Package ‘vectorialcalculus’

January 19, 2026

Type Package

Title Vector Calculus Tools for Visualization and Analysis

Version 1.0.5

Maintainer Julian Mauricio Fajardo <julian.fajardo1908@gmail.com>

Description Provides pedagogical tools for visualization and numerical computation in vector calculus. Includes functions for parametric curves, scalar and vector fields, gradients, divergences, curls, line and surface integrals, and dynamic 2D/3D graphical analysis to support teaching and learning. The implemented methods follow standard treatments in vector calculus and multivariable analysis as presented in Marsden and Tromba (2011) <ISBN:9781429215084>, Stewart (2015) <ISBN:9781285741550>, Thomas, Weir and Hass (2018) <ISBN:9780134438986>, Larson and Edwards (2016) <ISBN:9781285255869>, Apostol (1969) <ISBN:9780471000051>, Spivak (1971) <ISBN:9780805390216>, Schey (2005) <ISBN:9780071369080>, Colley (2019) <ISBN:9780321982384>, Lizarazo Osorio (2020) <ISBN:9789585450103>, Sievert (2020) <ISBN:9780367180165>, and Borowko (2013) <ISBN:9781439870791>.

License MIT + file LICENSE

URL https://github.com/JulianFajardo1908/vectorialcalculus

BugReports https://github.com/JulianFajardo1908/vectorialcalculus/issues

Depends R (>= 4.1.0)

Imports tibble, stats, utils

Suggests plotly, pracma, ggplot2, grDevices, purrr, testthat (>= 3.0.0)

Config/testthat/edition 3

Encoding UTF-8

Language en

RoxygenNote 7.3.3

NeedsCompilation no

Author Julio Lizarazo Osorio [aut],
Julian Mauricio Fajardo [aut, cre]

Repository CRAN

Date/Publication 2026-01-19 17:50:02 UTC

Contents

`arc_length3d` *Numeric arc length of a 3D parametric curve*

Description

Computes a numerical approximation to the arc length of the parametric curve $(X(t), Y(t), Z(t))$ on the interval $[a, b]$ by integrating the speed $\sqrt{(dx/dt)^2 + (dy/dt)^2 + (dz/dt)^2}$.

Usage

```
arc_length3d(
  X,
  Y,
  Z,
  a,
  b,
  h = 1e-06,
  method_int = c("romberg", "integrate"),
  n_samples = 400,
  plot = FALSE,
  plot_mode = "lines",
  plot_line = list(color = "blue", width = 3, dash = "solid"),
  plot_marker = NULL,
  plot_title = NULL,
 plot_scene = list(xaxis = list(title = "x(t)"), yaxis = list(title = "y(t)"), zaxis =
    list(title = "z(t)")),
  plot_bg = list(paper = "white", plot = "white")
)
```

Arguments

`X, Y, Z`	Functions of one variable `t` defining the parametric curve coordinates.
`a, b`	Numeric parameter limits for `t`.
`h`	Numeric step size for centered finite differences used to approximate the derivatives dX/dt, dY/dt, and dZ/dt.
`method_int`	Character string. Either `"romberg"` (requires the **pracma** package) or `"integrate"` (base R).
`n_samples`	Integer. Number of sample points used when plotting the curve (if `plot = TRUE`).
`plot`	Logical. If `TRUE`, the function also produces a 3D visualization of the curve using `plot_curve3d()`.
`plot_mode`	Character string passed to `plot_curve3d()` as the `mode` argument.
`plot_line`	List with line styling options passed to `plot_curve3d()`.

plot_marker	Optional list with marker styling options passed to plot_curve3d(), or NULL.
plot_title	Optional title for the plot. If NULL, a title including the estimated arc length is generated.
plot_scene	List specifying 3D axes and options, passed to plot_curve3d().
plot_bg	List with background colors, passed to plot_curve3d().

Details

Derivatives are approximated by centered finite differences and the integral is computed either by Romberg integration (via **pracma**) or by integrate from base R. Optionally, the curve can be visualized with plot_curve3d().

Value

A single numeric value: the approximated arc length of the curve on the interval $[a, b]$.

See Also

curve_sample3d(), plot_curve3d()

Examples

```
X <- function(t) t^2 * cos(t)
Y <- function(t) t^3 * sin(3 * t)
Z <- function(t) t
arc_length3d(X, Y, Z, 0, 2 * pi)

# \donttest{
# if (requireNamespace("plotly", quietly = TRUE)) {
#   arc_length3d(
#     X, Y, Z, 0, 2 * pi,
#     plot = TRUE,
#     plot_line = list(color = "red", width = 3),
#     n_samples = 300
#   )
# }
# }
```

binormal3d   *Binormal vectors along a 3D parametric curve*

Description

Computes numerical binormal vectors of a three-dimensional parametric curve at selected parameter values. The curve is given by three coordinate functions. At each evaluation point, the first and second derivatives of the curve are approximated numerically, and their cross-product direction is normalized to obtain the binormal vector.

Usage

```
binormal3d(
  X,
  Y,
  Z,
  a,
  b,
  t_points,
  h = 1e-04,
  plot = FALSE,
  n_samples = 400,
  vec_scale = NULL,
  vec_factor = 1,
  curve_line = list(color = "blue", width = 2, dash = "solid"),
  B_line = list(color = "black", width = 5, dash = "solid"),
  show_curve = TRUE,
  show_points = TRUE,
  point_marker = list(color = "black", size = 3, symbol = "circle"),
 scene = list(aspectmode = "data", xaxis = list(title = "x(t)"), yaxis = list(title =
    "y(t)"), zaxis = list(title = "z(t)")),
  bg = list(paper = "white", plot = "white"),
  tol = 1e-10
)
```

Arguments

`X`	Function returning the `x` coordinate of the curve as a function of the parameter `t`.
`Y`	Function returning the `y` coordinate of the curve as a function of the parameter `t`.
`Z`	Function returning the `z` coordinate of the curve as a function of the parameter `t`.
`a`	Lower endpoint of the parameter interval.
`b`	Upper endpoint of the parameter interval.
`t_points`	Numeric vector of parameter values for evaluation and optional plotting.
`h`	Step size for centered finite-difference approximations.
`plot`	Logical; if `TRUE`, produces a **plotly** 3D visualization showing the curve and the binormal vectors.
`n_samples`	Number of points used to sample and display the curve in the plot.
`vec_scale`	Base length used for binormal segments. If `NULL`, it is estimated as a small proportion of the size of the sampled curve.
`vec_factor`	Multiplicative factor applied to `vec_scale` to adjust segment length.
`curve_line`	List with **plotly** style options for drawing the base curve.
`B_line`	List with **plotly** style options for the binormal segments.

show_curve	Logical; if TRUE, the base curve is included in the plot.
show_points	Logical; if TRUE, the evaluation points are marked in the plot.
point_marker	List with **plotly** marker options for the evaluation points.
scene	List with 3D scene settings for the **plotly** figure.
bg	Background color configuration for the **plotly** figure.
tol	Numeric tolerance for detecting situations in which the derivative information is too weak to determine a stable binormal direction.

Details

For every value in t_points, the function:

- computes centered finite-difference approximations of the first and second derivatives,
- forms a direction perpendicular to both derivatives,
- normalizes that direction to obtain a unit binormal vector.

If the first derivative is extremely small or if the first and second derivative vectors are nearly parallel, the binormal direction cannot be determined reliably. In these cases, the function returns NA for the affected components.

Optionally, the function can display the curve and the associated binormal segments in an interactive 3D plot using **plotly**. The sampled curve, evaluation points and binormal segments can be shown or hidden independently.

Value

A tibble with columns t, x, y, z, Bx, By and Bz, containing the components of the binormal vector at each evaluation point.

