Package: rasengan (via r-universe)

May 22, 2026

Title Generation of Geometric Curves

Version 0.0.16

Description Provides functions to generate and sample geometric curves. Each function returns a data frame of 2D coordinates, suitable for visualization or further geometric processing.

License MIT + file LICENSE

Depends R (>= 4.1.0)

Imports affiner, cli, dplyr, purrr, rlang

Suggests carrier (>= 0.3.0), farver, ggplot2, grDevices, mirai (>= 2.5.1), testthat (>= 3.0.0), vdiffr

LinkingTo cpp11

Config/roxygen2/version 8.0.0

Config/testthat/edition 3

Encoding UTF-8

Roxygen list(markdown = TRUE)

Repository https://paithiov909.r-universe.dev

Date/Publication 2026-05-22 15:52:02 UTC

RemoteUrl https://github.com/paithiov909/rasengan

RemoteRef HEAD

RemoteSha bb205da8dd2a87b48d46c5c8648b44198bd9b211

Contents

`as_pattern` *Cast a vector into a native raster*

Description

Casts a vector of pixel values into a native raster. The vector must be of length `width * height`

Usage

```
as_pattern(x, width, height, ...)
```

Arguments

`x`	A vector to be cast into a native raster.
`width`, `height`	Integer scalars giving the width and height of the image.
`...`	Additional arguments.

Value

A `nativeRaster` object.

`bezier` *Bezier curve utilities*

Description

Functions for evaluating cubic Bezier curves and their associated vector quantities. A cubic Bezier curve is defined by four control points P_0, P_1, P_2, P_3, where P_0 and P_3 are endpoints and P_1, P_2 act as handles controlling the tangent directions.

All functions accept a 8 column numeric matrix or data frame containing control points, where each row encodes the four points as: `(x0, y0, x1, y1, x2, y2, x3, y3)`.

Usage

```
curve_bezier(n, control_pts)

bezier_derivative(n, control_pts)

bezier_normal(n, control_pts)

bezier_tangent(n, control_pts)
```

Arguments

`n`	An integer scalar; Number of points to sample along the curve.
`control_pts`	A numeric matrix or data frame of control points.

Details

- `curve_bezier(n, control_pts)`: Evaluate cubic Bezier curves at `n` evenly spaced parameter values in $t \in [0, 1]$. Returns a table of `(x, y)` positions for each curve `id`.
- `bezier_derivative(n, control_pts)`: Compute first derivatives of each curve at `n` points. The result contains the velocity vectors `(dx, dy)` at each `t`.
- `bezier_normal(n, control_pts)`: Compute unit normal vectors `(nx, ny)` along each curve. Normals are derived from the first derivative and represent the direction perpendicular to the tangent at each point.
- `bezier_tangent(n, control_pts)`: Compute unit tangent vectors `(tx, ty)` along each curve. Tangents represent the direction of movement along the curve and are obtained by normalizing the first derivative.

These functions are useful for geometric processing, constructing smooth interpolations, and generating direction fields for animation or stroke-based visual effects.

Value

A tibble.

See Also

compute_handles()

Other curve: curve-others

---

camera *3D world to camera transformation*

Description

3D world to camera transformation

Usage

```
lookat3d(eye, center, up = c(0, 1, 0))

persp3d(fovy, aspect, near = 0.1, far = 10)

viewport3d(width, height, ox = 0, oy = 0)
```

Arguments

eye	A numeric vector of length 3 giving the position of the camera.
center	A numeric vector of length 3 giving the position where the camera is looking at.
up	A numeric vector of length 3 giving the direction of the "up" vector for the camera.
fovy	A numeric scalar giving the field of view in radians.
aspect	A numeric scalar giving the aspect ratio.
near	A numeric scalar giving the distance to the near plane.
far	A numeric scalar giving the distance to the far plane.
width, height	A numeric scalar giving the width and height of the viewport.
ox, oy	A numeric scalar giving the offset of the viewport in pixels.

Value

A transform3d object.

See Also

Other camera: ndc_mul()

`chakra` *Methods for chakra objects*

Description

A `chakra` in rasengan is an object that evolves over time and can be observed as ordinary R data.

The methods `observe()`, `circulate()`, and `reset_state()` are generic functions that dispatch on the class of `x`. The name `chakra` is used as a conceptual interface rather than as a concrete S3 class.

