Package 'Battlefield'

November 11, 2026

Type Package

Title Swiss-army toolkit for selecting niche fronts and invasive margins in spatial transcriptomics data

Version 1.0.0

Description Battlefield is a Swiss-army toolkit originally developed to define and extract spatial spots from specific tissue regions—such as front regions, niche borders, invasive margins, and cluster interfaces—using spatial transcriptomics data or clustered tissue maps. It has since been extended to support trajectory selection and layer inspection, and now provides a collection of low-level utilities for spatial transcriptomics analysis. These utilities are primarily intended to be reused within higher-level analytical packages. It is designed to work with sequencing-based platforms such as Visium at several resolutions and Visium HD(binned).

URL https://github.com/ZheFrench/BattleField, https://zhefrench.github.io/Battlefield/

BugReports https://github.com/ZheFrench/BattleField/issues

License CeCILL | file LICENSE

Encoding UTF-8

LazyData FALSE

Depends R (>= 4.6)

biocViews Sequencing, Software, Transcriptomics, Spatial

Imports stats, RANN, dplyr, SummarizedExperiment, methods

Suggests BiocStyle, knitr, markdown, rmarkdown, SpatialExperiment, SpatialExperimentIO, VisiumIO, ggplot2, pheatmap, pals, OSTA.data, tidyr, STexampleData, testthat (>= 3.0.0), codetools, grid, tools

Config/testthat/edition 3

RoxygenNote 7.3.3

VignetteBuilder knitr

git_url https://git.bioconductor.org/packages/Battlefield

git_branch RELEASE_3_23

git_last_commit a0574ac

git_last_commit_date 2026-04-28

Repository Bioconductor 3.23

Date/Publication 2026-11-11

Author Jean-Philippe Villemin [aut, cre] (ORCID:
<https://orcid.org/0000-0002-1838-5880>),
European Research Council [fnd] (ERC-2022)

Maintainer Jean-Philippe Villemin <jpvillemin@gmail.com>

Contents

Battlefield-package . . . 3
add_borders_to_spe . . . 5
add_layers_to_spe . . . 6
add_trajectories_to_spe . . . 7
adjacent_endpoint . . . 8
bresenham_line . . . 9
build_all_borders . . . 10
build_all_cores . . . 11
build_one_line . . . 13
build_one_trajectory . . . 14
build_similar_trajectories . . . 15
closest_spot . . . 17
compute_centroids . . . 18
count_all_inlaids . . . 19
count_all_neighborhoods . . . 20
count_inlaid . . . 22
count_neighborhood . . . 23
create_all_layers . . . 24
create_cluster_layers . . . 26
detect_grid_type . . . 27
directed_cluster_interface_pairs . . . 29
estimate_spot_spacing . . . 30
filter_out_by_endpoint_clusters . . . 31
get_inlaid_spots . . . 32
get_neighborhood_params . . . 33
get_neighborhood_spots . . . 35
point_segment_distance_vec . . . 37
remove_used_points . . . 38
select_border_spots . . . 39
select_core_spots . . . 41
shift_point . . . 43
unit_normal_left . . . 43
visiumHD_16um_simulated_spe . . . 44
visiumHD_8um_simulated_spe . . . 45
visium_simulated_spe . . . 46

Index **48**

`Battlefield-package` *Battlefield: Swiss-army toolkit for spatial transcriptomics region selection*

Description

Battlefield provides low-level, modular utilities to define and extract spatial spots from specific regions in spatial transcriptomics data. It supports four core workflows: identifying cluster interfaces, characterizing intra-cluster layers, modeling inter-cluster trajectories, and analyzing neighborhood composition.

Key Features

Cluster Interfaces Identify and analyze boundary regions where distinct tissue or cell populations interact. See `select_border_spots`, `build_all_borders`, `directed_cluster_interface_pairs`.

Intra-cluster Layers Characterize spatial layers or gradients within a single cluster (border, intermediate, core). See `create_cluster_layers`, `create_all_layers`.

Inter-cluster Trajectories Model spatial transitions across multiple clusters via centroid-to-centroid Bresenham paths. See `build_one_trajectory`, `build_similar_trajectories`.

Neighborhood Composition Analyze k-NN cluster composition with optional distance constraints. See `count_neighborhood`, `get_neighborhood_spots`.

Core Workflow

1. Start with a `SpatialExperiment` object containing spot coordinates and cluster annotations. 2. Extract spot metadata via `colData(spe)` to create a working `data.frame`. 3. Apply selection functions (e.g., `select_border_spots`, `create_cluster_layers`) to identify regions of interest. 4. Integrate results back to spe using helper functions: `add_borders_to_spe`, `add_layers_to_spe`, `add_trajectories_to_spe`. 5. Use annotated spe for downstream analyses (e.g., ligand–receptor interactions via **BulkSignalR**).

Key Algorithms

- **Bresenham line rasterization** (`bresenham_line`): Traces discrete paths on a spot grid.
- **k-NN neighborhood detection** (via **RANN**): Identifies spatial neighbors and cluster composition.
- **Grid type inference** (`detect_grid_type`): Distinguishes square vs. hexagonal layouts.
- **Layer classification**: Uses radial distance quantiles to stratify clusters into border/intermediate/core regions.
- **Centroid-based trajectories**: Connects cluster centroids and selects nearby spots for trajectory analysis.

Important Data Shapes

All selection functions work with `data.frames` containing:

Input columns (required) `x`, `y` (numeric coordinates), `cluster` (any grouping type), `spot_id` (must match `colnames(spe)` for SPE integration).

Output columns (added by selection functions) `interface`, `directed_pair` (e.g., "3-4"), `is_border`, `is_core` (logical flags), `layer` ("border"/"intermediate"/"core"). Keep these column names stable across pipelines.

Example Datasets

Three simulated `SpatialExperiment` objects are included for testing:

- `visium_simulated_spe`: Square grid (Visium-like).
- `visiumHD_8um_simulated_spe`: Hexagonal grid (VisiumHD 8 µm binned).
- `visiumHD_16um_simulated_spe`: Hexagonal grid (VisiumHD 16 µm binned).

Load with: `data("visium_simulated_spe", package = "Battlefield")`

Links & References

- Main vignette: `vignettes/Battlefield-Main.Rmd` — comprehensive examples for all four core workflows.
- GitHub: https://github.com/ZheFrench/Battlefield

Author(s)

Maintainer: Jean-Philippe Villemin <jpvillemin@gmail.com> (ORCID)

Other contributors:

- European Research Council (ERC-2022) [funder]

See Also

Useful links:

- https://github.com/ZheFrench/BattleField
- https://zhefrench.github.io/Battlefield/
- Report bugs at https://github.com/ZheFrench/BattleField/issues

add_borders_to_spe *Add border or core spot selections to SpatialExperiment colData*

Description

This function takes spot selection results (either border or core spots) and adds them as annotations in the colData of a SpatialExperiment object.

Usage

```
add_borders_to_spe(spe, border = NULL, core = NULL, erase = FALSE)
```

Arguments

spe	A SpatialExperiment object from which the selection dataframes were derived.
border	Optional data.frame of border spots. Can be output from either [select_border_spots()] or [build_all_borders()]. Expected columns: spot_id, interface, mode.
core	Optional data.frame of core spots. Can be output from either [select_core_spots()] or [build_all_cores()]. Expected columns: spot_id, interface, mode.
erase	Logical. If 'TRUE', erase pre-existing battlefield columns before adding new ones. If 'FALSE' (default), issue a warning if columns already exist and skip adding them.

Details

At least 'border' must be provided. 'core' is optional. If 'core' is provided without 'border', an error is raised.

The following unified columns are added: - 'is_border': logical, TRUE if spot is a border spot, FALSE or NA otherwise - 'is_core': logical, TRUE if spot is a core spot, FALSE or NA otherwise - 'interface': the target interface cluster - 'border_mode': character, "inner", "outer", or NA (from the 'mode' column in input data)

Value

The SpatialExperiment object with updated colData.

Examples

```
data("visiumHD_16um_simulated_spe", package = "Battlefield")
spe <- visiumHD_16um_simulated_spe
df <- data.frame(
  spot_id = colnames(spe),
  x = SpatialExperiment::spatialCoords(spe)[, 1],
  y = SpatialExperiment::spatialCoords(spe)[, 2],
  cluster = SummarizedExperiment::colData(spe)$cluster
)
# Using build_all_borders and build_all_cores
```

```
all_borders <- build_all_borders(df, k = 6)
all_cores <- build_all_cores(df, all_borders, mode = "inner")

# Or using individual select functions
border_3_to_4 <- select_border_spots(df, cluster = 3, interface = 4, k = 6)
core_3_to_4 <- select_core_spots(df, all_borders, cluster = 3, interface = 4)
```

`add_layers_to_spe` *Add layer classifications to SpatialExperiment colData*

Description

This function takes layer classification results (border, intermediate, core) and adds them as annotations in the colData of a SpatialExperiment object.

Usage

```
add_layers_to_spe(spe, layer = NULL, erase = FALSE)
```

Arguments

`spe`	A SpatialExperiment object from which the layer dataframes were derived.
`layer`	A data.frame of layer classifications. Can be output from either [create_cluster_layers()] or [create_all_layers()]. Expected columns: spot_id, layer.
`erase`	Logical. If 'TRUE', erase pre-existing layer columns before adding new ones. If 'FALSE' (default), issue a warning if columns already exist and skip adding them.