Examples

```
X <- function(t) t * cos(t)
Y <- function(t) t * sin(3 * t)
Z <- function(t) t
binormal3d(X, Y, Z, a = 0, b = 2 * pi, t_points = c(pi / 3, pi, 5 * pi / 3))
```

---

critical_points_2d *Critical points of a two-variable function using gradient and Hessian*

Description

Identifies stationary points of a function of two variables over a given rectangular domain. A set of initial points is generated on a regular grid together with additional random starting points. Each start is refined using numerical optimization applied to the squared gradient norm. Points with a sufficiently small gradient norm are kept and then merged if they are too close to each other.

Usage

```
critical_points_2d(
  f,
  xlim,
  ylim,
  start_n = c(7L, 7L),
  n_rand = 40L,
  h = NULL,
  tol_grad = 1e-06,
  tol_merge = 0.001,
  tol_eig = 1e-06,
  maxit = 200,
  optim_method = c("BFGS", "Nelder-Mead"),
  plot = TRUE,
  grid_plot = c(60L, 60L),
  surface_colorscale = "YlGnBu",
  surface_opacity = 0.85,
 cp_colors = list(minimum = "#2ca02c", maximum = "#d62728", saddle = "#1f77b4", flat =
    "#ff7f0e"),
  cp_size = 6,
 scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
    zaxis = list(title = "f(x,y)"))
)
```

Arguments

`f`	Function of two variables `f(x, y)` returning a numeric scalar.
`xlim`	Numeric vector `c(xmin, xmax)` defining the domain in the x-direction.
`ylim`	Numeric vector `c(ymin, ymax)` defining the domain in the y-direction.
`start_n`	Integer vector of length two. Number of regular starting points per axis for the grid.
`n_rand`	Integer. Number of additional random starting points inside the domain.
`h`	Numeric step size for finite-difference gradients and Hessians. If `NULL`, an automatic step size is used.
`tol_grad`	Numeric tolerance. Points with gradient norm below this threshold are treated as stationary.
`tol_merge`	Numeric tolerance for merging nearby solutions.
`tol_eig`	Numeric tolerance used when classifying eigenvalues of the Hessian.
`maxit`	Maximum number of iterations permitted for numerical optimization.
`optim_method`	Character string naming an optimization method supported by `stats::optim`, such as `"BFGS"` or `"Nelder-Mead"`.
`plot`	Logical. If `TRUE`, a **plotly** surface with critical points is drawn.
`grid_plot`	Integer vector of length two. Resolution of the grid used when drawing the surface.

`surface_colorscale`	Character. Name of the Plotly colorscale for the surface.
`surface_opacity`	Numeric between 0 and 1 giving the opacity of the surface.
`cp_colors`	Named list mapping each critical point type to a color. Expected names: `"minimum"`, `"maximum"`, `"saddle"`, `"flat"`.
`cp_size`	Numeric. Size of the point markers.
`scene`	List of Plotly scene options (axis titles, aspect mode, and so on).

Details

Each surviving point is classified based on the eigenvalues of the numerical Hessian matrix. The Hessian classification uses four categories:

- `"minimum"` - both eigenvalues positive,
- `"maximum"` - both eigenvalues negative,
- `"saddle"` - mixed signs,
- `"flat"` - small eigenvalues, inconclusive classification.

Optionally, a 3D surface with the detected critical points can be displayed using **plotly**.

Value

A list with:

`critical_points` A data frame with columns `x`, `y`, `z`, the gradient norm, and the classification label.

`fig` A **plotly** object if `plot = TRUE`, or `NULL` otherwise.

Examples

```
f <- function(x, y) x^2 + y^2
critical_points_2d(f, xlim = c(-2, 2), ylim = c(-2, 2), plot = FALSE)
```

`critical_points_nd` *Critical points of a scalar field in n dimensions (no plot)*

Description

Searches for approximate critical points of a scalar field `f(x)` in dimension `n >= 3` over a rectangular domain. The algorithm looks for points where the gradient is close to zero by minimizing the squared gradient norm `g(x) = ||grad f(x)||^2` from multiple starting points.

Usage

```
critical_points_nd(
  f,
  bounds,
  start_grid = NULL,
  n_random = 50L,
  max_grid_starts = 2000L,
  h = NULL,
  tol_grad = 1e-06,
  tol_merge = 0.001,
  tol_eig = 1e-06,
  maxit = 200,
  optim_method = c("BFGS", "Nelder-Mead"),
  seed = NULL,
  store_hessian = FALSE
)
```

Arguments

`f`	Scalar field as `function(x)`, where `x` is a numeric vector of length `n`, returning a numeric scalar.
`bounds`	Domain bounds. Either: • an `n x 2` numeric matrix or data frame with columns `lower` and `upper`, or • a list of length `n`, each element a numeric vector `c(lower, upper)`.
`start_grid`	Integer vector of length `n` with the number of regular grid points per dimension used as deterministic starting points. If `NULL`, a default `rep(5, n)` is used. The total number of grid points is limited by `max_grid_starts`.
`n_random`	Integer. Number of additional random starting points sampled uniformly inside the domain defined by `bounds`.
`max_grid_starts`	Maximum number of deterministic grid starting points that are actually used. If the full grid would exceed this value, a random subset of that size is taken.
`h`	Step size for finite differences. Can be: • `NULL`: automatic componentwise step size `1e-4 * (1 + abs(x_i))`, • a numeric scalar: same step for all coordinates, • a numeric vector of length `n`: one step per coordinate.
`tol_grad`	Numeric threshold on the gradient norm used to accept a point as critical. Smaller values make the criterion more strict.
`tol_merge`	Numeric radius used to merge nearby critical point candidates (Euclidean distance).
`tol_eig`	Numeric tolerance used to decide whether Hessian eigenvalues are treated as positive, negative or close to zero for classification.
`maxit`	Maximum number of iterations allowed for each call to `stats::optim()`.

`optim_method`	Primary optimization method passed to `stats::optim()`, for example `"BFGS"` or `"Nelder-Mead"`. If the primary method fails with an error, the function falls back to `"Nelder-Mead"`.
`seed`	Optional integer seed for reproducibility of the random starting points.
`store_hessian`	Logical. If `TRUE`, the Hessian matrix at each critical point is stored and returned.

Details

Candidate points that are closer than `tol_merge` (in Euclidean distance) are merged into a single representative. Each accepted point is classified by the eigenvalues of the numerical Hessian as `"minimum"`, `"maximum"`, `"saddle"` or `"flat"`.

Gradients and Hessians are computed with second-order central finite differences.

Value

A list with components:

`critical_points` A data frame with columns `x1, ..., xn`, the function value `f`, the gradient norm `grad_norm`, and the classification label `class`.

`eigvals` A list of numeric vectors containing the Hessian eigenvalues for each critical point.

`hessians` If `store_hessian = TRUE`, a list of Hessian matrices (one per critical point). Otherwise `NULL`.

`starts_info` A list with information about the number of grid and random starting points actually used.

Examples

```
# Example 1: unique minimum at (1, 1, 1)
f1 <- function(x) sum((x - 1)^2)
B  <- rbind(c(-2, 3), c(-2, 3), c(-2, 3))  # 3D bounds
res1 <- critical_points_nd(
  f1,
  bounds      = B,
  start_grid  = c(5, 5, 5),
  n_random    = 50,
  seed        = 1
)
res1$critical_points

# Example 2: saddle at the origin in 3D
f2 <- function(x) x[1]^2 + x[2]^2 - x[3]^2
B2 <- rbind(c(-1, 1), c(-1, 1), c(-1, 1))
res2 <- critical_points_nd(
  f2,
  bounds      = B2,
  start_grid  = c(5, 5, 5),
  n_random    = 30,
  seed        = 123
)
res2$critical_points
```

```
# Example 3 (4D): multiple critical points
f3 <- function(x) sum(x^4 - 2 * x^2)
B3 <- do.call(rbind, replicate(4, c(-2, 2), simplify = FALSE))
res3 <- critical_points_nd(
  f3,
  bounds      = B3,
  start_grid  = rep(4, 4),
  n_random    = 200,
  seed        = 42
)
head(res3$critical_points)
```

`curl3d` *Numerical curl of a three-dimensional vector field*

Description

Computes the curl of a vector field in three dimensions at a given point, using second-order central finite differences. The field may optionally depend on a time parameter; if so, the curl is evaluated at a fixed time value.