Usage

```
observe(x, ...)

circulate(x, dt = 1, n_steps = 1, ...)

reset_state(x, ...)
```

Arguments

`x`	A chakra-like object.
`...`	Additional arguments passed to methods.
`dt`	A numeric scalar giving the time step.
`n_steps`	An integer scalar giving the number of steps to advance.

Details

Chakra objects may be implemented in different ways.

Some chakras are mutable external states backed by external pointers. For these objects, `circulate()` may update the underlying state in place.

Other chakras are pseudo-states. A pseudo-state stores enough parameters to compute its current observation, but does not necessarily mutate an external pointer. For these objects, `circulate()` usually returns a modified copy of `x`.

Because implementations may differ, users should generally assign the return value of `circulate()`:

```
x <- circulate(x, dt = 1 / 60)
observe(x)
```

The following generics are available:

- `observe(x, ...)`: Returns the current state as a tibble.
- `circulate(x, dt, n_steps, ...)`: Evolves `x` for `n_steps`.
- `reset_state(x, ...)`: Resets `x` to an initial or specified state, when supported.

Not all chakra implementations need to support meaningful reset behavior. For pseudo-states whose state is represented directly by an R object, it may be simpler to recreate the object instead of calling `reset_state()`.

Value

- `observe()` returns a tibble representing the current state.
- `circulate()` returns the evolved object. Some methods may also mutate the underlying state in place.
- `reset_state()` returns the reset object, when supported.

`compute_handles` *Construct Bezier control points from a sequence of positions*

Description

Generates cubic Bezier control points from an ordered sequence of 2D points. Given a `k * 2` matrix (or data frame) of points $P_0, P_1, \ldots, P_{k-1}$, this function returns a tibble representing a set of Bezier segments. Each output row encodes one cubic Bezier curve segment with control points `(x0, y0, x1, y1, x2, y2, x3, y3)`, forming a curve from `P_i` to `P_{i+1}`.

Different handle–generation strategies can be selected via `method`.

Usage

```
compute_handles(
  points,
  method = c("catmull", "linear", "normal", "tangent"),
  tension = 0.3,
  scale = 0.3,
  side = c("left", "right")
)
```

Arguments

`points`	A `k * 2` matrix or data frame of points. Each row is `(x, y)`.
`method`	Handle–generation strategy. One of `"catmull"`, `"linear"`, `"normal"`, `"tangent"`.
`tension`	Numeric scaling factor for `"linear"` method (default `0.3`).
`scale`	Numeric scaling factor for `"normal"` and `"tangent"` methods (default `0.3`).
`side`	For `"normal"` method, choose `"left"` or `"right"` to determine the orientation of the normal vector.

Details

These helper functions are intended to be used together with `curve_bezier()` and the vector–field functions `bezier_tangent()`, `bezier_normal()`, and `bezier_derivative()`. They are useful for smoothing polyline data, constructing interpolating curves, and generating stylized shapes for creative coding.

`method = "catmull":`

Uses Catmull–Rom spline to construct smooth Bezier segments. For each segment $P_i \to P_{i+1}$, interior points are computed as:

$$B_1 = P_i + (P_{i+1} - P_{i-1})/6, \quad B_2 = P_{i+1} - (P_{i+2} - P_i)/6.$$

Produces `k - 3` segments (requires 4 or more points).

`method = "linear":`

Places both handles along the segment direction. Handles are spaced from the endpoints by `tension * distance(P_i, P_{i+1})`. Produces `k - 1` straight-style Bezier segments.

`method = "normal":`

Handles are placed perpendicular to the segment direction, producing decorative, "bent" or "blooming" curves. The handle offset is scale $\times \|P_{i+1} - P_i\|$. Use `side = "left"` or `"right"` to control the normal direction.

`method = "tangent":`

Computes tangents using centered differences, giving a smooth, spline-like interpolation. The handle offset is scale $\times \|P_{i+1} - P_i\|$. Produces `k - 1` C^1–continuous Bezier segments.

Value

A tibble with columns `x0, y0, x1, y1, x2, y2, x3, y3`, containing one Bezier segment per row.

See Also

`curve_bezier()`

Examples

```
pts <- matrix(rnorm(20), ncol = 2)

# Tangent-based smoothing
segs <- compute_handles(pts, method = "tangent")
curve_bezier(n = 50, segs)
```

`curve-others` *Generate geometric curves*

Description

Generates a data frame of points along a geometric curve.

`"curve"` in rasengan is a generic term for a set of functions that take parameters `n` and generate a data frame with `n` rows and columns containing `x` and `y` along the geometric curve.