Details

The 'layer' parameter must be provided and contain at least one row.

The following column is added: - 'layer': character, one of "border", "intermediate", "core", or NA

Value

The SpatialExperiment object with updated colData.

Examples

```
data("visiumHD_16um_simulated_spe", package = "Battlefield")
spe <- visiumHD_16um_simulated_spe
df <- data.frame(
  spot_id = colnames(spe),
  x = SpatialExperiment::spatialCoords(spe)[, 1],
  y = SpatialExperiment::spatialCoords(spe)[, 2],
  cluster = SummarizedExperiment::colData(spe)$cluster
)
```

```
# Or for a single cluster
cluster_1_layers <- create_cluster_layers(df, target_cluster = 1, k = 6)
spe <- add_layers_to_spe(spe, layer = cluster_1_layers)
```

`add_trajectories_to_spe` *Add trajectory information to SpatialExperiment colData*

Description

This function takes trajectory results (e.g., from [build_similar_trajectories()]) and adds trajectory metadata to the colData of a SpatialExperiment object.

Usage

```
add_trajectories_to_spe(spe, trajectory = NULL, erase = FALSE)
```

Arguments

`spe`	A SpatialExperiment object from which the trajectory dataframes were derived.
`trajectory`	A data.frame of trajectory spots. Expected output from [build_similar_trajectories()]. Expected columns: spot_id, trajectory_id, offset, pos_on_seg, dist_to_seg.
`erase`	Logical. If 'TRUE', erase pre-existing trajectory columns before adding new ones. If 'FALSE' (default), issue a warning if columns already exist and skip adding them.

Details

The 'trajectories' parameter must be provided and contain at least one row.

The following columns are added: - 'trajectory_id': character, identifier for each trajectory (e.g., "main", "left_1", "right_2") - 'offset': numeric, the offset distance used to build each trajectory line - 'pos_on_seg': numeric in [0, 1], the position along the trajectory - 'dist_to_seg': numeric, the distance to the original segment

Value

The SpatialExperiment object with updated colData.

Examples

```
data("visiumHD_16um_simulated_spe", package = "Battlefield")
spe <- visiumHD_16um_simulated_spe
df <- data.frame(
  spot_id = colnames(spe),
  x = SpatialExperiment::spatialCoords(spe)[, 1],
  y = SpatialExperiment::spatialCoords(spe)[, 2],
  cluster = SummarizedExperiment::colData(spe)$cluster
```

```
)
centroids <- compute_centroids(df)
A <- centroids[centroids$cluster == 1, c("x", "y")]
B <- centroids[centroids$cluster == 3, c("x", "y")]
trajs <- build_similar_trajectories(df, A, B, top_n = 10, n_extra = 1, side =
"both")
spe <- add_trajectories_to_spe(spe, trajectory = trajs)
```

`adjacent_endpoint` *Pick a point adjacent to a selected endpoint, on a given side of a segment*

Description

Given an endpoint (typically the start or end of a previously selected path), this function computes a target point located one perpendicular step away from the endpoint, on the "left" or "right" side relative to the directed segment $A \rightarrow B$. It then returns the closest spot to that target among the remaining candidates ‘df_rest‘.

Usage

```
adjacent_endpoint(df_rest, endpoint, A, B, spacing, side = c("left", "right"))
```

Arguments

`df_rest`	A data frame of candidate spots containing at least columns ‘x‘ and ‘y‘.
`endpoint`	A one-row data frame with columns ‘x‘ and ‘y‘ representing the endpoint from which to step sideways. If it has multiple rows, only the first row is used.
`A`	A data frame with columns ‘x‘ and ‘y‘ representing point A defining the direction of the segment. Only the first row is used.
`B`	A data frame with columns ‘x‘ and ‘y‘ representing point B defining the direction of the segment. Only the first row is used.
`spacing`	Numeric scalar; step size (in the same coordinate units as ‘x‘/‘y‘) used to move perpendicularly from ‘endpoint‘.
`side`	Character; which side to step to relative to the vector $A \rightarrow B$. Must be ‘"left"‘ or ‘"right"‘.

Details

The unit direction vector is computed from $A \rightarrow B$ and a unit left normal $(-vy, vx)/||v||$ is derived. The target point is:

$$T = endpoint + sign \cdot spacing \cdot n$$

where ‘sign = +1‘ for ‘"left"‘ and ‘-1‘ for ‘"right"‘.

The closest spot is selected with `closest_spot()` using squared Euclidean distance.

Value

A one-row data frame (same columns as ‘df_rest‘) corresponding to the spot closest to the computed perpendicular target point.

Examples

```
df <- data.frame(x = c(0, 1, 1, 2, 2), y = c(0, 0, 1, 0, 1), id = 1:5)
A <- data.frame(x = 0, y = 0)
B <- data.frame(x = 2, y = 0)
endpoint <- data.frame(x = 1, y = 0)

# Step "left" of A->B (here, toward positive y)
adjacent_endpoint(df, endpoint, A, B, spacing = 1, side = "left")

# Step "right" of A->B (here, toward negative y)
adjacent_endpoint(df, endpoint, A, B, spacing = 1, side = "right")
```

---

bresenham_line *Rasterize a line between two points using Bresenham's algorithm*

Description

Generates integer grid coordinates along the line segment connecting two points using Bresenham's line algorithm.

Usage

```
bresenham_line(p0, p1, snap = TRUE)
```

Arguments

p0	A ‘data.frame‘ with at least columns ‘x‘ and ‘y‘. If it contains multiple rows, only the first row is used.
p1	A ‘data.frame‘ with at least columns ‘x‘ and ‘y‘. If it contains multiple rows, only the first row is used.
snap	Logical; if ‘TRUE‘ (default), ‘p0‘ and ‘p1‘ coordinates are rounded to the nearest integer before running the algorithm. If ‘FALSE‘, coordinates are truncated via ‘as.integer()‘.

Details

The function:

- Validates inputs are data frames with ‘x‘/‘y‘ columns and at least one row.
- Converts endpoints to integer coordinates (optionally rounding first).
- Applies the classic Bresenham algorithm to enumerate grid points.

This is useful for tracing discrete paths on an integer lattice (e.g., pixels, tile grids, spatial transcriptomics spot grids).

Value

A ‘data.frame‘ with columns ‘x‘ and ‘y‘ giving the integer grid points visited by the line, including both endpoints, in traversal order from ‘p0‘ to ‘p1‘.

Examples

```
p0 <- data.frame(x = 1.2, y = 2.7)
p1 <- data.frame(x = 7.9, y = 5.1)

# Default (snap = TRUE): rounds endpoints first
bresenham_line(p0, p1)

# No rounding (snap = FALSE): integer coercion truncates toward zero
bresenham_line(p0, p1, snap = FALSE)
```

---

build_all_borders *Build border spots for all oriented cluster pairs*

Description

This function iterates over all **ordered** cluster pairs (A -> B) and returns a single data.frame containing the border spots for each interface, as computed by a border-detection function (e.g. [select_border_spots()]).

Usage

```
build_all_borders(
  df,
  k = 6,
  max_dist = NULL,
  mode = "both",
  pairs = NULL,
  coord_cols = c("x", "y"),
  cluster_col = "cluster"
)
```

Arguments

df	A data.frame containing at least coordinate columns and a cluster label column.
k	Integer. Number of nearest neighbors to consider (excluding self) when computing borders. Default is 6.
max_dist	Optional numeric. Maximum Euclidean distance for neighbors to be considered. If ‘NULL‘, no distance filtering is applied.
mode	Character. One of "inner", "outer", or "both". Controls which border direction(s) to select for each pair. Default is "both".

pairs	Optional data.frame of oriented cluster pairs, typically produced by [directed_cluster_interface_pairs()]. Must contain ‘cluster‘ and ‘interface‘ columns. If ‘NULL‘, it is computed from ‘df[[cluster_col]]‘.
coord_cols	Character vector of length 2 giving the coordinate column names. Default is ‘c("x","y")‘.
cluster_col	Character. Name of the column containing cluster labels. Default is ‘"cluster"‘.

Details

If ‘pairs_df‘ is not provided, it is generated from the cluster labels using [directed_cluster_interface_pairs()].

Value

A data.frame produced by row-binding the result of border selection for each oriented pair. Typically contains the original columns of ‘df‘ plus interface annotation columns from the border selector.

Note on mode column: The ‘mode‘ column in the returned data.frame will always contain either "inner" or "outer", never "both". Even when the ‘mode‘ parameter is set to "both", each border spot row is labeled with its actual direction (cluster→interface is "inner", interface→cluster is "outer").

Examples

```
# Example with synthetic data
set.seed(1)
df_ex <- data.frame(
  x = rnorm(200),
  y = rnorm(200),
  cluster = sample(c("A","B","C"), 200, replace = TRUE)
)
all_borders <- build_all_borders(df_ex, k = 6)
head(all_borders)
```

---

build_all_cores *Build core spots for all oriented cluster pairs*

Description

This function iterates over all **ordered** cluster pairs (A -> B) and returns a single data.frame containing the core (control) spots for each pair, as computed by [select_core_spots()].