Usage

```
curl3d(field, x, y, z, h = NULL, tval = 0, method = c("central"))
```

Arguments

`field`	A function representing the vector field. It can be defined as `function(x, y, z)` or as `function(x, y, z, t)`. It must return a numeric vector `c(Fx, Fy, Fz)`.
`x, y, z`	Numeric scalars giving the coordinates of the evaluation point.
`h`	Step size for finite differences. It may be: • a single numeric value, • a numeric vector of length three, • or `NULL` (automatic selection).
`tval`	Time value used when the vector field depends on time. Default is `0`.
`method`	Differencing scheme. Currently only `"central"` is supported.

Details

The vector field must be a function that returns a numeric vector of length three representing the components of the field at a point. The curl is obtained by approximating the partial derivatives of the field components with respect to each coordinate direction using symmetric finite differences.

The step size for each coordinate can be:

- a single scalar used for all three axes,
- a numeric vector of length three providing separate steps for the x, y and z directions,
- or `NULL`, in which case automatic step sizes are chosen based on the evaluation point.

The method currently implemented is the second-order central differencing scheme. Smaller step sizes may provide more accurate results for rapidly varying fields, at the cost of increased sensitivity to floating-point error.

Value

A named numeric vector of length three containing the curl components at the evaluation point. The components are named `omega_x`, `omega_y` and `omega_z`.

Examples

```
# Simple rotating field: curl is constant in the third component
field1 <- function(x, y, z) c(-y, x, 0.6)
curl3d(field1, x = 0.1, y = -0.3, z = 2)

# Time-dependent example (time does not affect the curl):
field2 <- function(x, y, z, t) c(-y, x + t, z)
curl3d(field2, x = 1, y = 2, z = 3, tval = 5)

# Using a smaller step size for more precision:
curl3d(field1, x = 1, y = 1, z = 1, h = 1e-5)
```

`curvature_torsion3d` *Curvature and torsion of a 3D parametric curve*

Description

Computes numerical curvature and torsion of a three-dimensional parametric curve at a specific value of the parameter. The curve is described by three functions that give the coordinate components. All derivatives are approximated using centered finite differences of first, second and third order.

Usage

```
curvature_torsion3d(
  X,
  Y,
  Z,
  t0,
  h = 1e-04,
  plot = FALSE,
  window = 1,
  n_samples = 200,
  line = list(color = "red", width = 2, dash = "solid"),
  point = list(color = "black", size = 5, symbol = "circle"),
 scene = list(aspectmode = "data", xaxis = list(title = "x(t)"), yaxis = list(title =
   "y(t)"), zaxis = list(title = "z(t)")),
  bg = list(paper = "white", plot = "white"),
  tol = 1e-10
)
```

Arguments

X, Y, Z	Functions of t returning the three coordinate components of the parametric curve.
t0	Value of the parameter at which curvature and torsion are evaluated.
h	Step size for the finite-difference approximations. Smaller values give more accuracy but may amplify numerical noise.
plot	Logical; if TRUE, displays a 3D plot of a short segment of the curve around the evaluation point.
window	Length of the parameter interval shown when plot = TRUE. The interval is centered at t0.
n_samples	Number of points used to draw the curve segment in the 3D plot.
line	A list defining the visual style of the curve in the 3D plot.
point	A list defining the visual style of the marker placed at the evaluation point.
scene	A list with 3D axis settings for **plotly**.
bg	Background colors for the **plotly** figure.
tol	Numeric tolerance used to detect degenerate situations in which curvature or torsion cannot be reliably computed.

Details

The curvature at the evaluation point measures how sharply the curve bends at that location. It is computed from the first and second derivative vectors. The torsion measures how the curve deviates from being planar and is computed from the first, second and third derivative vectors. If the first derivative vector is nearly zero, or if the first and second derivative vectors are nearly parallel, the torsion becomes undefined; in such cases the function returns NA and provides a diagnostic message.

Optionally, the function can display a small segment of the curve around the evaluation point using **plotly**. The point where the curvature and torsion are computed is highlighted in the 3D plot.

Value

A list with:

- kappa: numerical curvature at the evaluation point.
- tau: numerical torsion at the evaluation point, or NA if the computation is unstable.
- t0: the parameter value where the evaluation was made.
- point: a numeric vector containing the coordinates of the curve at t0.
- r1, r2, r3: numeric approximations to the first, second and third derivative vectors at t0.

Examples

```
# Example curve
X <- function(t) t^2 * cos(t)
Y <- function(t) t^3 * sin(3 * t)
Z <- function(t) t
res <- curvature_torsion3d(X, Y, Z, t0 = pi)
res$kappa
res$tau

# \donttest{ if (requireNamespace("plotly", quietly = TRUE)) {
#   curvature_torsion3d(
#     X, Y, Z, t0 = pi, plot = TRUE,
#     window = 1.0, n_samples = 200,
#     line  = list(color = "red",  width = 2),
#     point = list(color = "black", size = 5, symbol = "circle")
#   )
# } }
```

`curve_sample3d` *Sample a 3D parametric curve*

Description

Generates a tibble with columns `t, x, y, z` by evaluating the parametric curve $(X(t), Y(t), Z(t))$ on the interval $[a, b]$ at a given number of sample points.

Usage

```
curve_sample3d(X, Y, Z, a, b, n_samples = 400)
```

Arguments

`X, Y, Z`	Functions of one variable `t`, e.g. `function(t) 2 * cos(t)`.
`a, b`	Numeric parameter limits for `t`.
`n_samples`	Integer. Number of sample points along the curve.

Value

A tibble with columns `t`, `x`, `y`, `z`, where `x = X(t)`, `y = Y(t)`, `z = Z(t)`.

See Also

`plot_curve3d()`, `arc_length3d()`

Examples

```
X <- function(t) 2 * cos(t)
Y <- function(t) 3 * sin(t)
Z <- function(t) t / 5
curve_sample3d(X, Y, Z, 0, 2 * pi, n_samples = 100)
```

`cylindrical_surface3d` *Ruled surface along a 3D parametric curve*

Description

Constructs a ruled surface generated by a three-dimensional parametric curve and a chosen direction field. At each sampled point on the curve, a straight segment is extended in a specified direction, producing a surface composed of line elements. Optionally, the resulting surface can be visualized using **plotly**.