Archimedean spirals:

`curve_archimedean` generates a 2D curve that is a general Archimedean spiral where the raidus `r` is defined as follows:

$$r = a + b \cdot \theta^{\frac{1}{c}}$$

According to the Wikipedia article, the normal Archimedean spiral occurs when `c = 1`. Other spirals falling into this group include the hyperbolic spiral (`c = -1`), Fermat's spiral (`c = 2`), and the lituus (`c = -2`).

Usage

```
curve_archimedean(n, a = 0, b = 1, c = 1, base = exp(1))

curve_cyclic_harmonic(n, k = 5, a = 1, b = 0.5, scale = 1, base = exp(1))

curve_epicycloid(n, or = 16, ir = 3.5, scale = 1, base = exp(1))

curve_epitrochoid(n, or = 16, ir = 3.5, b = 2.4, scale = 1, base = exp(1))

curve_gear(n, k = 10, a = 1, b = 16, scale = 1, base = exp(1))

curve_heart(n, scale = -1)

curve_hypocycloid(n, or = 16, ir = 3.5, scale = 1, base = exp(1))

curve_hypotrochoid(n, or = 16, ir = 3.5, b = 2.4, scale = 1, base = exp(1))

curve_lissajous(n, d = 10, e = 16, scale = 1, base = exp(1))

curve_ranunculoid(n, k = 6, scale = 1/k, base = exp(1))

curve_rose(n, k = 5, c = 1, scale = 1, base = exp(1))

curve_spirograph(n, or = 16, ir = 3.5, b = 2.4, scale = 1, base = exp(1))
```

Arguments

`n`	An integer scalar; Number of points to sample along the curve.
`a`, `b`, `c`, `d`, `e`, `k`	Numeric scalars; Parameters of the curve.
`base`	A numeric scalar; Base of the logarithm used to compute the spacing between points.
`scale`	A numeric scalar; Scaling factor for the curve.
`or`, `ir`	Numeric scalars; Outer and inner radius.

Value

- For polar equations, a tibble with columns `id`, `phi`, `r`, `x`, and `y`.
- For others, a tibble with columns `id`, `theta`, `x`, and `y`.

See Also

Other curve: `bezier`

`delaunay` *Compute Delaunay triangulation*

Description

Computes the Delaunay triangulation of a set of 2D points.

Usage

```
delaunay(seeds, x = x, y = y)
```

Arguments

`seeds`	A data frame containing the input points.
`x`, `y`	<`tidy-select`> Columns in `seeds` that contain the x- and y-coordinates.

Details

This function returns the triangulation in a form that is convenient for drawing. `vertices` stores the triangle vertices in long format, with three rows per triangle. `circumcenters` stores the corresponding circumcircle of each triangle.

Value

A list with two data frames:

- `circumcenters`: A tibble containing the triangle id, the x- and y-coordinates of the circumcenter, and the circumradius.
- `vertices`: A tibble containing the triangle id and the x- and y-coordinates of the three vertices of each triangle.

Examples

```
theta <- seq(-pi, pi, length.out = 12)
seeds <- data.frame(
  x = c(150 * cos(theta), 0),
  y = c(150 * sin(theta), 0)
)

tri <- delaunay(seeds)

tri$circumcenters
tri$vertices
```

---

expand *Expand grid*

Description

A thin wrapper for `expand.grid()` that returns a tibble while converting numeric columns to double.

Usage

```
expand(...)
```

Arguments

...	Arguments to be passed to `expand.grid()`.

Value

A tibble.

---

fbm *Generators for FBM and FBB*

Description

Creates a function that generates a fractional Brownian motion (FBM) or fractional Brownian bridge (FBB) time series with a given Hurst index.

Usage

```
fbm_from(power = 4, hurst_index = 0.5)

fbbridge_from(power = 4, hurst_index = 0.5)

fbbridge_2d_from(power = 4, hurst_index = 0.5)
```

Arguments

`power`	Integer.
`hurst_index`	Numeric in range `(0, 1)`.

Value

A function that takes a function `func` as its first argument and returns a time series (for `fbm_from()` and `fbbridge_from()`) or a matrix (for `fbm_2d_from()`). `func` is expected to take `n` as its first argument and return a numeric vector of length `n`.

`misc` *Miscellaneous functions*

Description

Miscellaneous functions

Usage

```
deg2rad(x)

rad2deg(x)

fract(x)

mag(mat, origin = c(0, 0))

pingpong(x, ...)
```

Arguments

`x`	A numeric vector.
`mat`	A numeric matrix or a data frame.
`origin`	A numeric vector to be subtracted from `mat`.
`...`	Additional arguments.