Usage

```
build_all_cores(
  df,
  border_df,
  mode = "both",
```

```
  pairs = NULL,
  coord_cols = c("x", "y"),
  cluster_col = "cluster"
)
```

Arguments

df	A data.frame containing at least coordinate columns and a cluster label column.
border_df	A data.frame of border spots from [build_all_borders()].
mode	Character. One of "inner", "outer", or "both". Controls which mode values from border_df to use when counting border spots. Default is "both".
pairs	Optional data.frame of oriented cluster pairs, typically produced by [directed_cluster_interface_pairs()]. Must contain ‘cluster‘ and ‘interface‘ columns. If ‘NULL‘, it is computed from ‘df[[cluster_col]]‘.
coord_cols	Character vector of length 2 giving the coordinate column names. Default is ‘c("x","y")‘.
cluster_col	Character. Name of the column containing cluster labels. Default is ‘"cluster"‘.

Details

If ‘pairs‘ is not provided, it is generated from the cluster labels using [directed_cluster_interface_pairs()].

Value

A data.frame produced by row-binding the result of core spot selection for each oriented pair. Typically contains the original columns of ‘df‘ plus annotation columns (is_core, interface, mode) from the core spot selector.

Note on mode column: The ‘mode‘ column in the returned data.frame reflects which border modes were used for each core spot selection: "inner", "outer", or "both". This differs from ‘build_all_borders()‘ which only returns "inner" or "outer".

Examples

```
# Example with synthetic data
set.seed(1)
df_ex <- data.frame(
  spot_id = paste0("spot_", seq_len(200)),
  x = rnorm(200),
  y = rnorm(200),
  cluster = sample(c("A","B","C"), 200, replace = TRUE)
)
all_borders <- build_all_borders(df_ex, k = 6)
all_cores <- build_all_cores(df_ex, all_borders, mode = "both")
head(all_cores)
```

`build_one_line` *Build a single line of spots along a segment*

Description

Selects spots near the segment defined by endpoints ‘A‘ and ‘B‘ using ‘build_one_trajectory()‘, then orders the selected spots by their projection parameter (from ‘A‘ to ‘B‘). The result is returned as a data frame.

Usage

```
build_one_line(df_rest, A, B, top_n = 19, max_dist = NULL)
```

Arguments

`df_rest`	A data frame of candidate spots containing at least columns ‘x‘ and ‘y‘.
`A`	A data frame with columns ‘x‘ and ‘y‘ defining endpoint A of the segment. If it contains multiple rows, only the first row is used.
`B`	A data frame with columns ‘x‘ and ‘y‘ defining endpoint B of the segment. If it contains multiple rows, only the first row is used.
`top_n`	Integer; number of closest spots to keep (default ‘19‘). If ‘NULL‘, no top-N truncation is applied.
`max_dist`	Numeric; if not ‘NULL‘, only spots with distance to the segment ‘<= max_dist‘ are kept.

Details

This function is a thin wrapper around ‘build_one_trajectory()‘ that returns only the ordered ‘selected‘ data frame (and not the segment metadata).

It uses the base R pipe ‘|>‘ and ‘dplyr::arrange()‘.

Value

A data frame of selected spots (subset of ‘df_rest‘) ordered along the segment (in increasing ‘pos_on_seg‘). The output includes the extra columns added by ‘build_one_trajectory()‘ (typically ‘dist_to_seg‘ and ‘pos_on_seg‘).

Examples

```
df <- data.frame(
  x = c(0, 1, 2, 3, 4, 2),
  y = c(0, 0, 0, 0, 0, 1),
  id = 1:6
)
A <- data.frame(x = 0, y = 0)
B <- data.frame(x = 4, y = 0)
```

```
build_one_line(df, A, B, top_n = 5)
```

`build_one_trajectory` *Select spots near a segment and order them along the segment*

Description

Computes the distance from each spot in ‘df‘ to the segment ‘[p0, p1]‘, optionally filters by a maximum distance, keeps the ‘top_n‘ closest spots, and finally orders the retained spots by their projection position ‘pos_on_seg‘ along the segment (from ‘p0‘ to ‘p1‘). This is the foundation function for building trajectory lines across spatial spots.

Usage

```
build_one_trajectory(df, p0, p1, top_n = 100, max_dist = NULL)
```

Arguments

`df`	A data frame of spots containing at least columns ‘x‘ and ‘y‘. Additional columns are preserved in the output.
`p0`	A data frame with columns ‘x‘ and ‘y‘ defining the first endpoint of the segment. If it has multiple rows, only the first row is used.
`p1`	A data frame with columns ‘x‘ and ‘y‘ defining the second endpoint of the segment. If it has multiple rows, only the first row is used.
`top_n`	Integer; number of closest spots to keep (default ‘100‘). If ‘NULL‘, no top-N truncation is applied.
`max_dist`	Numeric; if not ‘NULL‘, only spots with distance to the segment ‘<= max_dist‘ are kept.

Details

Distances are computed using `point_segment_distance_vec()`. Ordering uses the projection parameter $pos_o n_s eg = ((P - A) \cdot (B - A))/||B - A||^2$ clamped to ‘[0, 1]‘.

This function uses `dplyr` verbs (‘mutate‘, ‘filter‘, ‘arrange‘, ‘slice_head‘) via the ‘|>‘ pipe.

Value

A data frame containing the selected spots, with two extra columns:

dist_to_seg Euclidean distance from the spot to the segment.

pos_on_seg Clamped projection parameter in ‘[0, 1]‘ indicating position along the segment (0 at ‘p0‘, 1 at ‘p1‘).

The rows are ordered by ‘pos_on_seg‘ (i.e., from ‘p0‘ to ‘p1‘).

Examples

```
# Minimal example with base data.frame inputs
df <- data.frame(
  x = c(0, 1, 2, 3, 4),
  y = c(0, 1, 1, 2, 4),
  id = letters[1:5]
)
p0 <- data.frame(x = 0, y = 0)
p1 <- data.frame(x = 4, y = 0)

# Keep 3 closest spots (no distance threshold)
res <- build_one_trajectory(df, p0, p1, top_n = 3)
head(res)
```

`build_similar_trajectories` *Build parallel spot lines around a central segment*

Description

Constructs a central line of spots near the segment defined by endpoints ‘A‘ and ‘B‘, then builds additional parallel lines on the left, right, or both sides by translating the segment along its left unit normal. After each line is built, its spots are removed from the remaining pool to avoid reuse.

Usage

```
build_similar_trajectories(
  df,
  A,
  B,
  top_n = 19,
  n_extra = 2,
  side = c("left", "right", "both"),
  lane_width_factor = 1.15,
  max_dist = NULL
)
```

Arguments

`df`	A data frame of candidate spots containing at least columns ‘x‘ and ‘y‘. Additional columns are preserved.
`A`	A data frame with columns ‘x‘ and ‘y‘ defining endpoint A of the central segment. If it contains multiple rows, only the first row is used.
`B`	A data frame with columns ‘x‘ and ‘y‘ defining endpoint B of the central segment. If it contains multiple rows, only the first row is used.
`top_n`	Integer; number of closest spots to keep per line (default ‘19‘). If ‘NULL‘, no top-N truncation is applied in the underlying selection.

`n_extra` Integer; number of additional lines to build on each requested side (default ‘2‘). For example, ‘n_extra = 2‘ with ‘side = "both"‘ yields 1 center line + 2 left + 2 right.

`side` Character; which side(s) to build relative to the vector $A \to B$. One of ‘"left"‘, ‘"right"‘, or ‘"both"‘.

`lane_width_factor` Numeric scalar; multiplier applied to the estimated spot spacing to obtain the inter-line offset (default ‘1.15‘).

`max_dist` Numeric; if not ‘NULL‘, only spots with distance to each (translated) segment ‘<= max_dist‘ are considered when building each line.

Details

The inter-line distance is computed as:

$$w = lane_width_factor \cdot spacing$$

where ‘spacing‘ is the median nearest-neighbor distance estimated from ‘df‘.

Lines are generated by shifting both endpoints ‘A‘ and ‘B‘ by ‘offset * n‘, where ‘n‘ is the left unit normal of $A \to B$. Positive offsets correspond to the left side; negative offsets correspond to the right side.

This function relies on helpers such as ‘estimate_spot_spacing()‘, ‘unit_normal_left()‘, ‘shift_point()‘, ‘build_one_line()‘, and ‘remove_used_points()‘.

The implementation uses the base R pipe ‘|>‘ and ‘dplyr::mutate()‘ / ‘dplyr::bind_rows()‘. Ensure ‘dplyr‘ is installed.

Value

A data frame containing all selected spots from all lines, stacked together, with additional columns ‘trajectory_id‘ (e.g., ‘"main"‘, ‘"left_1"‘, ‘"right_1"‘) and ‘offset‘ (signed offset used for that line).

Examples

```
df <- data.frame(
  x = rep(1:10, each = 3),
  y = rep(1:3, times = 10),
  id = seq_len(30)
)
A <- data.frame(x = 1, y = 2)
B <- data.frame(x = 10, y = 2)

out <- build_similar_trajectories(df, A, B, top_n = 5, n_extra = 1, side =
"both")
head(out)
unique(out$trajectory_id)
```

`closest_spot` *Find the closest spot to a target point*

Description

Returns the row of ‘df‘ corresponding to the spot nearest to the target coordinates ‘(tx, ty)‘ using squared Euclidean distance.

Usage

```
closest_spot(df, tx, ty)
```

Arguments

`df`	A data frame containing at least numeric columns ‘x‘ and ‘y‘ representing spot coordinates.
`tx`	Numeric scalar; target x-coordinate.
`ty`	Numeric scalar; target y-coordinate.

Details

The function minimizes ‘(x - tx)^2 + (y - ty)^2‘. Squared distances are used for efficiency; taking the square root is unnecessary for argmin.