Usage

```
cylindrical_surface3d(
  X,
  Y,
  Z,
  a,
  b,
  s_range,
  dir,
  n_t = 200,
  n_s = 60,
  plot = FALSE,
  surface_colorscale = "Blues",
  surface_opacity = 0.35,
  show_surface_grid = TRUE,
  surface_grid_color = "rgba(60,80,200,0.25)",
  surface_grid_width = 1,
  show_curve = TRUE,
  curve_line = list(color = "red", width = 2, dash = "solid"),
  show_edge_a = TRUE,
  show_edge_b = FALSE,
  edge_line = list(color = "blue", width = 2, dash = "solid"),
  show_rulings = TRUE,
  rulings_count = 12,
  rulings_at = NULL,
  rulings_line = list(color = "black", width = 1, dash = "solid"),
  show_axis_grid = FALSE,
 scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
   zaxis = list(title = "z")),
    bg = list(paper = "white", plot = "white"),
   lighting = list(ambient = 1, diffuse = 0.15, specular = 0, roughness = 1, fresnel = 0)
 )
```

Arguments

`X, Y, Z`	Functions of one variable `t` returning the coordinate components of the base curve.
`a, b`	Numeric values giving the endpoints of the parameter interval.
`s_range`	Numeric vector of length two giving the lower and upper bounds for the ruling parameter.
`dir`	Numeric vector of length three `c(ux, uy, uz)` giving the ruling direction. It will be normalized internally.
`n_t, n_s`	Integers giving the sampling resolution along the `t` and `s` directions.
`plot`	Logical; if `TRUE`, displays the ruled surface using **plotly**.
`surface_colorscale`	Character string or vector specifying the colorscale used for the surface.
`surface_opacity`	Numeric value between 0 and 1 controlling the opacity of the surface.
`show_surface_grid`	Logical; if `TRUE`, draws grid lines on the surface.
`surface_grid_color`	Color for the surface grid lines.
`surface_grid_width`	Numeric width for the grid lines.
`show_curve`	Logical; if `TRUE`, overlays the generating curve.
`curve_line`	List of **plotly** style options for the curve.
`show_edge_a, show_edge_b`	Logical; if `TRUE`, draws the boundary edges at the extremes of the ruling parameter.
`edge_line`	List of **plotly** style options for boundary edges.
`show_rulings`	Logical; if `TRUE`, draws a subset of rulings on the surface.
`rulings_count`	Integer giving the number of rulings to draw when `rulings_at` is not provided.
`rulings_at`	Optional numeric vector giving the parameter values at which rulings should be displayed.
`rulings_line`	List of **plotly** style options for displayed rulings.
`show_axis_grid`	Logical; if `TRUE`, shows axis gridlines in the 3D scene.
`scene`	Optional list with 3D scene settings for **plotly**.
`bg`	Optional list with background color settings for the figure.
`lighting`	Optional list with lighting parameters for surface shading in **plotly**.

Details

The function samples the base curve at `n_t` parameter values and, for each sampled point, generates a set of points along a line segment determined by the ruling parameter. These segments are interpolated over the interval specified in `s_range`.

In this implementation, the ruling direction is given by a fixed three-dimensional vector `dir`. This vector is normalized internally before constructing the surface.

If `plot = TRUE`, a 3D visualization is produced using **plotly**. The plot may include the ruled surface, grid lines on the surface, boundary edges corresponding to the extremes of the ruling parameter, and a selection of rulings. The generating curve can also be displayed.

Value

A list with:

- `t_seq`, `s_seq`: parameter grids used to build the mesh,
- `Xmat`, `Ymat`, `Zmat`: matrices of coordinates for the ruled surface,
- `curve`: data frame with the sampled generating curve,
- `edge_a`, `edge_b`: data frames for boundary edges (possibly `NULL` if not requested),
- `u_hat`: normalized ruling direction vector.

Examples

```
X <- function(t) cos(t)
Y <- function(t) sin(t)
Z <- function(t) 0.3 * t
dir_vec <- c(0, 0, 1)
rs <- cylindrical_surface3d(
  X, Y, Z,
  a = 0, b = 2 * pi,
  s_range = c(-0.2, 0.2),
  dir = dir_vec,
  n_t = 100, n_s = 40,
  plot = FALSE
)
```

---

directional_derivative3d

Directional derivative in any dimension, with optional 2D visualization

---

Description

Computes a numerical directional derivative of a multivariate function at a given point, along a specified direction. The derivative is approximated using centered finite differences. If the dimension is two and `plot = TRUE`, the function displays a local visualization of the surface defined by `z = f(x, y)`, including the evaluation point, the direction, and the curve traced along that direction.

Usage

```
directional_derivative3d(
  f,
  x0,
  v,
  h = 1e-06,
  plot = FALSE,
  x_window = 2,
  y_window = 2,
  n_s = 180,
  n_r = 50,
  show_strip = TRUE,
  strip_colorscale = "Blues",
  strip_opacity = 0.3,
  show_surface_grid = TRUE,
  surface_grid_color = "rgba(60,80,200,0.25)",
  surface_grid_width = 1,
  curve_line = list(color = "red", width = 1),
  point_marker = list(color = "black", size = 3, symbol = "circle"),
  u_line = list(color = "black", width = 1),
  w_line = list(color = "black", width = 0.5),
  t_range = c(-2, 2),
 scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
    zaxis = list(title = "z")),
  bg = list(paper = "white", plot = "white"),
  tol = 1e-12
)
```

Arguments

`f`	A function returning a numeric scalar. It may be defined either as `f(x, y, ...)` with several numeric arguments, or as a function of a numeric vector, `f(x_vec)`.
`x0`	Numeric vector giving the evaluation point. Its length determines the dimension of the problem.
`v`	Numeric vector giving the direction along which the directional derivative is computed. Must be nonzero and have the same length as `x0`.
`h`	Numeric step size used for centered finite-difference approximations.
`plot`	Logical; if `TRUE` and `length(x0) == 2`, displays a 3D visualization of the local surface using **plotly**.
`x_window`, `y_window`	Numeric half-widths defining the size of the rectangular window around the evaluation point in the 2D case.
`n_s`	Integer giving the number of samples along the direction line in the 2D visualization.
`n_r`	Integer giving the number of samples across the strip in the 2D visualization.
`show_strip`	Logical; if `TRUE`, draws a surface strip around the evaluation point.

`strip_colorscale`	Character string specifying a **plotly** colorscale for the strip.
`strip_opacity`	Numeric value between 0 and 1 determining the opacity of the strip.
`show_surface_grid`	Logical; if `TRUE`, overlays a grid onto the surface strip.
`surface_grid_color`	Character string giving the grid line color.
`surface_grid_width`	Numeric value giving the grid line width.
`curve_line`, `point_marker`, `u_line`, `w_line`	Lists with **plotly** style options for the directional curve, evaluation point, and auxiliary lines.
`t_range`	Numeric vector of length two giving the parameter range for the directional curve in the 2D visualization.
`scene`, `bg`	Lists specifying 3D scene and background options when plotting.
`tol`	Numeric tolerance used for detecting numerical degeneracies.

Details

The function accepts two types of input for the function `f`:

- a function of several numeric arguments, for example `f(x, y, z, ...)`, or
- a function that takes a single numeric vector, such as `f(x_vec)`.

At the evaluation point `x0`, the function:

- normalizes the direction vector `v` to obtain a unit direction,
- computes forward and backward perturbations along this unit direction,
- evaluates the function at those perturbed points,
- estimates the directional derivative using a centered finite-difference formula.

In two dimensions, if `plot = TRUE`, the function builds a small rectangular window around `x0` and evaluates the function over a fine grid to produce a strip of the surface. It then overlays:

- the base point,
- the selected direction,
- the trajectory of the directional curve,
- (optionally) a surface grid and other plot elements.

Value

A list containing:

`D`	The directional derivative at `x0` along the normalized direction.
`v_hat`	The normalized direction vector.
`fx`, `fy`	Centered partial derivatives in two dimensions (only returned when `length(x0) == 2`).
`fig`	A **plotly** visualization when `plot = TRUE`, otherwise `NULL`.

Examples

```
# General n-dimensional usage:
f3 <- function(x, y, z) x^2 + y^2 + z
directional_derivative3d(f3, x0 = c(1, 0, 0), v = c(1, 1, 0))

# Two-dimensional example without plotting (fast, no plotly required):
f2 <- function(x, y) x^2 + y^2
directional_derivative3d(f2, x0 = c(0, 0), v = c(1, 2), plot = FALSE)
```

`divergence_field` *Numerical divergence of a vector field*

Description

Computes the divergence of a vector field at a given point using central finite differences. The vector field `field` must take a numeric vector `x` and return a numeric vector of the same length.

