, parents, weights and node_map, as `downsample_skeleton()`.

smat_auto_limit	<i>The limit_dist="auto" value for a scoring matrix.</i>
-----------------	--

Description

The `limit_dist="auto"` value for a scoring matrix.

Usage

```
smat_auto_limit(smat_values, dist_edges, dot_edges, use_alpha)
```

Arguments

smat_values	Numeric scoring matrix, or NULL for the built-in FCWB matrix.
dist_edges	Numeric vector of distance bin edges for smat_values.
dot_edges	Numeric vector of dot-product bin edges for smat_values.
use_alpha	Logical; when falling back to the built-in matrix, use the alpha-weighted variant.

Value

Numeric `limit_dist` value implied by the scoring matrix.

smooth_mesh

*Smooth a triangle mesh***Description**

Moves vertices and touches nothing else: the face matrix, the vertex count and the vertex order all come back unchanged, so anything indexed by vertex - synapses, radii, labels - is still attached to the vertex it was attached to.

Usage

```
smooth_mesh(
  faces,
  vertices,
  method = c("taubin", "laplacian", "humphrey"),
  iterations = 10L,
  lambda = NULL,
  mu = NULL,
  alpha = NULL,
  beta = NULL,
  weights = c("uniform", "inverse_distance", "cotangent"),
  preserve_border = FALSE,
  lock = NULL,
  volume_correction = FALSE,
  threads = NULL
)
```

Arguments

faces	Integer or numeric (F, 3) matrix of triangle vertex indices (0-based).
vertices	Numeric (V, 3) matrix of vertex coordinates. Must be finite.
method	One of "taubin", "laplacian" or "humphrey"; see Details.
iterations	Integer number of passes. For "taubin" one pass is a full lambda-then-mu pair, i.e. two sweeps over the mesh - not one, as <code>trimesh.smoothing.filter_taubin</code> counts them. Counting half-steps lets an odd iterations end on a shrink that nothing undoes. Default 10.
lambda	Numeric diffusion speed for "laplacian" and "taubin", in $[0, 1]$. Larger is more aggressive. Default 0.5.
mu	Numeric inflating pass for "taubin". Must be negative and larger in magnitude than lambda. Default -0.53.
alpha	Numeric, "humphrey" only: how hard vertices are pulled back towards their original positions, in $[0, 1]$. Default 0.1.
beta	Numeric, "humphrey" only: how much of that pull-back lands on the vertex itself rather than on its one-ring, in $[0, 1]$. Default 0.5.

weights	One of "uniform", "inverse_distance" or "cotangent". "uniform" counts every neighbour equally and also regularises the <i>sampling</i> , which means it slides vertices along the surface where the tessellation is uneven. "cotangent" is the discrete Laplace-Beltrami operator: it depends on the shape rather than on the triangulation, so it moves vertices along the normal and leaves them alone within the surface. That is usually what you want on meshes out of EM segmentation, whose triangles vary wildly in size and aspect.
preserve_border	Logical; pin every vertex on a mesh boundary - an endpoint of an edge used by exactly one face. Without this an open mesh's rim rolls inwards under any of these filters, because a boundary vertex's one-ring lies entirely to one side of it.
lock	Optional logical vector, one entry per vertex. A locked vertex comes back at bit-wise the same coordinates but still pulls on its neighbours, which is what makes it a boundary condition rather than a hole. Unioned with preserve_border, not an alternative to it. Same name and same meaning as <code>simplify_mesh()</code> 's lock.
volume_correction	Logical; see the section above.
threads	Integer cap on the thread count for this call, or NULL for the process-wide pool.

Details

Three methods:

"taubin" (**the default**) Alternating shrink and inflate passes, tuned so the two cancel below a cut-off frequency. Removes noise without removing the shape, and is the default for that reason.

"laplacian" The plain diffusion step: simple, effective, and it *shrinks*. At $\lambda = 0.5$ and five iterations - what `navis.smooth_mesh` ships - a neuron mesh loses 88% of its enclosed volume. Reach for it when the mesh is a means to an end rather than when its volume means something, or pair it with `volume_correction`.