If multiple spots are tied for the minimum distance, the first occurrence is returned (as per ‘which.min()‘).

Value

A one-row data frame (same columns as ‘df‘) corresponding to the closest spot. The result is always a data frame (‘drop = FALSE‘).

Examples

```
df <- data.frame(x = c(0, 2, 5), y = c(0, 2, 1), id = c("a", "b", "c"))
closest_spot(df, tx = 1.9, ty = 2.1)
```

`compute_centroids` *Compute cluster centroids (mean x/y per cluster)*

Description

Computes the centroid of each cluster by taking the mean of the ‘x‘ and ‘y‘ coordinates for each ‘cluster‘ level.

Usage

```
compute_centroids(df)
```

Arguments

`df`	A ‘data.frame‘ containing at least the columns ‘cluster‘, ‘x‘, and ‘y‘. ‘cluster‘ can be any type accepted by ‘aggregate()‘ grouping (e.g., integer, factor, character). ‘x‘ and ‘y‘ must be numeric.

Details

This is a thin wrapper around `aggregate` using `FUN = mean`. Missing values are handled according to ‘mean()‘’s default behavior (i.e., ‘NA‘s will propagate unless you pre-handle them).

Value

A ‘data.frame‘ with one row per cluster and columns:

cluster Cluster identifier.

x Mean x-coordinate for the cluster.

y Mean y-coordinate for the cluster.

Examples

```
df <- data.frame(
  cluster = c(1, 1, 2, 2),
  x = c(0, 2, 10, 12),
  y = c(1, 3, 5, 7)
)
compute_centroids(df)
```

`count_all_inlaids` *Count inlaid composition for all clusters*

Description

This function counts inlaid/annotation composition within each cluster in a single call. It returns a dataframe with counts of inlaid types for each source cluster.

Usage

```
count_all_inlaids(
  df,
  clusters = NULL,
  inlaid_col = "cluster",
  cluster_col = "cluster"
)
```

Arguments

`df`	A data.frame containing at least the columns ‘x‘, ‘y‘, cluster identifier column, and ‘spot_id‘. - ‘x‘, ‘y‘: numeric coordinates of spots - cluster column (name specified by ‘cluster_col‘): cluster assignment for #’ each spot - inlaid column (name specified by ‘inlaid_col‘): inlaid/annotation for each spot - ‘spot_id‘: unique identifier for each spot
`clusters`	Optional vector of cluster labels to process. If ‘NULL‘, all unique clusters in ‘df‘ are used.
`inlaid_col`	Character. Name of the column containing inlaid/annotation values. Default is ‘"cluster"‘.
`cluster_col`	Character. Name of the column containing cluster assignments. Default is ‘"cluster"‘.

Value

A data.frame with columns:

cluster The source cluster being analyzed.

inlaid The inlaid/annotation type.

count Number of spots with this inlaid type in the cluster.

proportion Proportion of this inlaid type among all spots in cluster.

Rows are grouped by source cluster and sorted by count within each group (descending).

Examples

```
data("visium_simulated_spe", package = "Battlefield")
spe <- visium_simulated_spe
df <- data.frame(
  spot_id = colnames(spe),
  x = SpatialExperiment::spatialCoords(spe)[, 1],
  y = SpatialExperiment::spatialCoords(spe)[, 2],
  cluster = SummarizedExperiment::colData(spe)$cluster,
inlaid = sample(paste0("type_", 1:3), length(colnames(spe)), replace = TRUE)
)
# Get inlaid statistics for all clusters
all_inlaids <- count_all_inlaids(df, inlaid_col = "inlaid")
head(all_inlaids)
```

`count_all_neighborhoods` *Count neighborhood composition for all clusters*

Description

This function computes neighborhood statistics for all clusters in a dataset in a single call. It returns a dataframe with counts of neighboring cluster types for each source cluster.

Usage

```
count_all_neighborhoods(
  df,
  clusters = NULL,
  k = 100,
  max_dist = NULL,
  inlaid_col = NULL,
  coords = c("x", "y"),
  cluster_col = "cluster"
)
```

Arguments

`df`	A data.frame containing at least the columns ‘x‘, ‘y‘, cluster identifier column, and ‘spot_id‘. - ‘x‘, ‘y‘: numeric coordinates of spots - cluster column (name specified by ‘cluster_col‘): cluster assignment for each spot - ‘spot_id‘: unique identifier for each spot (optional, required for detailed analysis)
`clusters`	Optional vector of cluster labels to process. If ‘NULL‘, all unique clusters in ‘df‘ are used.
`k`	Integer. Number of nearest neighbors to consider. Default is 100. Larger values capture more distant neighbors.

max_dist	Numeric. Maximum distance from the target to consider. If ‘NULL‘ (default), all neighbors up to k are included without distance filtering.
inlaid_col	Character. Name of the column containing inlaid/annotation values to count in neighborhoods. If ‘NULL‘ (default), uses the ‘cluster_col‘ value. This allows counting neighborhood composition by any column (e.g., cell type, tissue type).
coords	Character vector of length 2 giving the coordinate column names. Default is ‘c("x", "y")‘.
cluster_col	Character. Name of the column containing cluster assignments. Default is ‘"cluster"‘.

Details

This is a batch processing function that applies [count_neighborhood()] to all clusters and combines the results into a single dataframe. Each row represents a neighbor cluster found around a source cluster, with counts and proportions.

Value

A data.frame with columns:

cluster The source cluster being analyzed.

neighborhood The neighbor cluster type or inlaid value.

count Number of neighbor spots of this cluster type.

proportion Proportion of this cluster among all neighbors of the cluster.

Rows are grouped by source cluster and sorted by count within each group (descending).

Examples

```
data("visium_simulated_spe", package = "Battlefield")
spe <- visium_simulated_spe
df <- data.frame(
  spot_id = colnames(spe),
  x = SpatialExperiment::spatialCoords(spe)[, 1],
  y = SpatialExperiment::spatialCoords(spe)[, 2],
  cluster = SummarizedExperiment::colData(spe)$cluster
)
# Get neighborhood statistics for all clusters
all_neighbors <- count_all_neighborhoods(df, k = 50)
head(all_neighbors)
```

`count_inlaid` *Count inlaid composition within a source cluster*

Description

This function counts the composition of an inlaid/annotation column within a target source cluster. Unlike [count_neighborhood()], this counts types *inside* the source cluster itself, not around it.

Usage

```
count_inlaid(df, cluster, inlaid_col = "cluster", cluster_col = "cluster")
```

Arguments

`df`	A data.frame containing at least the columns ‘x‘, ‘y‘, cluster identifier column, and ‘spot_id‘. - ‘x‘, ‘y‘: numeric coordinates of spots - cluster column (name specified by ‘cluster_col‘): cluster assignment for each spot - inlaid column (name specified by ‘inlaid_col‘): inlaid/annotation for each spot - ‘spot_id‘: unique identifier for each spot
`cluster`	The cluster label to analyze inlaid composition for.
`inlaid_col`	Character. Name of the column containing inlaid/annotation values. Default is ‘"cluster"‘ (to analyze cluster composition within another grouping).
`cluster_col`	Character. Name of the column containing cluster assignments. Default is ‘"cluster"‘.

Value

A data.frame with columns:

inlaid Inlaid/annotation value.

count Number of spots with this inlaid type in the cluster.

proportion Proportion of this inlaid type among all spots in cluster.

Rows are sorted by count in descending order.

Examples

```
data("visium_simulated_spe", package = "Battlefield")
spe <- visium_simulated_spe
df <- data.frame(
  spot_id = colnames(spe),
  x = SpatialExperiment::spatialCoords(spe)[, 1],
  y = SpatialExperiment::spatialCoords(spe)[, 2],
  cluster = SummarizedExperiment::colData(spe)$cluster,
inlaid = sample(paste0("type_", 1:3), length(colnames(spe)), replace = TRUE)
)
# Count inlaid types within cluster 1
```

```
inlaid_counts <- count_inlaid(df, cluster = 1, inlaid_col = "inlaid")
inlaid_counts
```

`count_neighborhood` *Count annotated spot types in the neighborhood of a cluster or point*

Description

This function computes summary statistics of cluster types in the neighborhood of a target cluster or a specific point. It uses [get_neighborhood_spots()] internally to identify neighbors, then counts how many belong to each cluster type.

Usage

```
count_neighborhood(
  df,
  cluster = NULL,
  spot_id = NULL,
  k = 100,
  max_dist = NULL,
  inlaid_col = NULL,
  coords = c("x", "y"),
  cluster_col = "cluster"
)
```

Arguments

`df`	A data.frame containing at least the columns ‘x‘, ‘y‘, cluster identifier column, and ‘spot_id‘. - ‘x‘, ‘y‘: numeric coordinates of spots - cluster column (name specified by ‘cluster_col‘): cluster assignment for each spot - ‘spot_id‘: unique identifier for each spot (optional, required for detailed analysis)
`cluster`	The cluster label for which to analyze the neighborhood. Either ‘cluster‘ or ‘spot_id‘ must be provided, but not both.
`spot_id`	Character. The spot identifier to find neighbors for. Either ‘cluster‘ or ‘spot_id‘ must be provided, but not both.
`k`	Integer. Number of nearest neighbors to consider. Default is 100. Larger values capture more distant neighbors.
`max_dist`	Numeric. Maximum distance from the target to consider. If ‘NULL‘ (default), all neighbors up to k are included without distance #’ filtering.
`inlaid_col`	Character. Name of the column containing inlaid/annotation values to count in neighborhoods. If ‘NULL‘ (default), uses the ‘cluster_col‘ value. This allows counting neighborhood composition by any column (e.g., cell type, tissue type).
`coords`	Character vector of length 2 giving the coordinate column names. Default is ‘c("x", "y")‘.
`cluster_col`	Character. Name of the column containing cluster assignments. Default is ‘"cluster"‘.