Usage

```
divergence_field(field, x0, h = NULL, plot = FALSE)
```

Arguments

`field`	Function of the form `field(x)` returning a numeric vector of the same length as `x`.
`x0`	Numeric vector giving the evaluation point.
`h`	Step size for finite differences. Can be: • `NULL`: an automatic step is selected as `1e-4 * (1 + abs(x0))`; • A numeric scalar: same step for all components; • A numeric vector of the same length as `x0`.
`plot`	Logical; if `TRUE` and the dimension is 2 or 3, a basic visualization is drawn with **plotly**.

Details

Optionally, if the dimension is 2 or 3 and `plot = TRUE`, a simple visualization is produced using **plotly**.

Value

A numeric scalar: the divergence evaluated at `x0`.

Examples

```
field <- function(x) c(x[1] + x[2], x[2] - x[1])
divergence_field(field, c(0.5, -0.2))
```

frenet_frame3d *Frenet-Serret frame for a 3D parametric curve*

Description

Computes the Frenet-Serret frame, that is, the tangent, normal and binormal vectors of a three dimensional parametric curve at selected values of the parameter. The frame is obtained from numerical approximations of the first and second derivatives of the curve. Optionally, the curve and the three vector fields can be displayed in a 3D interactive visualization using **plotly**.

Usage

```
frenet_frame3d(
  X,
  Y,
  Z,
  a,
  b,
  t_points,
  h = 1e-04,
  plot = FALSE,
  n_samples = 400,
  vec_scale = NULL,
  curve_line = list(color = "blue", width = 2, dash = "solid"),
  T_line = list(color = "red", width = 4, dash = "solid"),
  N_line = list(color = "green", width = 4, dash = "solid"),
  B_line = list(color = "black", width = 4, dash = "solid"),
  show_curve = TRUE,
  show_points = TRUE,
  point_marker = list(color = "blue", size = 3, symbol = "circle"),
 scene = list(aspectmode = "data", xaxis = list(title = "x(t)"), yaxis = list(title =
    "y(t)"), zaxis = list(title = "z(t)")),
  bg = list(paper = "white", plot = "white"),
  tol = 1e-10
)
```

Arguments

X	Function returning the x coordinate of the curve as a function of the parameter t.
Y	Function returning the y coordinate of the curve.
Z	Function returning the z coordinate of the curve.
a	Lower endpoint of the parameter interval.
b	Upper endpoint of the parameter interval.
t_points	Numeric vector with the parameter values where the frame is computed and optionally plotted.

`h`	Step size for centered finite difference approximations.
`plot`	Logical; if `TRUE`, shows a 3D **plotly** visualization of the curve together with the three vector fields.
`n_samples`	Number of points used to sample the curve for plotting.
`vec_scale`	Base scaling factor for the vector segments. If `NULL`, it is estimated from the overall size of the sampled curve.
`curve_line`	Style options for drawing the base curve.
`T_line`	Style options for tangent vector segments.
`N_line`	Style options for normal vector segments.
`B_line`	Style options for binormal vector segments.
`show_curve`	Logical; if `TRUE`, the base curve appears in the plot.
`show_points`	Logical; if `TRUE`, the evaluation points are marked on the curve.
`point_marker`	Plotly marker style for the evaluation points.
`scene`	Plotly 3D scene configuration.
`bg`	Background settings for the **plotly** figure.
`tol`	Numeric tolerance used to detect degenerate derivative situations.

Details

At each parameter value in `t_points`, the function:

- computes finite difference approximations of the first and second derivatives of the curve,
- normalizes the first derivative to obtain the unit tangent direction,
- uses the first and second derivatives to construct a principal normal direction,
- constructs the binormal direction as a unit vector orthogonal to both the tangent and the normal,
- evaluates a numerical estimate of the curvature using the same derivative information.

If the derivative information is too small or nearly degenerate (for example, when the tangent direction cannot be reliably obtained), some components of the frame may be set to `NA`. The tolerance parameter `tol` controls how these situations are detected.

When `plot = TRUE`, the function displays:

- a sampled representation of the curve,
- the evaluation points,
- short line segments indicating the tangent, normal and binormal directions at each evaluation point.

All visual elements can be styled or shown selectively through the corresponding arguments.

Value

A tibble containing the parameter values and the coordinates of:

- the point on the curve,
- the tangent vector,
- the normal vector,
- the binormal vector,
- a numerical estimate of the curvature.

Columns are named `t`, `x`, `y`, `z`, `Tx`, `Ty`, `Tz`, `Nx`, `Ny`, `Nz`, `Bx`, `By`, `Bz`, `kappa`.

Examples

```
X <- function(t) t*cos(t)
Y <- function(t) t*sin(3*t)
Z <- function(t) t
frenet_frame3d(
  X, Y, Z, a = 0, b = 2*pi,
  t_points = c(pi/3, pi, 5*pi/3)
)
```

`gradient_direction2d` *Animate gradient and directional derivative on level curves (2D)*

Description

Produces a Plotly animation showing level curves of a scalar field together with the gradient direction at a point and a rotating unit direction vector. The directional derivative value is displayed on screen for each frame. A highlight is shown when the rotating direction aligns with the gradient, which corresponds to the maximum directional derivative.

Usage

```
gradient_direction2d(
  f,
  x0,
  y0,
  xlim,
  ylim,
  n_grid = 70L,
  theta_vals = NULL,
  h = 1e-04,
  arrow_scale = NULL,
  frame_ms = 220L,
  transition_ms = 220L,
  title = NULL,
  safe_mode = TRUE,
  align_tol = 0.08
)
```

Arguments

`f`	Function. A real-valued function f(x,y). It must accept two numeric arguments and return numeric values.
`x0`	Numeric scalar. x-coordinate of the base point.
`y0`	Numeric scalar. y-coordinate of the base point.
`xlim`	Numeric vector of length 2. Range for x in the contour plot.
`ylim`	Numeric vector of length 2. Range for y in the contour plot.
`n_grid`	Integer. Grid size per axis for the contour computation.
`theta_vals`	Numeric vector. Angles (radians) used as frames. If NULL, a default sequence from 0 to 2*pi is used.
`h`	Numeric scalar. Step size for central differences.
`arrow_scale`	Numeric scalar. Scale factor for drawing arrows. If NULL, an automatic scale based on the plot window is used.
`frame_ms`	Integer. Frame duration in milliseconds.
`transition_ms`	Integer. Transition duration in milliseconds.
`title`	Character. Plot title. If NULL, a default title is used.
`safe_mode`	Logical. If TRUE, use calmer animation defaults intended to reduce flicker and visual stress.
`align_tol`	Numeric scalar. Angular tolerance (radians) used to decide when the rotating direction is considered aligned with the gradient.

Details

The scalar field is

$$z = f(x, y).$$

At the point

$$(x_0, y_0),$$

the gradient vector is

$$\nabla f(x_0, y_0) = \left(\frac{\partial f}{\partial x}(x_0, y_0), \frac{\partial f}{\partial y}(x_0, y_0) \right).$$

For a unit direction

$$\mathbf{u}(\theta) = (\cos\theta, \sin\theta),$$

the directional derivative is

$$D_{\mathbf{u}} f(x_0, y_0) = \nabla f(x_0, y_0) \cdot \mathbf{u}(\theta).$$

The maximum value over unit directions is

$$\max_{\|\mathbf{u}\|=1} D_{\mathbf{u}} f(x_0, y_0) = \|\nabla f(x_0, y_0)\|,$$

and it occurs when

$$\mathbf{u}(\theta)$$

points in the same direction as

$$\nabla f(x_0, y_0).$$

Partial derivatives are approximated numerically by central differences.

Value

A plotly object (htmlwidget) with animation frames.