"humphrey" The HC filter of Vollmer et al., which fights shrinkage by pulling each vertex back towards where it started rather than towards a lower frequency. The gentler of the two on fine detail worth keeping.

Value

Numeric (V, 3) matrix of new vertex coordinates, in the input's order.

The volume correction

`volume_correction = TRUE` rescales the result **about its centroid** so the enclosed volume matches the input's. About the centroid is the one place this deliberately differs from `trimesh.smoothing.filter_laplacian`, which is what `navis.smooth_mesh` calls: upstream rescales about the *origin*, which is not a shape operation. On the 722817260 test neuron at `navis`' own defaults it displaces the mesh by 41 μm , and the mesh is 19-26 μm across; it is also not translation invariant, so the same mesh smoothed at two different offsets comes out two different shapes, and far enough from the origin the volume ratio goes negative and the cube root returns NaN.

The correction runs once, at the end, which is not an approximation of running it every iteration but exactly equal to it: every filter here is an affine combination of a vertex and a normalised average of its neighbours, and those commute with a uniform scaling.

On a closed mesh the correction is exactly what it says. A mesh that is *not* closed still usually gets one, and deliberately: both measurements cone every face back to the same anchor, so their ratio stays a consistent measure of how much the surface shrank even where neither number is an enclosed volume on its own. That matters because meshes worth smoothing are almost never watertight. What is left is the genuinely undecidable case - the ratio is zero, infinite, NaN or negative, a flat sheet being the clean example - where the vertices come back smoothed but unscaled and a warning says so.

See Also

[simplify_mesh\(\)](#), the other half of mesh cleanup.

Examples

```
# A 5x5 grid with its middle vertex lifted out of the plane.
n <- 5
faces <- do.call(rbind, unlist(lapply(0:(n - 2), function(i) {
  lapply(0:(n - 2), function(j) {
    rbind(
      c(i * n + j, (i + 1) * n + j, (i + 1) * n + j + 1),
      c(i * n + j, (i + 1) * n + j + 1, i * n + j + 1)
    )
  })
}), recursive = FALSE))
vertices <- cbind(rep(0:(n - 1), each = n), rep(0:(n - 1), n), 0)
storage.mode(vertices) <- "double"
vertices[13, 3] <- 1

# One full-strength Laplacian pass puts it back in the plane its ring spans.
out <- smooth_mesh(faces, vertices, method = "laplacian", lambda = 1, iterations = 1)
out[13, 3]

# Pin the rim so an open mesh does not roll inwards.
out <- smooth_mesh(faces, vertices, preserve_border = TRUE)
identical(out[1, ], vertices[1, ])
```

smooth_skeleton

Smooth a skeleton with a moving average along each neurite.

Description

Takes the tracing jitter out of a skeleton without touching its topology or its node count: every node keeps its identity and its parent, and only its coordinates move. Roots, branch points and leafs are pinned — a branch point that drifted would drag three neurites apart — so this is safe to run before measuring angles, tortuosity or tangent vectors, all of which a raw traced skeleton overstates.

Usage

```
smooth_skeleton(parents, x, y, z, window = 5, values = NULL, threads = NULL)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
x, y, z	Numeric vectors of node coordinates, one entry per node.
window	Integer; nodes in the window, counting the node itself. Even values round down to the odd value below, since the window is symmetric. 0 and 1 are no-ops.
values	Optional numeric vector or (N, K) matrix: a per-node field to smooth <i>instead of</i> the coordinates.
threads	Optional integer; number of threads. NULL uses all cores.

Details

Pass values to smooth some other per-node field instead — a width, say. The window is a count of *nodes*, so nothing here reads a geometric meaning into what it is handed and a width smooths by the same code as an x; x, y and z are then only checked for length, not read.

Value

With values = NULL, a list with x, y and z: the new coordinates, in the input's node order. With values given, the new values in the shape they came in — a numeric vector for a vector, an (N, K) matrix for a matrix.