Value

A numeric vector.

modifications *Simply value modifications*

Description

Some of these functions are derived from the ambient package.

Usage

```
blend(x, y, mask)

normalise(x, margin = 2)

normalize(x, from = range(x), to = c(0, 1))

cap(x, lower = 0, upper = 1)

pulse(x, mask)

wrap(x, lower, upper)
```

Arguments

x, y	A numeric vector.
mask	A numeric scalar, typically between 0 and 1.
margin	margin for apply(). Ignored if x is not a matrix.
from	A numeric vector of length 2. The range of x to use for normalization.
to	A numeric vector of length 2. The output domain to normalize to.
lower, upper	A numeric scalar. The lower and upper bounds to cap to.

Value

A numeric vector.

ndc_mul *Perspective division*

Description

Multiplication of two matrices after normalizing the first one.

Usage

```
ndc_mul(lhs, rhs)

lhs %!*% rhs
```

Arguments

`lhs`, `rhs` A numeric matrix.

Value

A numeric matrix.

See Also

Other camera: `camera`

`noise` *Create a noise generator*

Description

Creates a noise generator function that wraps FastNoiseLite.

Usage

```
noise_2d(
  noise_type = c("OpenSimplex2", "OpenSimplex2S", "Cellular", "Perlin", "ValueCubic",
    "Value"),
  frequency = 0.01,
  fractal_type = c("None", "FBm", "Rigid", "PingPong"),
  octaves = 3,
  lacunarity = 2,
  gain = 0.5,
  weighted_strength = 0,
  ping_pong_strength = 2,
  distance_function = c("EuclideanSq", "Euclidean", "Manhattan", "Hybrid"),
  return_type = c("Distance", "CellValue", "Distance2", "Distance2Add", "Distance2Sub",
    "Distance2Mul", "Distance2Div"),
  jitter = 1
)

noise_3d(
  noise_type = c("OpenSimplex2", "OpenSimplex2S", "Cellular", "Perlin", "ValueCubic",
    "Value"),
  frequency = 0.01,
  fractal_type = c("None", "FBm", "Rigid", "PingPong"),
  octaves = 3,
  lacunarity = 2,
  gain = 0.5,
  weighted_strength = 0,
  ping_pong_strength = 2,
  distance_function = c("EuclideanSq", "Euclidean", "Manhattan", "Hybrid"),
```

```
  return_type = c("Distance", "CellValue", "Distance2", "Distance2Add", "Distance2Sub",
    "Distance2Mul", "Distance2Div"),
  jitter = 1,
  rotation_type = c("None", "ImproveXYPlanes", "ImproveXZPlanes")
)
```

Arguments

`noise_type`	A string; Noise type to use.
`frequency`	A numeric scalar; Frequency.
`fractal_type`	A string; Fractal type.
`octaves`	A numeric scalar; Number of octaves.
`lacunarity`	A numeric scalar; Lacunarity (the frequency multiplier between each octave).
`gain`	A numeric scalar; Gain (the relative strength of noise from each layer when compared to the last).
`weighted_strength`	A numeric scalar; Weighted strength for fractal noise. Keep between `0` and `1` to maintain `[-1, 1]` output bounding.
`ping_pong_strength`	A numeric scalar; Ping-pong strength for 'PingPong' fractal noise.
`distance_function`	A string; Distance function for cellular noise.
`return_type`	A string; Return type for cellular noise.
`jitter`	A numeric scalar; Jitter for cellular noise.
`rotation_type`	A string; Rotation type for 3D noise.

Value

A function that takes arguments:

- `x`, `y`, (or just `data`) and `seed` for 2D noise
- `x`, `y`, `z`, (or just `data`) and `seed` for 3D noise

and returns noise values. Note that `seed` is set to a random value by default, so if you want to use the same seed for multiple calls, you need to explicitly set it.

See Also

Documentation · Auburn/FastNoiseLite Wiki

Examples

```
if (requireNamespace("dplyr", quietly = TRUE)) {
  nz <- expand(
    id = seq_len(4),
    x = seq(0, 32, by = 2),
```

```
    y = seq(0, 32, by = 2)
  ) |>
    dplyr::mutate(
      val = noise_3d()(data = dplyr::pick(id, x, y))
    )
}
## Not run:
if (require("ggplot2", quietly = TRUE)) {
  ggplot(nz) +
    geom_tile(aes(x = x, y = y, fill = val)) +
    facet_wrap(~ id)
}

## End(Not run)
```

`path_clothoid` *Generate an Euler spiral or biarc curve path*

Description

Computes a path connecting two points, each with a specified orientation angle. Two methods are available: a single Euler spiral, or a biarc composed of two spiral segments.