Examples

```
library(plotly)

f <- function(x, y) x^2 + 2*y^2
gradient_direction2d(
  f = f,
  x0 = 0.6,
  y0 = 0.4,
  xlim = c(-1.5, 1.5),
  ylim = c(-1.5, 1.5),
  safe_mode = TRUE,
  align_tol = 0.06
)
```

`gradient_scalar` *Gradient of a scalar field in R^n*

Description

Computes a numerical approximation of the gradient of a scalar function at a given point using central finite differences. The function `f` is assumed to take a numeric vector as input and return a scalar.

Usage

```
gradient_scalar(f, x0, h = NULL, plot = FALSE)
```

Arguments

`f`	Function of a numeric vector `f(x)` returning a numeric scalar.
`x0`	Numeric vector giving the evaluation point.
`h`	Numeric step size for finite differences. Can be: • `NULL` (default): a step is chosen as `1e-4 * (1 + abs(x0))` for each component;

- A scalar, used for all components;
- A numeric vector of the same length as `x0`.

`plot`	Logical; if `TRUE` and `length(x0)` is 2 or 3, draws the gradient vector with **plotly**.

Details

Optionally, if the input point has length 2 or 3 and `plot = TRUE`, a simple visualization of the gradient vector is produced using **plotly**.

Value

A numeric vector of the same length as `x0` with the components of the gradient.

Examples

```
f <- function(v) exp(-(v[1]^2 + v[2]^2)) + 0.3 * sin(2 * v[1] * v[2])
gradient_scalar(f, c(0.6, -0.4))
```

`integrate_double_polar`

Numerical Double Integration in Polar Coordinates

Description

Calculates the definite double integral of a function f(x, y) over a polar region R, defined by constant limits for theta and functional limits for the radius r, R: $g1(theta) <= r <= g2(theta), alpha <= theta <= beta$. The integration order used is r dr dtheta. Uses the Composite Simpson's Rule for numerical approximation.

Usage

```
integrate_double_polar(
  f,
  theta_min,
  theta_max,
  r_limit1,
  r_limit2,
  n_theta = 100,
  n_r = 100,
  plot_domain = TRUE
)
```

Arguments

`f`	A function R of two variables, `f(x, y)`, returning a numeric value (the original integrand).
`theta_min`	The constant lower limit for the outer integral (alpha).
`theta_max`	The constant upper limit for the outer integral (beta).
`r_limit1`	A function R of one variable defining the inner integral's lower limit (r = g1(theta)).
`r_limit2`	A function R of one variable defining the inner integral's upper limit (r = g2(theta)).
`n_theta`	Number of subintervals for the outer integration (theta). Must be even. Default is 100.
`n_r`	Number of subintervals for the inner integration (r). Must be even. Default is 100.
`plot_domain`	Logical. If TRUE, generates a ggplot2 plot of the integration domain in the Cartesian plane. Default is TRUE.

Value

A list containing:

- `integral_value`: The calculated approximation of the integral.
- `domain_plot`: The ggplot2 object representing the domain (if plot_domain = TRUE).

`integrate_double_xy` *Unified Numerical Double Integration*

Description

Calculates the definite double integral of a function f(x, y) over a general region D, which can be defined as Type I (dy dx) or Type II (dx dy). Uses the Composite Simpson's Rule for numerical approximation.

Usage

```
integrate_double_xy(
  f,
  const_min,
  const_max,
  limit1,
  limit2,
  region_type = "type1",
  n_outer = 100,
  n_inner = 100,
  plot_domain = TRUE
)
```

Arguments

`f`	A function in R of two variables, f(x, y), returning a numeric value.
`const_min`	The constant lower limit of the outer integration (a for Type I, c for Type II).
`const_max`	The constant upper limit of the outer integration (b for Type I, d for Type II).
`limit1`	A function in R of one variable defining the inner integral's lower limit (h1(x) or h1(y)).
`limit2`	A function in R of one variable defining the inner integral's upper limit (h2(x) or h2(y)).
`region_type`	A string specifying the region type: "type1" (dy dx) or "type2" (dx dy). Default is "type1".
`n_outer`	Number of subintervals for the outer integration. Must be even. Default is 100.
`n_inner`	Number of subintervals for the inner integration. Must be even. Default is 100.
`plot_domain`	Logical. If TRUE, generates a ggplot2 plot of the integration domain. Default is TRUE.

Value

A list containing:

- `integral_value`: The calculated approximation of the integral.
- `domain_plot`: The ggplot2 object representing the domain (if plot_domain = TRUE).

`integrate_triple_general` *Numerical Triple Integration over a General Region*

Description

Calculates the definite triple integral of a function f(x, y, z) over a general region W defined by the order dz dy dx. The region W is defined by constant limits for the outer integral (x), functional limits depending on x for the middle integral (y), and functional limits depending on both x and y for the inner integral (z). Uses the Composite Simpson's Rule for numerical approximation.

Usage

```
integrate_triple_general(
  f,
  x_min,
  x_max,
  y_limit1,
  y_limit2,
  z_limit1,
  z_limit2,
  n_outer = 50,
  n_middle = 50,
  n_inner = 50,
  plot_xy_domain = TRUE
)
```

Arguments

`f`	A function in R of three variables, `f(x, y, z)`, returning a numeric value.
`x_min`	The constant lower limit for the outermost integral (x = a).
`x_max`	The constant upper limit for the outermost integral (x = b).
`y_limit1`	A function in R of one variable defining the middle integral's lower limit (y = h1(x)).
`y_limit2`	A function in R of one variable defining the middle integral's upper limit (y = h2(x)).
`z_limit1`	A function in R of two variables defining the inner integral's lower limit (z = g1(x, y)).
`z_limit2`	A function in R of two variables defining the inner integral's upper limit (z = g2(x, y)).
`n_outer`	Number of subintervals for the outermost integral (x). Must be even. Default is 50.
`n_middle`	Number of subintervals for the middle integral (y). Must be even. Default is 50.
`n_inner`	Number of subintervals for the innermost integral (z). Must be even. Default is 50.
`plot_xy_domain`	Logical. If TRUE, generates a ggplot2 plot of the projection of the domain W onto the xy-plane. Default is TRUE.

Value

A list containing:

- `integral_value`: The calculated approximation of the integral.
- `domain_plot`: The ggplot2 object representing the xy-projection domain (if `plot_xy_domain = TRUE`).

`lagrange_check` *Optimality check with Lagrange multipliers and bordered Hessian*

Description

Evaluates first- and second-order optimality conditions for a smooth constrained optimization problem with equality constraints at a given candidate point. The function checks the Lagrange conditions, builds the bordered Hessian, and classifies the candidate as a minimum, maximum or indeterminate/saddle according to the signs of the leading principal minors.

Usage

```
lagrange_check(f, g, x, h = NULL, tol = 1e-06)
```

Arguments

`f`	Objective function. It must be `function(x)` and return a single numeric value. The argument `x` is a numeric vector of length `n`.
`g`	Equality constraints. Either a single function `function(x)` returning a numeric vector of length `m`, or a list of scalar functions `function(x)`, one per constraint.
`x`	Numeric vector giving the candidate point at which the optimality conditions are evaluated.
`h`	Step size for finite differences. It can be: • a single numeric value used for all coordinates, • a numeric vector of length `n` with one step per coordinate, • or `NULL`, in which case step sizes are chosen as `1e-4 * (1 + abs(x))` componentwise.
`tol`	Numeric tolerance used to judge feasibility of the constraints, the effective rank of the Jacobian, near singularity of matrices and very small principal minors.

Details

Consider a problem of minimizing or maximizing a scalar function `f(x)` subject to `m` equality constraints collected in `g(x) = 0`, where `x` is a vector in R^n and `g(x)` is a vector in R^m.