```
smooth_skeleton_gaussian
```

Smooth a skeleton with a Gaussian kernel along each neurite.

Description

The same operation as smooth_skeleton() with a softer, scale-based kernel: sigma is a distance in the units of the coordinates rather than a count of nodes, so the amount of smoothing does not change when the skeleton is resampled. The kernel measures distance *along* the neurite rather than between the points, which would otherwise let the far arm of a hairpin pull on the near one. Segment ends are pinned by reflecting the neurite about them.

Usage

```
smooth_skeleton_gaussian(
  parents,
  x,
  y,
  z,
  sigma,
  truncate = 4,
```

```

    values = NULL,
    threads = NULL
)

```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
x, y, z	Numeric vectors of node coordinates, one entry per node. Read only to measure distance along each neurite when values is given.
sigma	Numeric; kernel width, as a distance along the neurite.
truncate	Numeric; how many sigma out to keep summing. 4 covers all but 1e-4 of the kernel's mass.
values	Optional numeric vector or (N, K) matrix: a per-node field to smooth <i>instead of</i> the coordinates. Stack the coordinates into it to smooth both in one pass — the kernel is measured over the untouched input geometry either way.
threads	Optional integer; number of threads. NULL uses all cores.

Details

Pass values to smooth some other per-node field instead — a width, say. Unlike `smooth_skeleton()`, that field cannot simply replace x, y and z: this kernel's weights come from distance *along the neurite*, so it goes on measuring over the coordinates while smoothing values. Handing it a width as though it were geometry would make "distance" the cumulative absolute change in width — a plausible-looking number and a meaningless kernel — so the two stay separate arguments.

Value

With values = NULL, a list with x, y and z: the new coordinates, in the input's node order. With values given, the new values in the shape they came in — a numeric vector for a vector, an (N, K) matrix for a matrix.

stitch_fragments	<i>Find the minimal-length edges that reconnect the fragments of a skeleton.</i>
------------------	--

Description

Given a per-node component label and node coordinates, this returns the set of new edges that would join the fragments into a single tree while minimising the total added length (a minimum spanning tree over the fragments). It does *not* modify the skeleton — see `heal_skeleton` for the one-shot version.

Usage

```
stitch_fragments(components, x, y, z, w, mask, max_dist, threads = NULL)
```

Arguments

components	Integer vector giving each node's connected component, e.g. the output of <code>connected_components()</code> . Only equality of labels matters.
x, y, z	Numeric vectors of node coordinates, one entry per node.
w	Optional numeric vector giving a 4th coordinate, one entry per node. The search then happens in 4D, so nodes with similar w look closer together; pass a (scaled) radius here to prefer bridging fragments of similar calibre. Note that <code>max_dist</code> is then measured in 4D too. NULL searches in plain 3D.
mask	Optional logical vector marking the nodes that may be used as endpoints for a new edge; NULL allows every node. A fragment without a single eligible node cannot be connected.
max_dist	Optional numeric upper bound on the length of any single new edge; NULL means no limit. Fragments whose closest eligible nodes are farther apart than this are left disconnected.
threads	Optional integer; number of threads. NULL uses all cores. Worth capping when stitching many skeletons across several processes — see <code>set_num_threads()</code> , which is cheaper than passing this on every call.

Value

List with `from` and `to` (integer vectors of 0-based node indices, one pair per new edge) and `dist` (numeric edge lengths). At most $(\text{\#fragments} - 1)$ edges.

strahler_index	<i>Calculate Strahler Index.</i>
----------------	----------------------------------

Description

Calculate Strahler Index.

Usage