Details

This function wraps [get_neighborhood_spots()] and summarizes the results by counting spots from each neighboring cluster or inlaid type. In cluster mode, the source cluster itself is excluded from the counts. In point mode, all neighboring clusters are counted.

The result shows the composition of the neighborhood: how many spots of each cluster type surround the target.

Value

A data.frame with columns:

neighborhood Cluster label or inlaid value (from neighboring spots).

count Number of times this cluster/type appears in the neighborhood.

proportion Proportion of this cluster/type relative to all neighbors.

Rows are sorted by count in descending order.

Examples

```
data("visium_simulated_spe", package = "Battlefield")
spe <- visium_simulated_spe
df <- data.frame(
  spot_id = colnames(spe),
  x = SpatialExperiment::spatialCoords(spe)[, 1],
  y = SpatialExperiment::spatialCoords(spe)[, 2],
  cluster = SummarizedExperiment::colData(spe)$cluster
)
# Count cluster types in neighborhood of cluster 1
neighbor_counts <- count_neighborhood(df, cluster = 1, k = 50)
neighbor_counts

# Count cluster types around a specific point
first_spot <- df$spot_id[1]
point_neighbors <- count_neighborhood(df, spot_id = first_spot, k = 10)
point_neighbors
```

---

create_all_layers *Create and visualize layers for multiple clusters*

Description

Batch create layer classifications for multiple clusters and optionally generate individual plots for each cluster.

Usage

```
create_all_layers(
  df,
  clusters = NULL,
  k = 6,
  max_dist = NULL,
  intermediate_quantile = 0.5,
  coord_cols = c("x", "y"),
  cluster_col = "cluster"
)
```

Arguments

df	A data.frame containing at least coordinate and cluster columns.
clusters	Optional vector of cluster labels to process. If 'NULL', all unique clusters are used.
k	Integer. Number of nearest neighbors to consider. Default is 6.
max_dist	Optional numeric. Maximum Euclidean distance for neighbors.
intermediate_quantile	Numeric between 0 and 1. The quantile of distances to border used to define the intermediate layer threshold. Default is 0.5 (median).
coord_cols	Character vector of length 2 giving the coordinate column names. Default is 'c("x","y")'.
cluster_col	Character. Name of the column containing cluster labels. Default is '"cluster"'.

Value

A data.frame combining all processed clusters with the 'layer' column added.

Examples

```
data("visiumHD_16um_simulated_spe", package = "Battlefield")
spe <- visiumHD_16um_simulated_spe
df <- data.frame(
  spot_id = colnames(spe),
  x = SpatialExperiment::spatialCoords(spe)[, 1],
  y = SpatialExperiment::spatialCoords(spe)[, 2],
  cluster = SummarizedExperiment::colData(spe)$cluster
)
all_layers <- create_all_layers(df, k = 6, intermediate_quantile = 0.5)
head(all_layers)
```

`create_cluster_layers` *Create layer classification for spots in a cluster*

Description

This function classifies spots within a target cluster into three layers: core, intermediate, and border. The border layer consists of spots at the cluster interface (touching other clusters). The core layer consists of spots far from any border. The intermediate layer bridges the two, with depth defined by proximity to the border.

Usage

```
create_cluster_layers(
  df,
  target_cluster,
  k = 6,
  max_dist = NULL,
  intermediate_quantile = 0.5,
  coord_cols = c("x", "y"),
  cluster_col = "cluster"
)
```

Arguments

`df`	A data.frame containing at least the coordinate columns and a cluster label column.
`target_cluster`	Cluster label for which to create layers.
`k`	Integer. Number of nearest neighbors to consider (excluding self) when identifying border spots. Default is 6.
`max_dist`	Optional numeric. Maximum Euclidean distance for neighbors. If ‘NULL‘, no distance filtering is applied.
`intermediate_quantile`	Numeric between 0 and 1. The quantile of distances to border used to define the intermediate layer threshold. Default is 0.5 (median). #’ Spots with distance to border <= this quantile threshold are classified as intermediate. Use lower values (e.g., 0.33) for narrower intermediate layers, higher values (e.g., 0.75) for wider intermediate layers.
`coord_cols`	Character vector of length 2 giving the coordinate column names. Default is ‘c("x","y")‘.
`cluster_col`	Character. Name of the column containing cluster labels. Default is ‘"cluster"‘.

Details

Layer definitions: - **Border**: Spots in the target cluster that touch at least one spot from a different cluster (kNN-based). - **Intermediate**: Non-border spots whose distance to the nearest

border spot is within the ‘intermediate_quantile‘ threshold of all non-border spots. - **Core**: All remaining non-border spots that are far from the border.

The ‘intermediate_quantile‘ parameter controls the depth of the intermediate layer: - 0.33: Narrow intermediate layer (only very close to border) - 0.50: Moderate intermediate layer (median distance threshold) - 0.75: Wide intermediate layer (most non-border spots included)

Value

A data.frame based on ‘df‘ filtered to contain only spots from ‘target_cluster‘, with an additional column: - ‘layer‘: character, one of "border", "intermediate", or "core"

Examples

```
data("visiumHD_16um_simulated_spe", package = "Battlefield")
spe <- visiumHD_16um_simulated_spe
df <- data.frame(
  spot_id = colnames(spe),
  x = SpatialExperiment::spatialCoords(spe)[, 1],
  y = SpatialExperiment::spatialCoords(spe)[, 2],
  cluster = SummarizedExperiment::colData(spe)$cluster
)
# Default: moderate depth
layers <- create_cluster_layers(df, target_cluster = 1, k = 6)
table(layers$layer)

# Narrow intermediate layer
layers_narrow <- create_cluster_layers(df, target_cluster = 1, k = 6,
intermediate_quantile = 0.33)
table(layers_narrow$layer)
```

---

detect_grid_type *Detect the grid type (square vs hexagonal) from spatial coordinates*

Description

This function inspects k-nearest-neighbor distances in a 2D coordinate set to infer whether the points lie on a **square** or **hexagonal** lattice.

Usage

```
detect_grid_type(
  df,
  coords = c("x", "y"),
  k = 12,
  tolerance = 0.1,
  verbose = TRUE
)
```

Arguments

df	A data.frame containing spatial coordinates.
coords	Character vector of length 2 giving the coordinate column names. Default is 'c("x","y")'.
k	Integer. Number of neighbors to consider (internally uses 'k + 1' to include the point itself, then removes it). Default is 12.
tolerance	Numeric. Tolerance for matching the characteristic ratio to $\sqrt{2}$ or $\sqrt{3}$. Default is 0.1.
verbose	Logical. If 'TRUE', prints diagnostic messages. Default is 'TRUE'.

Details

The method:

1. Computes the nearest-neighbor distance for each point and uses the median of those distances as an estimate of the grid step size.
2. Normalizes all neighbor distances by this step size.
3. Looks for a characteristic secondary distance peak:
   - square grid: ratio $\approx \sqrt{2}$
   - hexagonal grid: ratio $\approx \sqrt{3}$

The grid step size is estimated as the median of the per-point nearest-neighbor distance. The "median distance ratio" is computed from normalized neighbor distances restricted to the interval (1.2, 2.0), which tends to capture the second-shell distances on regular grids.

Value

A list with:

grid_type One of '"square"', '"hexagonal"', or '"unknown"'.

step Estimated grid step size (median nearest-neighbor distance).

median_ratio Median normalized distance ratio in the (1.2, 2.0) range.

Examples

```
# --- Example 1: square grid ---
sq <- expand.grid(x = 0:9, y = 0:9)
res_sq <- detect_grid_type(sq, coords = c("x","y"), k = 12, tolerance = 0.1)
res_sq$grid_type

# --- Example 2: hexagonal grid (pointy-top axial-like layout) ---
nx <- 10; ny <- 10
hex <- expand.grid(i = 0:(nx-1), j = 0:(ny-1))
hex$x <- hex$i + 0.5 * (hex$j %% 2)
hex$y <- (sqrt(3)/2) * hex$j
hex <- hex[, c("x","y")]
res_hex <- detect_grid_type(hex, coords = c("x","y"), k = 12, tolerance =
0.1)
```

```
res_hex$grid_type
```

`directed_cluster_interface_pairs` *Build all oriented cluster pairs (A -> B) and their directed_pair labels*

Description

Given a vector of cluster labels, this function returns all **ordered** (oriented) pairs of distinct clusters. For each pair (‘cluster‘, ‘interface‘), it also creates a ‘directed_pair‘ string label such as ‘"A-B"‘.

Usage

```
directed_cluster_interface_pairs(
  cluster_labels,
  interface_separator = "-",
  sort_clusters = FALSE
)
```

Arguments

`cluster_labels`	A vector of cluster labels (character, factor, numeric, etc.). ‘NA‘ values are ignored.
`interface_separator`	Character string used to join ‘cluster‘ and ‘interface‘ into the ‘directed_pair‘ label. Default is ‘"-"‘.
`sort_clusters`	Logical. If ‘TRUE‘, unique cluster labels are sorted (lexicographically) before building pairs. If ‘FALSE‘, preserves first appearance order. Default is ‘FALSE‘.