At the candidate point `x`, the function:

- Approximates the gradient of `f` and the gradients of each constraint using second-order central finite differences.
- Builds the Jacobian matrix `J` of the constraints (rows are gradients of each constraint).
- Approximates the Hessian matrix of `f` and the Hessian of each constraint, also by central finite differences.
- Forms the Hessian of the Lagrangian by combining the Hessian of `f` and the Hessians of the constraints with the Lagrange multipliers.
- Builds the bordered Hessian matrix using the Jacobian and the Hessian of the Lagrangian.
- Computes leading principal minors of the bordered Hessian and uses their signs to classify the candidate.

The classification is based on the standard bordered Hessian test: after multiplying each leading principal minor by (-1)^m, if all resulting values are positive the point is classified as a minimum; if their signs alternate (negative, positive, negative, and so on), the point is classified as a maximum. In any other case, the result is reported as indeterminate. All derivatives (gradients and Hessians) are computed numerically with central finite differences of second order. The step sizes can be given explicitly or chosen automatically from the scale of the point `x`.

Value

A list with components:

- `ok_stationarity`: numeric value of the norm of the stationarity residual. When constraints are present, this measures how close the gradient of `f` is to the linear combination given by the Jacobian transpose and the Lagrange multipliers.
- `ok_feasible`: maximum absolute value of the constraint vector `g(x)` at the candidate point.
- `lambda`: numeric vector of length `m` with the Lagrange multipliers.
- `J`: Jacobian matrix of the constraints at `x`, with dimension `m x n`.
- `H_f`: Hessian matrix of the objective function at `x`, of size `n x n`.
- `H_g`: list of Hessian matrices corresponding to each constraint function, each of size `n x n`.
- `H_L`: Hessian matrix of the Lagrangian at `x`.
- `B`: bordered Hessian matrix, of size `(m + n) x (m + n)` when constraints are present.
- `minors`: `data.frame` with one row per leading principal minor. It contains the order of the minor, its signed value and the logarithm of the absolute determinant used in the computation.
- `clasificacion`: character value equal to `"minimo"`, `"maximo"` or `"indeterminado"`, according to the bordered Hessian criterion.
- `notas`: character vector with diagnostic messages about the rank of the Jacobian, near singularity of matrices or any numerical issues detected.

Examples

```
## 1) Minimum with one constraint:
##    f(x,y) = x^2 + y^2,  g(x,y) = x + y - 1 = 0  -> (0.5, 0.5)
f1 <- function(x) x[1]^2 + x[2]^2
g1 <- function(x) x[1] + x[2] - 1
lagrange_check(f1, g1, x = c(0.5, 0.5))

## 2) Maximum with one constraint:
##    f(x,y) = -(x^2 + y^2),  g(x,y) = x + y - 1 = 0  -> (0.5, 0.5)
f2 <- function(x) -(x[1]^2 + x[2]^2)
lagrange_check(f2, g1, x = c(0.5, 0.5))

## 3) Two constraints in R^3 (minimum norm with two planes)
f3 <- function(x) sum(x^2)
g3 <- list(
  function(x) x[1] + x[2] + x[3] - 1,
  function(x) x[1] - x[3]
)
## Candidate solution: x1 = x3, 2*x1 + x2 = 1  ->  x = (1/3, 1/3, 1/3)
lagrange_check(f3, g3, x = c(1, 1, 1) / 3)
```

`line_integral2d` *Line integral of a scalar field along a planar curve, with optional 3D visualization*

Description

Computes a numerical line integral of a scalar field along a parametric curve in the plane. The function integrates the quantity `f(r(t)) * speed(t)`, where `r(t)` is the parametric curve and `speed(t)` is the length of its derivative. Optionally, it produces a 3D visualization that includes a surface representing `z = f(x, y)`, the curve itself in the plane, a lifted version of the curve showing `z = f(x(t), y(t))`, and a vertical curtain over the curve.

Usage

```
line_integral2d(
  f,
  r,
  a,
  b,
  plot = TRUE,
  n_curve = 400,
  n_curtain_v = 40,
  n_surf_x = 80,
  n_surf_y = 80,
  colorscale = "Viridis",
  surface_opacity = 0.65,
  show_surface_grid = TRUE,
  surface_grid_color = "rgba(80,80,80,0.25)",
  surface_grid_width = 1,
  curtain = 0.4,
  curtain_colorscale = c("white", "#2a9d8f"),
  curve_color = "black",
  curve_width = 3,
  lifted_color = "red",
  lifted_width = 2,
  h = NULL,
  method = c("adaptive", "simpson"),
  n_simpson = 1000,
 scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
    zaxis = list(title = "z")),
  bg = list(paper = "white", plot = "white")
)
```

Arguments

`f`	A scalar field, given as `function(x, y)` returning a numeric value.

`r`	A parametric curve, given as `function(t)` that returns a numeric vector of length two, interpreted as `c(x, y)`.
`a, b`	Numeric scalars giving the parameter limits, with `b > a`.
`plot`	Logical; if `TRUE`, produces a visualization using **plotly**.
`n_curve`	Number of sample points along the curve for plotting.
`n_curtain_v`	Number of subdivisions in the vertical direction for the curtain.
`n_surf_x, n_surf_y`	Grid resolution for sampling the surface `z = f(x, y)`.
`colorscale`	Color scale for the surface. It may be a **plotly** scale name, a single color, or a vector of colors defining a gradient.
`surface_opacity`	Numeric opacity for the surface, between 0 and 1.
`show_surface_grid`	Logical; if `TRUE`, overlays grid lines on the surface.
`surface_grid_color`	Character string giving the color of surface grid lines.
`surface_grid_width`	Numeric width of surface grid lines.
`curtain`	Numeric value between 0 and 1 giving the opacity of the vertical curtain. A value of zero disables the curtain.
`curtain_colorscale`	Color scale for the curtain, using the same formats accepted by `colorscale`.
`curve_color`	Color for the curve drawn at height zero.
`curve_width`	Width of the curve drawn at height zero.
`lifted_color`	Color for the lifted curve whose height is `f(x(t), y(t))`.
`lifted_width`	Width of the lifted curve.
`h`	Step size used for centered finite differences when computing the derivative of `r(t)`. If `NULL`, a default is chosen automatically.
`method`	Integration method; may be `"adaptive"` (using `stats::integrate`) or `"simpson"` (using a composite Simpson rule).
`n_simpson`	Number of subintervals for the Simpson method. This value is automatically adjusted to be even.
`scene`	List configuring the 3D scene in **plotly**.
`bg`	List with background color settings for the figure, with entries such as `paper` and `plot`.

Details

The function evaluates the scalar field along the curve and computes an approximation of the derivative of `r(t)` using centered finite differences. The integral can be computed either through a built-in adaptive integration routine or via a composite Simpson rule with a user specified number of subintervals.

For visualization, the function:

- builds a rectangular surface `z = f(x, y)` using only the endpoints of the curve,
- plots the curve in the plane,
- plots a lifted copy of the curve where the height is given by the scalar field,
- optionally constructs a vertical curtain over the curve by extruding the height values.

Value

A list with:

- `value`: the numeric value of the line integral,
- `fig`: a **plotly** figure when `plot = TRUE`, or `NULL` otherwise.

Examples

```
f <- function(x, y) x^2 + y^2
r <- function(t) c(t*cos(t), t*sin(3*t))
line_integral2d(f, r, a = 0, b = 2*pi, plot = FALSE)
```

`line_integral3d_work` *3D line integral with work visualization*

Description

Compute a numerical approximation of the line integral of a vector field along a parametric space curve and optionally draw a three dimensional visualization of the curve together with arrows of the field.