```
strahler_index(parents, greedy, to_ignore, min_twig_size)
```

Arguments

parents	Integer vector of 0-based parent indices (roots are < 0).
greedy	Logical; use the greedy variant of the algorithm.
to_ignore	Optional integer vector of node indices to skip.
min_twig_size	Optional integer; ignore twigs shorter than this.

Value

Integer vector with the Strahler index of each node.

subtree_height	<i>Height of the subtree below each node.</i>
----------------	---

Description

A node’s height is the geodesic distance from it down to the farthest leaf below it; leaves have a height of 0 and a root carries the length of the longest root-to-leaf path in its component.

Usage

subtree_height(parents, weights)

Arguments

- parents Integer vector of 0-based parent indices (roots are < 0).
- weights Optional numeric vector of child-to-parent edge weights; NULL counts edges (hop distance).

Value

Numeric vector with the height of each node.

symmetrize	<i>Symmetrise a score matrix</i>
------------	----------------------------------

Description

Combines cell (i, j) with (j, i), since NBLAST is not symmetric. The idiomatic (m + t(m)) / 2 builds two further n x n matrices on the way to the answer; this builds one, the result.

Usage

symmetrize(scores, symmetry = c("mean", "min", "max", "none"), n_cores = NULL)

Arguments

- scores Square (n, n) numeric score matrix. The diagonal is not read, so self-scores may be left on it.
- symmetry How to combine (i, j) with (j, i): "mean" (the default), "min", "max", or "none" to mirror the upper triangle onto the lower instead of combining.
- n_cores Optional integer thread count; NULL uses all cores.

Details

You do not need this before `nblast_hclust()` or `nblast_dist()` – both take a `symmetry` argument and fold the same combine into their own fused pass. Reach for it when something *else* has to read the matrix: plotting it, writing it out, or handing it to a function that assumes symmetry.

Value

A symmetric (n, n) matrix, with `dimnames` carried over. `scores` itself is unchanged – R's value semantics forbid writing to it, which is the one thing this cannot do that the Python version can.

See Also

`nblast_dist()` and `nblast_hclust()`, which symmetrise as part of a single pass over the matrix.

Examples

```
m <- matrix(c(1, 0.4, 0.8, 1), 2, 2)
symmetrize(m)
```

`synapse_flow_centrality`

Synapse flow centrality for each node.

Description

`presynapses`/`postsynapses` give the number of pre-/post-synapses at each node. `mode` is one of "centrifugal", "centripetal" or "sum".

Usage

```
synapse_flow_centrality(parents, presynapses, postsynapses, mode)
```

Arguments

<code>parents</code>	Integer vector of 0-based parent indices (roots are < 0).
<code>presynapses</code>	Integer vector: number of presynapses at each node.
<code>postsynapses</code>	Integer vector: number of postsynapses at each node.
<code>mode</code>	Character; one of "centrifugal", "centripetal" or "sum".

Value

Integer vector with the synapse flow centrality of each node.

synblast

*Forward syNBLAST of every query neuron against every target neuron.***Description**

Returns an (n_query, n_target) score matrix.

Usage

```
synblast(
  q_points,
  q_types,
  t_points,
  t_types,
  smat_values,
  dist_edges,
  dot_edges,
  normalize,
  n_cores,
  precision,
  progress
)
```

Arguments

q_points	List of (N, 3) numeric matrices of query connector coordinates.
q_types	List of integer vectors of query per-connector type ids.
t_points	List of (N, 3) numeric matrices of target connector coordinates.
t_types	List of integer vectors of target per-connector type ids.
smat_values	Numeric scoring matrix, or NULL for the built-in FCWB matrix.
dist_edges	Numeric vector of distance bin edges for smat_values.
dot_edges	Numeric vector of dot-product bin edges for smat_values.
normalize	Logical; normalise each score by the query self-match score.
n_cores	Optional integer thread count; NULL or <= 0 uses all cores.
precision	Integer; compute in 32- or 64-bit floats.
progress	Logical; display a progress bar.

Value

Numeric (n_query, n_target) score matrix.

synblast_allbyall *All-by-all forward syNBLAST over synapse clouds.*

Description

points are lists of (N, 3) connector coordinate matrices and types the matching per-connector integer type ids. Returns an (n, n) score matrix.

Usage

```
synblast_allbyall(
    points,
    types,
    smat_values,
    dist_edges,
    dot_edges,
    normalize,
    n_cores,
    precision,
    progress
)
```

Arguments

points	List of (N, 3) numeric matrices of connector coordinates, one per neuron.
types	List of integer vectors of per-connector type ids, aligned with points.
smat_values	Numeric scoring matrix, or NULL for the built-in FCWB matrix.
dist_edges	Numeric vector of distance bin edges for smat_values.
dot_edges	Numeric vector of dot-product bin edges for smat_values.
normalize	Logical; normalise each score by the query self-match score.
n_cores	Optional integer thread count; NULL or <= 0 uses all cores.
precision	Integer; compute in 32- or 64-bit floats.
progress	Logical; display a progress bar.

Value

Numeric (n, n) score matrix; cell (i, j) is query i vs target j.

tps_affine	<i>The affine part of a thin-plate spline</i>
------------	---

Description

A thin-plate spline decomposes into an affine part and a non-affine ("bending") part. This returns the affine, which is what the spline converges to far from the landmarks.

Usage