Usage

```
path_clothoid(
  start = c(0, 0, pi/4),
  end = c(25, 10, pi),
  max_n = 100,
  max_iter_num = 1000,
  biarch = TRUE
)
```

Arguments

`start`	A numeric vector of length 3 specifying the starting point and heading direction: `c(x, y, theta)`, where `theta` is the orientation angle in radians.
`end`	A numeric vector of length 3 specifying the goal point and heading direction: `c(x, y, theta)`, where `theta` is the orientation angle in radians.
`max_n`	A numric scalar; Maximum number of points to sample along the path.
`max_iter_num`	A numric scalar; Maximum number of iterations used in the internal parameter estimation algorithm (used only when `biarch = FALSE`).
`biarch`	A logical scalar; If `TRUE`, use a biarc approximation consisting of two Euler spiral segments. If `FALSE`, compute a single spiral curve via numerical estimation.

Details

When `biarch = TRUE`, the path is constructed from two spiral segments computed using an internal algorithm that attempts to connect the start and goal points with \(G^1 \) continuity. This approach may be more robust in cases where a single spiral does not converge well.

When `biarch = FALSE`, the function attempts to compute a single Euler spiral that smoothly interpolates between the two endpoints and their respective directions. This may fail to converge in some configurations.

Value

A data frame with columns `x`, `y`, and `theta`, giving the sampled positions and orientations along the generated path. If the parameter estimation fails, the returned data may contain missing values, which are automatically removed.

See Also

CoffeeKumazaki/euler_spiral

Other path: `path_mouse()`

Examples

```
## Not run:
path <- path_clothoid()
with(path, plot(x, y, type = "l", asp = 1))

## End(Not run)
```

---

path_mouse *Generate a human-like mouse movement path*

Description

Simulates a human-like mouse movement trajectory between two points using a physics-inspired algorithm with gravity, wind, and random waiting times.

Usage

```
path_mouse(
  start = c(0, 0),
  end = c(100, 100),
  mouse_speed = 3,
  gravity = 9,
  wind = 3,
  min_wait = 5,
  max_wait = 15,
  max_step = 10,
```

```
  target_area = 10,
  seed = sample.int(1337, 1)
)

wind_mouse(
  start = c(0, 0),
  end = c(100, 100),
  mouse_speed = 3,
  gravity = 9,
  wind = 3,
  min_wait = 5,
  max_wait = 15,
  max_step = 10,
  target_area = 10,
  seed = sample.int(1337, 1)
)
```

Arguments

`start`	A numeric vector of length 2 giving the starting coordinates (x, y).
`end`	A numeric vector of length 2 giving the target coordinates (x, y).
`mouse_speed`	A numeric scalar. The base speed factor that affects the dynamics of movement.
`gravity`	A numeric scalar that controls the strength of "pull" toward the target.
`wind`	A numeric scalar that controls the random "wind-like" movement during the motion.
`min_wait`	A numeric scalar. Minimum wait time (ms) per step, affects timestamp `t`.
`max_wait`	A numeric scalar. Maximum wait time (ms) per step.
`max_step`	A numeric scalar. Maximum movement length per iteration.
`target_area`	A numeric scalar. Radius within which the movement slows down and stabilizes.
`seed`	An integer scalar. Random seed for reproducible paths.

Details

This function generates a sequence of points resembling how a human might move a mouse cursor from `start` to `end`. It is a port from arevi/wind-mouse.

Value

A data frame with columns:

- `x`: X coordinate of the cursor.
- `y`: Y coordinate of the cursor.
- `t`: Cumulative time in milliseconds since the start.

See Also

Other path: path_clothoid()

Examples

```
## Not run:
path <- path_mouse(start = c(0, 0), end = c(300, 200), seed = 123)
with(path, plot(x, y, type = "l", asp = 1, main = "WindMouse Path"))

## End(Not run)
```

---

reorder *Reorder objects by cyclic shifts or scanning patterns*

Description

A collection of functions for reordering one-dimensional vectors, native rasters, or the rows of data frames by simple index manipulations.

These operations are designed for creative coding workflows, where controlling the traversal order of grids or sequences is useful for producing structured, semi-regular visual variations.