Value

A data.frame with columns:

cluster Source cluster label (character).

interface Target cluster label (character).

directed_pair Directed pair label, e.g. "A-B".

If fewer than 2 unique (non-NA) clusters are present, returns an empty data.frame with the same columns.

Examples

```
cl <- c("1","1","2","3", NA)
directed_cluster_interface_pairs(cl)
```

`estimate_spot_spacing` *Estimate spot spacing from nearest-neighbor distances*

Description

Estimates the typical spacing between spots by computing each spot's nearest-neighbor distance (excluding itself) and returning the median of those distances. For large datasets, the estimate is computed on a random subsample for speed.

Usage

```
estimate_spot_spacing(df, sample_n = 1000)
```

Arguments

`df`	A data frame containing at least columns 'x' and 'y' (numeric) representing spot coordinates.
`sample_n`	Integer; maximum number of spots to sample (default '1000'). If 'nrow(df) > sample_n', a random subset of size 'sample_n' is used; otherwise all spots are used.

Details

Nearest neighbors are computed with `RANN::nn2()` using 'k = 2'. The first neighbor is the point itself (distance 0), so the function uses the second neighbor distance 'nn$nn.dists[, 2]' as the true nearest-neighbor distance.

Missing values are handled via 'median(..., na.rm = TRUE)'.

Value

A numeric scalar: the median nearest-neighbor distance (in the same units as 'x' and 'y').

Examples

```
set.seed(1)
df <- data.frame(
  x = rep(1:5, each = 5),
  y = rep(1:5, times = 5)
)
estimate_spot_spacing(df)
```

`filter_out_by_endpoint_clusters` *Filter lines by endpoint cluster membership*

Description

Filters trajectory data by checking the cluster labels at each trajectory's endpoints. For every 'trajectory_id', the start endpoint is defined as the spot with the smallest projection parameter 'pos_on_seg', and the end endpoint as the spot with the largest 'pos_on_seg'. Only trajectories whose start cluster is in 'allowed_start_clusters' *and* whose end cluster is in 'allowed_end_clusters' are kept.

Usage

```
filter_out_by_endpoint_clusters(
  out,
  allowed_start_clusters,
  allowed_end_clusters
)
```

Arguments

`out`	A data frame containing selected spots with columns 'trajectory_id', 'cluster', and 'pos_on_seg', typically the output of 'build_similar_trajectories()'.
`allowed_start_clusters`	Vector of allowed cluster labels for the start endpoint.
`allowed_end_clusters`	Vector of allowed cluster labels for the end endpoint.

Details

The endpoint clusters are computed per 'trajectory_id':

- 'start_cluster = cluster[which.min(pos_on_seg)]'
- 'end_cluster = cluster[which.max(pos_on_seg)]'

If multiple spots share the same minimum/maximum 'pos_on_seg', the first is taken (as per 'which.min()' / 'which.max()').

This function uses 'dplyr' ('group_by', 'summarise', 'filter', 'pull') and the base R pipe '|>'.

Value

The same data frame as 'out', filtered to keep only the lines matching the allowed endpoint cluster constraints.

Examples

```
# Minimal example
out <- data.frame(
  trajectory_id = c("L1","L1","L2","L2"),
  pos_on_seg = c(0.0, 1.0, 0.0, 1.0),
  cluster  = c("A", "B", "A", "C"),
  x = 1:4, y = 1:4
)

# Keep only trajectories starting in A and ending in B
filter_out_by_endpoint_clusters(out, allowed_start_clusters = "A",
allowed_end_clusters = "B")
```

`get_inlaid_spots` *Get inlaid spots within a source cluster*

Description

This function retrieves all spots from a source cluster along with their inlaid/annotation values. Unlike [get_neighborhood_spots()], this returns spots *inside* the source cluster itself, not around it.

Usage

```
get_inlaid_spots(
  df,
  cluster,
  inlaid_col = "cluster",
  cluster_col = "cluster",
  coords = c("x", "y")
)
```

Arguments

`df`	A data.frame containing at least the columns ‘x‘, ‘y‘, cluster identifier column, inlaid column, and ‘spot_id‘. - ‘x‘, ‘y‘: numeric coordinates of spots - cluster column (name specified by ‘cluster_col‘): cluster assignment for each spot - inlaid column (name specified by ‘inlaid_col‘): inlaid/annotation for each spot - ‘spot_id‘: unique identifier for each spot
`cluster`	The cluster label to retrieve inlaid spots for.
`inlaid_col`	Character. Name of the column containing inlaid/annotation values. Default is ‘"cluster"‘.
`cluster_col`	Character. Name of the column containing cluster assignments. Default is ‘"cluster"‘.
`coords`	Character vector of length 2 giving the coordinate column names. Default is ‘c("x", "y")‘.

Value

A data.frame with columns:

spot_id Unique spot identifier.

x X coordinate.

y Y coordinate.

inlaid Inlaid/annotation value.

cluster The source cluster being analyzed.

is_inlaid Logical, always TRUE for returned rows.

Rows are sorted by inlaid and spot_id.

Examples

```
data("visium_simulated_spe", package = "Battlefield")
spe <- visium_simulated_spe
df <- data.frame(
  spot_id = colnames(spe),
  x = SpatialExperiment::spatialCoords(spe)[, 1],
  y = SpatialExperiment::spatialCoords(spe)[, 2],
  cluster = SummarizedExperiment::colData(spe)$cluster,
inlaid = sample(paste0("type_", 1:3), length(colnames(spe)), replace = TRUE)
)
# Get all inlaid spots within cluster 1
inlaid_spots <- get_inlaid_spots(df, cluster = 1, inlaid_col = "inlaid")
head(inlaid_spots)
```

---

get_neighborhood_params

Suggest neighborhood parameters based on detected grid type

Description

This helper detects the spatial grid type (square vs hexagonal) and returns recommended neighborhood parameters for building local adjacency / neighbor queries (e.g., for border detection or spatial graphs).

Usage

```
get_neighborhood_params(
  df,
  coords = c("x", "y"),
  square_connectivity = c(4, 8),
  tolerance = 0.1,
  verbose = TRUE
)
```

Arguments

df	A data.frame containing spatial coordinates.
coords	Character vector of length 2 giving the coordinate column names. Default is 'c("x","y")'.
square_connectivity	Integer-like choice for square grids: '4' (Von Neumann) or '8' (Moore). Default is 'c(4, 8)' which selects the first value ('4').
tolerance	Numeric. Passed to [detect_grid_type()] to match characteristic ratios (sqrt(2) / sqrt(3)). Default is 0.1.
verbose	Logical. If 'TRUE', prints diagnostic messages. Default is 'TRUE'.

Details

It relies on [detect_grid_type()] to estimate the grid step size and grid geometry, then chooses typical parameters:

- **Hexagonal (Visium)**: 6-neighborhood, radius ~ step
- **Square (Visium HD)**: 4- or 8-neighborhood, radius ~ step (4) or ~ sqrt(2)*step (8)

Returned values:

grid_type Detected grid type: '"hexagonal"', '"square"', or '"unknown"'.

connectivity Nominal grid connectivity (6 for hex, 4 or 8 for square).

r Recommended distance threshold (radius) to define adjacency.

k_max A suggested upper bound for kNN queries (used as a safe cap).

comment Human-readable description of the choice.

Notes:

- 'radius' is set to '1.01 * step' (or '1.01 * sqrt(2) * step' for #' 8-connectivity) to be slightly permissive under small numerical noise.
- 'k' is deliberately larger than the nominal connectivity to make sure enough candidates are retrieved before applying distance-based filtering.

Value

A named list with neighborhood parameters: 'grid_type', 'connectivity', 'radius', 'k', and 'comment'.

Examples

```
# Square grid example
sq <- expand.grid(x = 0:9, y = 0:9)
get_neighborhood_params(sq, square_connectivity = 4, verbose = FALSE)

# Hexagonal grid example
nx <- 10; ny <- 10
hex <- expand.grid(i = 0:(nx-1), j = 0:(ny-1))
```

```
hex$x <- hex$i + 0.5 * (hex$j %% 2)
hex$y <- (sqrt(3)/2) * hex$j
hex <- hex[, c("x","y")]
get_neighborhood_params(hex, verbose = FALSE)
```

`get_neighborhood_spots` *Get neighborhood spots around a target cluster or point*

Description

This function identifies all spots in the neighborhood of a target cluster or a specific point using k-nearest neighbors search. It returns all neighbor spots that were identified, marking which ones are neighbors.

Usage

```
get_neighborhood_spots(
  df,
  cluster = NULL,
  spot_id = NULL,
  k = 100,
  max_dist = NULL,
  inlaid_col = NULL,
  coords = c("x", "y"),
  cluster_col = "cluster"
)
```

Arguments

`df`	A data.frame containing at least the columns ‘x‘, ‘y‘, cluster identifier column, and ‘spot_id‘. - ‘x‘, ‘y‘: numeric coordinates of spots - cluster column (name specified by ‘cluster_col‘): cluster assignment for each spot - ‘spot_id‘: unique identifier for each spot
`cluster`	The cluster label for which to analyze the neighborhood. Either ‘cluster‘ or ‘spot_id‘ must be provided, but not both.
`spot_id`	Character. The spot identifier to find neighbors for. Either ‘cluster‘ or ‘spot_id‘ must be provided, but not both.
`k`	Integer. Number of nearest neighbors to consider. Default is 100. Larger values capture more distant neighbors. If k exceeds the number of spots, it will be capped at nrow(df) - 1.
`max_dist`	Numeric. Maximum distance from the target to consider. If ‘NULL‘ (default), all neighbors up to k are included without distance filtering.

inlaid_col	Character. Name of the column containing inlaid/annotation values to return in the ‘neighborhood‘ column. If ‘NULL‘ (default), uses the ‘cluster_col‘ value. This allows counting neighborhood composition by any column (e.g., cell type, tissue type).
coords	Character vector of length 2 giving the coordinate column names. Default is ‘c("x", "y")‘.
cluster_col	Character. Name of the column containing cluster assignments. Default is ‘"cluster"‘.