```
tps_affine(reg)
```

Arguments

reg A tps_transform, from [tps_transform\(\)](#).

Value

A (4, 4) homogeneous transformation matrix, with last row c(0, 0, 0, 1).

See Also

[tps_transform\(\)](#), [mls_affine\(\)](#)

Examples

```
# landmarks related by a pure affine give back exactly that affine
src <- rbind(c(0, 0, 0), c(1, 0, 0), c(0, 1, 0), c(0, 0, 1), c(1, 1, 1))
trg <- src * 2 + matrix(c(3, 4, 5), nrow(src), 3, byrow = TRUE)
tps_affine(tps_transform(src, trg))
```

tps_coefs	<i>Coefficients of a fitted thin-plate spline</i>
-----------	---

Description

Returns the fit itself: W, the weights of the non-affine part, and A, the coefficients of the affine part. Together with the source landmarks these fully determine the transform, so they are what you serialise if you want to store a fit rather than repeat it.

Usage

```
tps_coefs(reg)
```

Arguments

reg A tps_transform, from `tps_transform()`.

Value

A list with elements source ((M, 3)), W ((M, 3)) and A ((4, 3), whose first row is the translation).

See Also

`tps_transform()`

Examples

```
src <- rbind(c(0, 0, 0), c(10, 10, 10), c(100, 100, 100), c(80, 10, 30))
trg <- rbind(c(1, 15, 5), c(9, 18, 21), c(80, 99, 120), c(5, 10, 80))
co <- tps_coefs(tps_transform(src, trg))
dim(co$W)
dim(co$A)
```

tps_transform	<i>Fit a thin-plate spline to landmark pairs</i>
---------------	--

Description

A thin-plate spline interpolates the source landmarks onto the target landmarks exactly, and between them minimises the integral bending norm – the smoothest warp consistent with the landmarks. This is the usual fallback when no image registration exists for a pair of template spaces, and what nat's tpsreg provides.

Usage

```
tps_transform(source, target)
```

Arguments

source	An (M, 3) matrix of source landmarks. A data frame is also accepted – if it has x/y/z columns (in any case) those are used, whatever else it carries, otherwise the first three columns.
target	An (M, 3) matrix of target landmarks, one per source landmark.

Details

The fit happens once, here, and the resulting object can then be applied as often as you like with `tps_xform()`. That split matters: the fit is *cubic* in the landmark count while applying it is linear, so refitting per call would dominate any real workload.

There is no batch_size to tune, unlike the navis implementation this mirrors. The (n_points, n_landmarks) distance matrix is fused into the accumulation rather than built, so peak memory is the output matrix regardless of how many points or landmarks are involved.

Value

An object of class `tps_transform`.

Inverting

A thin-plate spline has no closed-form inverse. To map the other way, fit the other way round – `tps_transform(target, source)` – which is a fresh fit, not an inversion.

See Also

`tps_xform()`, `mls_transform()`

Examples

```
src <- rbind(c(0, 0, 0), c(10, 10, 10), c(100, 100, 100), c(80, 10, 30))
trg <- rbind(c(1, 15, 5), c(9, 18, 21), c(80, 99, 120), c(5, 10, 80))
tps <- tps_transform(src, trg)
tps

# landmarks are reproduced exactly
stopifnot(all.equal(tps_xform(tps, src), trg))

tps_xform(tps, rbind(c(0, 0, 0), c(50, 50, 50)))
```

tps_xform	<i>Apply a thin-plate spline to points</i>
-----------	--

Description

Apply a thin-plate spline to points

Usage

```
tps_xform(reg, xyz, n_cores = NULL)
```

Arguments

reg	A <code>tps_transform</code> , from <code>tps_transform()</code> .
xyz	An (N, 3) matrix of coordinates. A data frame is also accepted – if it has x/y/z columns (in any case) those are used, whatever else it carries, otherwise the first three columns – as is a bare length-3 vector. <code>nat::xyzmatrix()</code> gives you this for a neuron.
n_cores	Number of threads. NULL (default) uses all cores.

Value

An (N, 3) matrix of transformed coordinates.

See Also

`tps_transform()`, `tps_affine()`

Examples