Usage

```
rings_index(nrow, ncol, byrow = FALSE)

rings(x, nrow = NULL, ncol = NULL, byrow = FALSE)

## Default S3 method:
rings(x, nrow = NULL, ncol = NULL, byrow = FALSE)

## S3 method for class 'data.frame'
rings(x, nrow = NULL, ncol = NULL, byrow = FALSE)

## S3 method for class 'nativeRaster'
rings(x, nrow = NULL, ncol = NULL, byrow = FALSE)

shift(x, k)

## Default S3 method:
shift(x, k)

## S3 method for class 'data.frame'
shift(x, k)

## S3 method for class 'nativeRaster'
shift(x, k)
```

```
snake_index(nrow, ncol, byrow = FALSE)

snake(x, nrow = NULL, ncol = NULL, byrow = FALSE)

## Default S3 method:
snake(x, nrow = NULL, ncol = NULL, byrow = FALSE)

## S3 method for class 'data.frame'
snake(x, nrow = NULL, ncol = NULL, byrow = FALSE)

## S3 method for class 'nativeRaster'
snake(x, nrow = NULL, ncol = NULL, byrow = FALSE)

spiral_index(nrow, ncol, byrow = FALSE)

spiral(x, nrow = NULL, ncol = NULL, byrow = FALSE)

## Default S3 method:
spiral(x, nrow = NULL, ncol = NULL, byrow = FALSE)

## S3 method for class 'data.frame'
spiral(x, nrow = NULL, ncol = NULL, byrow = FALSE)

## S3 method for class 'nativeRaster'
spiral(x, nrow = NULL, ncol = NULL, byrow = FALSE)

stride_index(n, step = 2L)

stride(x, step = 2L)

## Default S3 method:
stride(x, step = 2L)

## S3 method for class 'data.frame'
stride(x, step = 2L)

## S3 method for class 'nativeRaster'
stride(x, step = 2L)

zigzag_index(nrow, ncol, byrow = FALSE)

zigzag(x, nrow = NULL, ncol = NULL, byrow = FALSE)

## Default S3 method:
zigzag(x, nrow = NULL, ncol = NULL, byrow = FALSE)

## S3 method for class 'data.frame'
zigzag(x, nrow = NULL, ncol = NULL, byrow = FALSE)
```

```
## S3 method for class 'nativeRaster'
zigzag(x, nrow = NULL, ncol = NULL, byrow = FALSE)
```

Arguments

nrow, ncol	Integers giving the conceptual number of rows and columns for grid-based scanning patterns. If one of them is NULL, it is inferred from the length of x.
byrow	Logical; whether to interpret the implicit grid in row-major (TRUE) or column-major (FALSE) order. For native rasters, byrow is interpreted as !byrow.
x	A vector or data frame to be reordered.
k	An integer shift amount for shift().
n	An integer scalar; the length of the index vector to generate.
step	Integer stride size for stride() and stride_index().

Details

The following methods are available:

- rings(x, nrow, ncol): Reorders x or the rows of a data frame by scanning the grid in *concentric rings*: the outer border first, followed by progressively inner borders, until the center is reached.
- rings_index(nrow, ncol): Returns the index vector that enumerates the cells of a grid in concentric rectangular rings from the outside inward.
- shift(x, k): Performs a circular shift of the vector x by k positions. For data frames, rows are shifted instead of elements.
- snake(x, nrow, ncol): Reorders x (or rows of a data frame) according to a *snake* (serpentine) traversal of a conceptual grid of size nrow * ncol. Odd-numbered rows are read left-to-right, even rows right-to-left.
- snake_index(nrow, ncol): Returns only the index vector representing the snake traversal.
- spiral(x, nrow, ncol): Reorders x or a data frame by following a spiral scan of a grid, starting from the outermost ring toward the center.
- spiral_index(nrow, ncol): Computes the spiral traversal indices.
- stride(x, step): Reorders x by interleaving elements with a stride of length step.
- stride_index(n, step): Returns the index vector used by stride().
- zigzag(x, nrow, ncol): Reorders x (or rows of a data frame) according to a diagonal zigzag traversal of a conceptual nrow * ncol grid. Each diagonal where i + j is constant is processed together, with the direction alternating between forward and reverse.
- zigzag_index(nrow, ncol): Returns the index vector for a zigzag traversal of a grid, following alternating diagonal scanlines (similar to JPEG DCT block ordering).

These functions are lightweight and composable, and are intended for preprocessing sequences or point grids in creative visualization workflows.