Details

The function can work in two modes: - **Cluster mode**: If ‘cluster‘ is provided, computes k-nearest neighbors for all spots in that cluster. Returns all neighbors found (excluding source cluster spots). - **Point mode**: If ‘spot_id‘ is provided, finds the k-nearest neighbors to specific spot.

Value

A data.frame with columns:

spot_id Unique spot identifier.

x X coordinate.

y Y coordinate.

neighborhood Value from ‘inlaid_col‘ (or cluster if inlaid_col is NULL) of the neighbor spot.

cluster The source cluster or spot_id being analyzed.

is_neighborhood Logical, always TRUE for returned rows.

Rows are sorted by neighborhood and spot_id.

Examples

```
data("visium_simulated_spe", package = "Battlefield")
spe <- visium_simulated_spe
df <- data.frame(
  spot_id = colnames(spe),
  x = SpatialExperiment::spatialCoords(spe)[, 1],
  y = SpatialExperiment::spatialCoords(spe)[, 2],
  cluster = SummarizedExperiment::colData(spe)$cluster
)
# Get all neighbor spots for cluster 1
neighbors_cluster <- get_neighborhood_spots(df, cluster = 1, k = 50)
head(neighbors_cluster)

# Get neighbors for a specific point
first_spot <- df$spot_id[1]
neighbors_point <- get_neighborhood_spots(df, spot_id = first_spot, k = 10)
head(neighbors_point)
```

`point_segment_distance_vec` *Point-to-segment distance (vectorized)*

Description

Computes the Euclidean distance from one or more points ‘(px, py)‘ to the line segment defined by endpoints ‘A(ax, ay)‘ and ‘B(bx, by)‘. The projection is clamped to the segment (i.e., ‘pos_on_seg‘ is restricted to ‘[0, 1]‘).

Usage

```
point_segment_distance_vec(px, py, ax, ay, bx, by)
```

Arguments

`px, py`	Numeric vectors (or scalars) giving the x/y coordinates of the query point(s).
`ax, ay`	Numeric scalars giving the x/y coordinates of endpoint A.
`bx, by`	Numeric scalars giving the x/y coordinates of endpoint B.

Details

Let ‘v = B - A‘ and ‘pos_on_seg = ((P - A) · v) / ||v||^2‘. The closest point on the infinite line is ‘A + pos_on_seg v‘; clamping ‘pos_on_seg‘ to ‘[0, 1]‘ yields the closest point on the segment.

If ‘A‘ and ‘B‘ are identical (‘||v||^2 == 0‘), the function returns the distance from ‘P‘ to ‘A‘.

Value

A numeric vector of distances, with length equal to ‘max(length(px), length(py))‘ (after R’s usual vector recycling rules).

Examples

```
# Single point to a horizontal segment
point_segment_distance_vec(px = 1, py = 2, ax = 0, ay = 0, bx = 3, by = 0)

# Multiple points (vectorized)
px <- c(0, 1, 2, 3)
py <- c(1, 1, 1, 1)
point_segment_distance_vec(px, py, ax = 0, ay = 0, bx = 3, by = 0)

# Degenerate segment (A == B)
point_segment_distance_vec(px = c(0, 1), py = c(0, 1), ax = 0, ay = 0, bx =
0, by = 0)
```

`remove_used_points` *Remove rows whose rounded (x, y) coordinates have already been used*

Description

Filters ‘df‘ to drop any rows whose rounded coordinate key (as produced by ‘.xy_key(x, y)‘) appears in ‘used_df‘.

Usage

```
remove_used_points(df, used_df)
```

Arguments

`df`	A data frame containing at least columns ‘x‘ and ‘y‘.
`used_df`	A data frame containing at least columns ‘x‘ and ‘y‘ defining the set of already-used points to exclude.

Details

Matching is performed on rounded coordinates, not exact floating-point values. Internally, keys are computed as ‘paste0(round(x), "_", round(y))‘.

This function uses ‘dplyr::filter()‘ (and the ‘|>‘ pipe), so ‘dplyr‘ must be installed and a pipe operator must be available.

Value

A filtered data frame with the same columns as ‘df‘, excluding rows whose rounded ‘(x, y)‘ matches any row in ‘used_df‘.

Examples

```
df <- data.frame(x = c(0.1, 1.2, 1.6, 2.0),
                 y = c(0.2, 3.4, 3.4, 9.5),
                 id = 1:4)
used_df <- data.frame(x = c(1.49, 2.1),
                      y = c(3.49, 9.49))

# Removes rows rounding to (1,3) and (2,9)
remove_used_points(df, used_df)
```

`select_border_spots` *Select border spots from cluster A that touch cluster B (and flag junctions)*

Description

This function identifies **directed interface spots**: spots belonging to ‘cluster‘ (A) that have at least one neighbor in ‘interface‘ (B) within a k-nearest-neighbors neighborhood (optionally constrained by a distance cutoff).

Usage

```
select_border_spots(
  df,
  cluster,
  interface,
  mode = "inner",
  k = 7,
  max_dist = NULL,
  coord_cols = c("x", "y"),
  cluster_col = "cluster"
)
```

Arguments

`df`	A data.frame containing at least the coordinate columns and a cluster label column.
`cluster`	Cluster label for the source cluster (A). Spots in this cluster are tested for adjacency to ‘interface‘.
`interface`	Cluster label for the target cluster (B).
`mode`	Character. One of "inner", "outer", or "both". - "inner": returns border spots from cluster → interface (default). - "outer": returns border spots from interface → cluster (swaps cluster/interface). - "both": returns border spots from both directions, row-bound together. Default is "inner".
`k`	Integer. Number of nearest neighbors to consider (excluding self). Internally uses ‘k + 1‘ to include self then removes it. Default is 7.
`max_dist`	Optional numeric. Maximum Euclidean distance for a neighbor to be considered (distance cutoff). If ‘NULL‘, no distance filtering is applied.
`coord_cols`	Character vector of length 2 giving the coordinate column names. Default is ‘c("x","y")‘.
`cluster_col`	Character. Name of the column containing cluster labels. Default is ‘"cluster"‘.

Details

In addition, it flags spots that also touch **other clusters** (i.e., junction / multi-interface spots) and reports which other clusters are touched.

Steps:

1. Builds a kNN neighborhood for each spot in ‘cluster‘.
2. Optionally removes neighbors farther than ‘max_dist‘.
3. Marks a spot as a border spot if it has at least one neighbor in ‘interface‘.
4. Flags a spot as multi-interface if it also has a neighbor belonging to a cluster different from ‘cluster‘ and ‘interface‘.

Note on mode column: The ‘mode‘ column in the returned data.frame will always contain either "inner" or "outer", even if the ‘mode‘ parameter is set to "both". When ‘mode="both"‘, the function internally calls the selection algorithm twice (once for cluster→interface and once for interface→cluster), then row-binds the results, so each row is labeled with its actual direction.

The returned data.frame contains only spots from ‘cluster‘ that touch ‘interface‘, with columns:

spot_id Spot identifier.

x X coordinate.

y Y coordinate.

cluster Cluster label (same as ‘cluster‘).

interface The value of ‘interface‘.

directed_pair Interface label combining cluster and ‘interface‘, e.g. ‘"A-B"‘.

undirected_pair Undirected pair label (both directions), e.g. ‘"A-B / B-A"‘.

is_border Always ‘TRUE‘ for returned rows (kept for clarity).

is_border_multiple ‘TRUE‘ if the spot also touches other clusters.

other_adjacent_borders Comma-separated list of other clusters touched, or ‘NA‘.

mode The mode used for selection: "inner", "outer", or "both".

Value

A data.frame subset of ‘df‘ containing border spots from ‘cluster‘ to ‘interface‘ with columns: spot_id, x, y, cluster, interface, directed_pair, is_border, is_border_multiple, other_adjacent_borders.

Examples

```
# Minimal example (requires RANN and dplyr)
set.seed(1)
df_ex <- data.frame(
  x = rnorm(200),
  y = rnorm(200),
  cluster = sample(c("A","B","C"), 200, replace = TRUE)
)
res <- select_border_spots(df_ex, cluster = "A", interface = "B", k = 7)
head(res)
```

`select_core_spots` *Select core (non-interface) spots for a directed pair*

Description

This function selects core (non-interface) spots from a cluster that match the count of border spots for a specific directed pair. The ‘mode‘ value is read from the ‘border_df‘ dataframe for this cluster/interface pair. If multiple mode values exist in border_df for the pair, a warning is printed and the first value is used.