```
src <- rbind(c(0, 0, 0), c(10, 10, 10), c(100, 100, 100), c(80, 10, 30))
trg <- rbind(c(1, 15, 5), c(9, 18, 21), c(80, 99, 120), c(5, 10, 80))
tps <- tps_transform(src, trg)
tps_xform(tps, c(50, 50, 50))

## Not run:
# transform a neuron
n <- nat::Cell07PNs[[1]]
nat::xyzmatrix(n) <- tps_xform(tps, nat::xyzmatrix(n))

## End(Not run)
```

trace_loops

Walk directed half-edges into closed rings.

Description

Greedy: at a non-manifold boundary vertex several half-edges leave at once, so this takes whichever is still free. Every half-edge lands in exactly one ring, which is what makes this cover the whole boundary — a cycle basis quietly drops the edges that are not part of a simple cycle, and those holes stay open.

Usage

```
trace_loops(halfedges)
```

Arguments

halfedges	Integer or numeric (K, 2) matrix of directed half-edges, as returned by <code>boundary_halfedges</code> or <code>exposed_halfedges</code> .
-----------	---

Details

Every ring comes back simple — no vertex twice. A greedy walk does not give that on its own: at a pinch, where several boundary edges meet at one point, it can leave and re-enter the same vertex, and what it traced is then a figure of eight rather than a polygon, which no triangulator downstream is defined on. So a walk is cut where it crosses itself and the pieces are handed on separately.

A walk that runs into a dead end abandons what is still in hand, and so is a ring of fewer than three vertices, so the rings need not account for every half-edge handed in. Cycles already split off from an abandoned walk are kept.

Value

List with `rings` (integer vector: every ring's vertices, end to end) and `offsets` (integer vector, one longer than there are rings). Ring `i` is `rings[(offsets[i] + 1):offsets[i + 1]]` — the offsets are 0-based bounds, as the vertex indices are.

<code>triangulate_rings</code>	<i>Triangulate boundary rings, wound against the direction they run in.</i>
--------------------------------	---

Description

Three attempts, in order: ear-clipping the ring flattened through its area-weighted (Newell) normal, then through its best-fit plane, and failing both, ear-clipping it in three dimensions without flattening at all. All three are always closed and always correctly wound.

Usage

```
triangulate_rings(rings, offsets, vertices, threads = NULL)
```

Arguments

<code>rings</code>	Integer vector of ring vertices, end to end (<code>trace_loops</code> ' rings).
<code>offsets</code>	Integer vector of 0-based ring bounds (<code>trace_loops</code> ' offsets). Must be non-decreasing and run from 0 to <code>length(rings)</code> .
<code>vertices</code>	Numeric (<code>V</code> , 3) matrix of vertex coordinates.
<code>threads</code>	Optional integer; number of threads. <code>NULL</code> uses all cores.

Details

A ring only gets past the first attempt if the flattening self-intersects. That is not the same as the ring being un-planar — a gently curved ring can cast a crossed shadow and a folded one need not — and since the first two attempts are both projections, they tend to fail together. The third has no plane to be defeated by.

The cap winds *against* its ring, because the ring runs the way the faces it still has wind it: a cap that agreed would have the two disagreeing about which side is out.

Value

Integer (`M`, 3) matrix of new faces (0-based vertex indices), ring by ring. A ring of `k` vertices always caps to `k - 2` triangles.

unique_edges	<i>Unique undirected edges of a triangle mesh.</i>
--------------	--

Description

Each face (a, b, c) contributes the edges (a, b), (b, c), (c, a). Edges are undirected, so each pair is normalised to [min, max] before dedup; self-loops from degenerate faces are kept.

Usage