Value

A reordered object of the same type as the input (`x`), or an integer vector of indices for functions ending in `_index`.

`seq_ease` *Interpolate between two values*

Description

Derived from coolbutuseless/displease.

Usage

```
seq_ease(x1, x2, n = 100, ease = function(t) ease_in_out(t, "cubic"))

seq_color(
  col1,
  col2,
  n = 100,
  ease = function(t) ease_in_out(t, "cubic"),
  colorspace = "hcl"
)
```

Arguments

`x1`, `x2`	Numeric scalars.
`n`	`length.out` parameter for `base::seq()`.
`ease`	A function to ease the sequence.
`col1`, `col2`	Character scalars. The colors to interpolate between.
`colorspace`	A string. Color space in which to do the interpolation. See `farver::convert_colour()` for details.

Value

- For `seq_ease()`, a numeric vector.
- For `seq_color()`, a character vector.

`smoothstep` *Smoothing functions*

Description

Smoothing functions

Usage

```
smoothstep(t)

smootherstep(t)

ease_bezier(t, x1, y1, x2, y2)

ease_in(
  t,
  type = c("sine", "quad", "cubic", "quart", "quint", "exp", "circle", "elastic", "back",
    "bounce")
)

ease_out(
  t,
  type = c("sine", "quad", "cubic", "quart", "quint", "exp", "circle", "elastic", "back",
    "bounce")
)

ease_in_out(
  t,
  type = c("sine", "quad", "cubic", "quart", "quint", "exp", "circle", "elastic", "back",
    "bounce")
)
```

Arguments

`t`	A numeric vector.
`x1`, `y1`, `x2`, `y2`	Numeric scalars for bezier easing.
`type`	A string. The type of easing function to use.

Value

A numeric vector.

sph_harm *Evaluate real spherical harmonics*

Description

Evaluates the real spherical harmonic basis function of degree `l` and order `m`.

`x` must be either:

- a matrix with 2 columns giving spherical coordinates `theta` and `phi`, or
- a matrix with 3 columns giving Cartesian direction vectors `x`, `y`, and `z`.

When `x` has 3 columns, each row is interpreted as a direction vector. Direction vectors should typically be normalized to unit length before evaluation.

Usage

```
sph_harm(x, l, m)
```

Arguments

`x`	A numeric matrix containing spherical coordinates or direction vectors.
`l`	Integer scalar giving the degree of the spherical harmonic.
`m`	Integer scalar giving the order of the spherical harmonic.

Details

For spherical coordinates, the first column is interpreted as `theta` and the second column as `phi`.

For Cartesian coordinates, rows are interpreted as direction vectors `(x, y, z)`.

Value

A numeric vector containing the evaluated real spherical harmonic values.

state_bouncing_pts *Generate a state for bouncing points*

Description

Generates a chakra state for bouncing points.

Usage

```
state_bouncing_pts(
  seeds,
  bbox = c(-1, -1, 1, 1),
  restitution = 1,
  x = x,
  y = y,
  vx = vx,
  vy = vy
)
```

Arguments

`seeds`	A data frame containing the initial point positions.
`bbox`	Numeric vector of length 4; The bounding box of the bouncing points as `c(xmin, ymin, xmax, ymax)`.
`restitution`	Numeric scalar; The restitution of the bouncing points.
`x`, `y`, `vx`, `vy`	<data-masking> Expressions specifying data columns for the initial state.

Value

An external pointer.

See Also

chakra

Other state: `state_orbits()`

Examples

```
seeds <-
 dplyr::tibble(
   x = runif(10, -1, 1),
   y = runif(10, -1, 1),
   angle = runif(10, -pi, pi),
   vx = cos(angle),
   vy = sin(angle)
 )

state <- state_bouncing_pts(seeds)

observe(state)

circulate(state) |>
 observe()
```

`state_orbits` *Create a state for orbital motion*

Description

Creates a pseudo-chakra state in which points revolve around a common origin with constant angular velocity. Initial phases and orbital radii are inferred from the input coordinates.

Usage

```
state_orbits(
  seeds,
  origin = c(0, 0),
  x = x,
  y = y,
  omega = 1,
  id = dplyr::row_number()
)
```

Arguments

`seeds`	A data frame containing the initial point positions.
`origin`	Numeric vector of length 2 giving the center of rotation.
`x`, `y`	<`data-masking`> Coordinates used as the initial point positions.
`omega`	<`data-masking`> Angular velocity. May be either a scalar or a column evaluated in `seeds`.
`id`	<`data-masking`> Identifier for each orbit. Defaults to row numbers.