Usage

```
select_core_spots(
  df,
  border_df,
  cluster,
  interface,
  mode = "both",
  coord_cols = c("x", "y"),
  cluster_col = "cluster"
)
```

Arguments

`df`	A data.frame containing at least the coordinate columns and a cluster label column.
`border_df`	A data.frame of border spots from [build_all_borders()], containing columns ‘spot_id‘, ‘cluster‘, ‘interface‘, ‘directed_pair‘, etc.
`cluster`	Character. Label of the cluster from which to select inner spots.
`interface`	Character. Label of the target cluster for the directed pair.
`mode`	Character. One of "inner", "outer", or "both". Controls which mode values from border_df to use when counting border spots. When mode="both", both "inner" and "outer" mode rows from border_df are used. Default is "both".
`coord_cols`	Character vector of length 2 giving the coordinate column names. Default is ‘c("x","y")‘.
`cluster_col`	Character. Name of the column containing cluster labels. Default is ‘"cluster"‘.

Details

For example with ‘cluster="1"‘ and ‘interface="2"‘: - If mode in df is "inner": counts only border spots from 1→2 - If mode in df is "outer": counts only border spots from 2→1 - If mode in df is "both": counts border spots from both 1→2 AND 2→1

This returns N random inner spots from the cluster, where N equals the border count.

Steps:

1. Counts directed border spots based on ‘mode‘ parameter.
2. Gets all spots in ‘cluster‘ that are not at any border.
3. Randomly samples the same number of core spots as the border count.
4. Returns these sampled core spots with ‘interface‘ annotation.

If not enough core spots exist to match the border count, all available core spots are returned with a warning.

Value

A data.frame of sampled core spots from ‘cluster‘, with all columns from ‘df‘ plus ‘interface‘ annotation.

Examples

```
# Create synthetic spatial transcriptomics data with 3 clusters
df_ex <- data.frame(
  spot_id = paste0("spot_", seq_len(300)),
  x = rnorm(300),
  y = rnorm(300),
  cluster = sample(c("A", "B", "C"), 300, replace = TRUE)
)

# Step 1: Identify all border spots between clusters
all_borders <- build_all_borders(df_ex, k = 6)

# Step 2: Select core spots for the directed pair A→B
# matching the number of border spots found
cores_inner <- select_core_spots(
  df_ex,
  border_df = all_borders,
  cluster = "A",
  interface = "B",
  mode = "inner"
)
head(cores_inner)

# Step 3: Select core spots considering both directions (A→B AND B→A)
cores_both <- select_core_spots(
  df_ex,
  border_df = all_borders,
  cluster = "A",
  interface = "B",
  mode = "both"
)
head(cores_both)
```

`shift_point` *Shift a point along a given direction vector*

Description

Translates a point ‘P‘ by an amount ‘offset‘ in the direction ‘(nx, ny)‘. The returned point is:

$$P' = (P_x + offset \cdot nx,\ P_y + offset \cdot ny).$$

Usage

```
shift_point(P, nx, ny, offset)
```

Arguments

`P`	A data frame with columns ‘x‘ and ‘y‘ representing the point to shift. If it contains multiple rows, only the first row is used.
`nx`	Numeric scalar; x-component of the direction vector.
`ny`	Numeric scalar; y-component of the direction vector.
`offset`	Numeric scalar; translation magnitude (in the same units as ‘x‘/‘y‘). Positive values move in the ‘(nx, ny)‘ direction; negative values move in the opposite direction.

Value

A one-row data frame with columns ‘x‘ and ‘y‘ giving the shifted point.

Examples

```
# Shift the point (1, 2) by 3 units in the direction (0, 1)
P <- data.frame(x = 1, y = 2)
shift_point(P, nx = 0, ny = 1, offset = 3)
```

`unit_normal_left` *Compute the left unit normal of the directed segment A->B*

Description

Returns the unit-length normal vector pointing to the left of the directed segment $A \to B$. For a direction vector $v = (vx, vy)$, the left normal is $(-vy, vx)$ normalized by $||v||$.

Usage

```
unit_normal_left(A, B)
```

Arguments

A	A data frame with columns 'x' and 'y' representing point A. If it contains multiple rows, only the first row is used.
B	A data frame with columns 'x' and 'y' representing point B. If it contains multiple rows, only the first row is used.

Value

A numeric vector of length 2 with named components:

nx x-component of the left unit normal.

ny y-component of the left unit normal.

Examples

```
# Horizontal segment to the right: (0,0) -> (1,0)
#    Left normal points upward: (0, 1)
A <- data.frame(x = 0, y = 0)
B <- data.frame(x = 1, y = 0)
unit_normal_left(A, B)
```

---

visiumHD_16um_simulated_spe

Simulated VisiumHD 16 µm binned SpatialExperiment dataset

Description

A simulated `SpatialExperiment` object mimicking a 10x Genomics Visium spatial transcriptomics layout. The dataset contains spot-level spatial coordinates and cluster annotations, and is intended for testing.

Usage

```
data(visiumHD_16um_simulated_spe)
```

Format

A `SpatialExperiment` object with:

assays One dummy assay (required container structure)

colData Spot metadata including `barcode` and `cluster`

spatialCoords Numeric matrix of x/y spot coordinates

Details

This dataset contain only gene expression values for one gene named FAKE_GENE. The dummy assay is included solely to satisfy the `SummarizedExperiment` container requirements.

Spot coordinates follow a Visium-like squared grid. Cluster labels are simulated and have no biological meaning.

The layout is intended to mimic a Visium HD *binned* layer (e.g., 16 µm bin size), where each spot represents one bin arranged on an adjacent square grid.

Source

Simulated internally for package development.

See Also

- `SpatialExperiment`
- `compute_centroids`

visiumHD_16um_simulated_spe

Examples

```
data(visiumHD_16um_simulated_spe)

visiumHD_16um_simulated_spe
```

---

visiumHD_8um_simulated_spe

Simulated VisiumHD 8 µm binned SpatialExperiment dataset

Description

A simulated `SpatialExperiment` object mimicking a 10x Genomics Visium spatial transcriptomics layout. The dataset contains spot-level spatial coordinates and cluster annotations, and is intended for testing.

Usage

```
data(visiumHD_8um_simulated_spe)
```

Format

A `SpatialExperiment` object with:

assays One dummy assay (required container structure)

colData Spot metadata including `barcode` and `cluster`

spatialCoords Numeric matrix of x/y spot coordinates

Details

This dataset contain only gene expression values for one gene named FAKE_GENE. The dummy assay is included solely to satisfy the `SummarizedExperiment` container requirements.

Spot coordinates follow a Visium-like squared grid. Cluster labels are simulated and have no biological meaning.

The layout is intended to mimic a Visium HD *binned* layer (e.g., 8 µm bin size), where each spot represents one bin arranged on an adjacent square grid.

Source

Simulated internally for package development.

See Also

- `SpatialExperiment`
- `compute_centroids`

visiumHD_8um_simulated_spe

Examples

```
data(visiumHD_8um_simulated_spe)

visiumHD_8um_simulated_spe
```

---

visium_simulated_spe *Simulated Visium SpatialExperiment dataset*

Description

A simulated `SpatialExperiment` object mimicking a 10x Genomics Visium spatial transcriptomics layout. The dataset contains spot-level spatial coordinates and cluster annotations, and is intended for testing.

Usage

```
data(visium_simulated_spe)
```

Format

A `SpatialExperiment` object with:

assays One dummy assay (required container structure)

colData Spot metadata including `barcode` and `cluster`

spatialCoords Numeric matrix ofx/y spot coordinates

Details

This dataset contain only gene expression values for one gene named FAKE_GENE. The dummy assay is included solely to satisfy the `SummarizedExperiment` container requirements.

Spot coordinates follow a Visium-like hexagonal grid. Cluster labels are simulated and have no biological meaning.

Source

Simulated internally for package development.

See Also

- `SpatialExperiment`
- `compute_centroids`

visium_simulated_spe

Examples

```
data(visium_simulated_spe)

visium_simulated_spe
```

Index

∗ **datasets**
visium_simulated_spe, 46
visiumHD_16um_simulated_spe, 44
visiumHD_8um_simulated_spe, 45
∗ **internal**
Battlefield-package, 3

add_borders_to_spe, *3*, 5
add_layers_to_spe, *3*, 6
add_trajectories_to_spe, *3*, 7
adjacent_endpoint, 8
aggregate, *18*

Battlefield *(*Battlefield-package*)*, 3
Battlefield-package, 3
bresenham_line, *3*, 9
build_all_borders, *3*, 10
build_all_cores, 11
build_one_line, 13
build_one_trajectory, *3*, 14
build_similar_trajectories, *3*, 15

closest_spot, 17
compute_centroids, 18, *45–47*
count_all_inlaids, 19
count_all_neighborhoods, 20
count_inlaid, 22
count_neighborhood, *3*, 23
create_all_layers, *3*, 24
create_cluster_layers, *3*, 26

detect_grid_type, *3*, 27
directed_cluster_interface_pairs, *3*, 29

estimate_spot_spacing, 30

filter_out_by_endpoint_clusters, 31

get_inlaid_spots, 32
get_neighborhood_params, 33
get_neighborhood_spots, *3*, 35

point_segment_distance_vec, 37

remove_used_points, 38

select_border_spots, *3*, 39
select_core_spots, 41
shift_point, 43
SpatialExperiment, *3*, *44–47*

unit_normal_left, 43

visium_simulated_spe, *4*, 46
visiumHD_16um_simulated_spe, *4*, 44
visiumHD_8um_simulated_spe, *4*, 45