```
unique_edges(faces, vertices = NULL, threads = NULL)
```

Arguments

- faces Integer or numeric (F, 3) matrix of triangle vertex indices (0-based).
- vertices Optional numeric (V, 3) matrix of vertex coordinates. If given, also returns the euclidean length of each unique edge.
- threads Optional integer; number of threads. NULL uses all cores.

Value

List with edges (integer (n_unique, 2) matrix of [min, max] rows, sorted ascending by (max, min)) and lengths (numeric edge lengths, or NULL when vertices was NULL).

Index

adjacency, 4
all_dists_to_root, 4

betweenness, 5
boundary_halfedges, 5
break_segments, 6
bridges, 6

child_to_parent_dists, 7
classify_nodes, 8
cmtk_affine (cmtk_properties), 8
cmtk_dims (cmtk_properties), 8
cmtk_domain (cmtk_properties), 8
cmtk_properties, 8
cmtk_read, 9
cmtk_read(), 8, 10–13
cmtk_spacing (cmtk_properties), 8
cmtk_versions (cmtk_properties), 8
cmtk_xform, 10
cmtk_xform(), 9, 10, 13
cmtk_xform_inv, 12
cmtk_xform_inv(), 9–11
connected_components, 14
connected_components_graph, 14
connected_components_graph(), 44
contract_nodes, 15
contract_vertices, 15

descendant_counts, 16
descendants, 16
dist_to_root, 17
downsample_skeleton, 17

elastix_affine (elastix_properties), 19
elastix_grid_origin
 (elastix_properties), 19
elastix_grid_size (elastix_properties),
 19
elastix_grid_spacing
 (elastix_properties), 19

elastix_kinds (elastix_properties), 19
elastix_probe_invertible, 18
elastix_properties, 19
elastix_read, 20
elastix_read(), 18, 19, 21, 22
elastix_xform, 21
elastix_xform(), 20, 23
elastix_xform_inv, 22
elastix_xform_inv(), 18, 20–22
exposed_halfedges, 24

fast_hclust, 24
fast_hclust(), 52

generate_segments, 25
geodesic_clusters, 26
geodesic_distances, 27
geodesic_farthest, 28
geodesic_farthest_mesh, 28
geodesic_matrix_graph, 29
geodesic_matrix_mesh, 30
geodesic_matrix_mesh(), 43
geodesic_mst_graph, 32
geodesic_mst_mesh, 33
geodesic_nearest, 34
geodesic_nearest_mesh, 34
geodesic_pairs, 35
geodesic_path, 36
geodesic_predecessors, 37
get_num_threads, 38

has_cycles, 38
heal_skeleton, 39

leaf_order, 40
level_set_components, 41
longest_path, 41
longest_paths, 42

mesh_connected_components, 43
minimum_spanning_tree, 44

mls_affine, 45
mls_affine(), 48, 77
mls_transform, 46
mls_transform(), 45–48, 79
mls_xform, 47
mls_xform(), 46, 47

nblast, 48
nblast(), 53
nblast_allbyall, 49
nblast_allbyall(), 51, 52, 55
nblast_dist, 50
nblast_dist(), 52, 74
nblast_hclust, 51
nblast_hclust(), 25, 51, 55, 74
nblast_knn, 53
nblast_pairs, 55
node_indices, 56

parents_from_edges, 57
paths_to_root, 58
plot(), 52
prune_twigs, 58

reroot, 59
reroot_rewire, 59
resample_skeleton, 60

set_num_threads, 61
simplify_mesh, 61
simplify_mesh(), 63, 68, 69
simplify_mesh_lossless, 63
simplify_rdp, 64
simplify_skeleton, 65
simplify_vw, 65
smat_auto_limit, 66
smooth_mesh, 67
smooth_skeleton, 69
smooth_skeleton_gaussian, 70
stats::as.dendrogram(), 52
stats::cutree(), 52
stats::dist(), 50
stats::hclust(), 24, 25, 40, 51, 52
stitch_fragments, 71
strahler_index, 72
subtree_height, 73
symmetrize, 73
synapse_flow_centrality, 74
synblast, 75
synblast_allbyall, 76

tps_affine, 77
tps_affine(), 46, 80
tps_coefs, 77
tps_transform, 78
tps_transform(), 46, 47, 77–80
tps_xform, 79
tps_xform(), 78, 79
trace_loops, 80
triangulate_rings, 81

unique_edges, 82
unique_edges(), 44