Details

The initial positions are interpreted as points on circular orbits around `origin`. For each point, the orbital radius and initial phase are computed internally from the supplied coordinates.

The resulting state stores:

- orbital radius
- initial phase
- angular velocity
- current time

Coordinates at the current time can be retrieved with `observe()`.

Value

A list.

See Also

chakra

Other state: state_bouncing_pts()

Examples

```
seeds <-
  curve_epicycloid(16) |>
  dplyr::mutate(
    omega = seq(-1, 1, length.out = dplyr::n())
  )

state <-
  state_orbits(
    seeds,
    x = x,
    y = y,
    omega = omega
  )

observe(state)

state |>
  circulate(dt = pi / 4) |>
  observe()
```

`trace_flow` *Trace trajectories through a flow*

Description

Generates trajectories from a set of seed points using a user-supplied *flow function*. Each seed is passed to `flow_fn`, which is responsible for producing a trajectory of length `n_steps`.

Usage

```
trace_flow(
  seeds,
  flow_fn,
  x = x,
  y = y,
  id = dplyr::row_number(),
  n_steps = 10,
  step_size = 1,
  ...
)
```

Arguments

`seeds`	A data frame containing seed points.
`flow_fn`	A function that generates trajectories. It must have the signature `function(seed, n_steps, st` where: • `seed` is a data frame containing the rows corresponding to a single seed group, • `n_steps` is an integer giving the number of steps, • `step_size` is a numeric scalar controlling step size, and • `params` is a list of additional parameters. The function should return a data frame at least with columns `step`, `x`, and `y`.
`x, y`	<data-masking> Expressions specifying the x and y coordinates of the seeds.
`id`	<data-masking> Expression specifying grouping of seeds. Each group is treated as a single seed and passed to `flow_fn`. Defaults to `dplyr::row_number()`.
`n_steps`	Integer scalar giving the number of steps in each trajectory.
`step_size`	Numeric scalar controlling the step size passed to `flow_fn`.
`...`	Additional parameters passed to `flow_fn`. These are collected into a list and supplied as the `params` argument.

Details

`trace_flow()` applies `flow_fn` independently to each seed group. The grouping is defined by the `id` argument. For each group, `flow_fn` is called once and is expected to return a full trajectory.

The design places full control of trajectory generation in `flow_fn`, allowing implementations that are vectorized, iterative, or backed by compiled code. This makes it possible to express a wide range of flow-like behaviors, including vector fields, noise-driven updates, and custom dynamical systems.

Evaluation of seeds is parallelized via `purrr::in_parallel()`.

Value

A tibble with columns:

- `seed`: integer identifier for each seed group,
- `step`: step index within each trajectory,
- `x`, `y`: coordinates of the trajectory.

Examples

```
# Simple flow: constant drift
flow_fn <- function(seed, n_steps, step_size, params) {
  x <- numeric(n_steps)
  y <- numeric(n_steps)
```

```
  cur <- as.matrix(seed[, c("x", "y"), drop = FALSE])
  for (i in seq_len(n_steps)) {
    x[i] <- cur[, 1]
    y[i] <- cur[, 2]
    cur <- cur + step_size * c(0.1, 0)
  }
  data.frame(step = seq_len(n_steps), x = x, y = y)
}

seeds <- data.frame(
  x = runif(5),
  y = runif(5)
)

trace_flow(seeds, flow_fn, x = x, y = y, n_steps = 20, step_size = 0.05)

# Using additional parameters
flow_noise <- function(seed, n_steps, step_size, params) {
  x <- numeric(n_steps)
  y <- numeric(n_steps)
  cur <- as.matrix(seed[, c("x", "y"), drop = FALSE])
  for (i in seq_len(n_steps)) {
    x[i] <- cur[, 1]
    y[i] <- cur[, 2]
    cur[, 1] <- cur[, 1] +
      params$nx(cur[, 1], cur[, 2]) * step_size
    cur[, 2] <- cur[, 2] +
      params$ny(cur[, 1], cur[, 2]) * step_size
  }
  data.frame(step = seq_len(n_steps), x = x, y = y)
}

trace_flow(
  seeds,
  flow_noise,
  x = x,
  y = y,
  n_steps = 50,
  step_size = 0.02,
  nx = function(x, y) sin(x + y),
  ny = function(x, y) cos(x - y)
)
```

Index