Usage

```
line_integral3d_work(
  field,
  r,
  a,
  b,
  plot = TRUE,
  n_curve = 600,
  n_field = 7,
  field_ranges = NULL,
  pad = 0.15,
  arrows = c("both", "line", "cone", "none"),
  arrow_scale = 0.1,
  normalize_bias = 1,
 scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
    zaxis = list(title = "z")),
  bg = list(paper = "white", plot = "white")
)
```

Arguments

field	Function that represents the vector field. It must be a function of three or four numeric arguments. In the three argument form the arguments are x, y and z. In the four argument form the arguments are x, y, z and t. The function must return a numeric vector of length three.
r	Function of one numeric argument t that returns a numeric vector c(x, y, z) of length three with the coordinates of the curve.
a	Numeric scalar with the lower value of the parameter interval.
b	Numeric scalar with the upper value of the parameter interval. It must satisfy `b > a`.
plot	Logical value. If `TRUE`, a plotly object with the field arrows and the curve is created.
n_curve	Integer number of sampled points on the curve.
n_field	Integer number of grid points per axis for the field arrows. The total number of arrows is `n_field^3`.
field_ranges	Optional list with named components `x`, `y` and `z`, each a numeric vector of length two giving the range used to build the field grid. If `NULL`, the ranges are taken from the bounding box of the curve and expanded by `pad`.
pad	Numeric fraction used to expand the automatic field ranges.
arrows	Character string that selects which arrow style is drawn. One of `"both"`, `"line"`, `"cone"` or `"none"`.
arrow_scale	Numeric factor that controls the length of the arrows as a fraction of the size of the domain.
normalize_bias	Numeric value used to regularize the scaling of the field magnitude when computing arrows.
scene	List of plotly scene options passed to `plotly::layout()`.
bg	List with two character elements named `paper` and `plot` that control the background colours in the plotly layout.

Details

The parameter t runs from a to b. The curve r(t) must return a numeric vector of length three. The field field(x, y, z) may optionally also depend on t through a fourth argument.

Value

A list with components:

- `value`: numeric value of the line integral (work).
- `fig`: plotly object when `plot = TRUE`, otherwise `NULL`.
- `field_box`: list with numeric ranges used for the field grid, with components `x`, `y` and `z`.

Examples

```
field <- function(x, y, z) c(-y, x, 0.2*z)
r <- function(t) c(cos(t), sin(t), 0.25*t)
out <- line_integral3d_work(
  field = field, r = r, a = 0, b = 2*pi,
  plot = FALSE, n_curve = 200, n_field = 5
)
out$value
```

---

line_integral_vector2d

2D line integral of a vector field with visualization

Description

This function computes a numerical approximation of the line integral of a planar vector field along a parametric curve r(t) on the interval from a to b. The derivative of the curve is approximated by finite differences and the integral is evaluated either by an adaptive numerical integrator or by a composite Simpson rule.

Usage

```
line_integral_vector2d(
  field,
  r,
  a,
  b,
  plot = TRUE,
  n_curve = 600,
  grid_n = 15,
  padding = 0.15,
  h = NULL,
  method = c("adaptive", "simpson"),
  n_simpson = 1000,
  arrow_scale = 0.08,
  normalize_bias = 1,
  field_color = "rgba(0,0,0,0.55)",
  field_width = 1.8,
  traj_palette = "RdBu",
  traj_width = 5,
  show_markers = FALSE,
 scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
    zaxis = list(title = "z")),
  bg = list(paper = "white", plot = "white")
)
```

Arguments

`field`	Vector field in the plane. A function `function(x, y)` that returns a numeric vector of length 2 `c(Fx, Fy)`.
`r`	Parametric curve in the plane. A function `function(t)` that returns a numeric vector of length 2 `c(x, y)`.
`a`	Numeric scalar. Left endpoint of the parameter interval.
`b`	Numeric scalar. Right endpoint of the parameter interval. Must satisfy `b > a`.
`plot`	Logical. If `TRUE`, an interactive **plotly** figure is created with the field and the curve.
`n_curve`	Integer. Number of parameter values used to sample the curve.
`grid_n`	Integer. Number of grid points per axis used to draw the vector field arrows.
`padding`	Numeric scalar. Relative margin added around the bounding box of the curve when building the field grid.
`h`	Numeric scalar or `NULL`. Step size used in the finite difference approximation of the derivative of the curve. If `NULL`, a small step is chosen automatically, based on the length of the interval `b - a`.
`method`	Character string. Integration method for the line integral. One of `"adaptive"` (uses `stats::integrate`) or `"simpson"` (composite Simpson rule).
`n_simpson`	Integer. Number of subintervals used when `method = "simpson"`. If it is odd, it is increased by one internally.
`arrow_scale`	Numeric scalar. Controls the overall length of the field arrows as a fraction of the plot span.
`normalize_bias`	Numeric scalar. Saturation parameter used to avoid extremely long arrows for large field magnitudes.
`field_color`	Character string. Color used for the field arrows.
`field_width`	Numeric scalar. Line width for the field arrows.
`traj_palette`	Color scale used to represent the power along the trajectory. Passed to **plotly** as a colorscale name.
`traj_width`	Numeric scalar. Line width for the trajectory.
`show_markers`	Logical. If `TRUE`, markers are drawn along the trajectory in addition to the line.
`scene`	List with **plotly** scene options (axis titles, aspect mode, etc.) passed to `plotly::layout()`.
`bg`	List with background colors for **plotly**, with components `paper` and `plot`.

Details

Optionally, the function can build an interactive **plotly** figure that shows a grid of arrows for the vector field and the curve colored by the local power field(r(t)) * r'(t).

Value

A list with components:

- `value`: numeric value of the line integral.
- `samples`: data frame with sampled points, velocities and power along the trajectory.
- `fig`: **plotly** object when `plot = TRUE`, otherwise `NULL`.

Examples

```
# Simple example:
# field(x, y) = (y, -x), r(t) = (cos t, sin t), t in [0, 2*pi]
line_integral_vector2d(
  field = function(x, y) c(y, -x),
  r = function(t) c(cos(t), sin(t)),
  a = 0, b = 2*pi, plot = FALSE
)
```

`newton_raphson2d` *Newton-Raphson method for systems in R^2 with animation (Plotly)*

Description

Applies the Newton-Raphson method to solve a nonlinear system in two variables:

$$\mathbf{F}(x, y) = \mathbf{0},$$

where

$$\mathbf{F}(x, y) = \begin{pmatrix} f_1(x, y) \\ f_2(x, y) \end{pmatrix}.$$

At an iterate

$$\mathbf{x}_n = (x_n, y_n),$$

the Newton update is

$$\mathbf{x}_{n+1} = \mathbf{x}_n - J(\mathbf{x}_n)^{-1}\mathbf{F}(\mathbf{x}_n),$$

where the Jacobian matrix is

$$J(x, y) = \begin{pmatrix} \frac{\partial f_1}{\partial x}(x, y) & \frac{\partial f_1}{\partial y}(x, y) \\ \frac{\partial f_2}{\partial x}(x, y) & \frac{\partial f_2}{\partial y}(x, y) \end{pmatrix}.$$

If a Jacobian function is not provided, partial derivatives are approximated numerically by central differences.

Usage

```
newton_raphson2d(
  f,
  x0,
  J = NULL,
  h = 1e-04,
  max_iter = 10L,
  tol = 1e-08,
  xlim = NULL,
  ylim = NULL,
  n_grid = 120L,
  frame_ms = 700L,
  transition_ms = 450L,
  title = NULL,
  safe_mode = TRUE
)
```

Arguments

`f`	Function. A function f(x,y) that returns a numeric vector of length 2: c(f1(x,y), f2(x,y)).
`x0`	Numeric vector of length 2. Initial guess c(x0, y0).
`J`	Optional function. A function J(x,y) returning a 2x2 numeric matrix (the Jacobian). If NULL, a numerical Jacobian is used.
`h`	Numeric scalar. Step size for numerical partial derivatives (when J is NULL).
`max_iter`	Integer. Maximum number of Newton iterations.
`tol`	Numeric scalar. Stopping tolerance based on the infinity norm: max(|f1|,|f2|) <= tol.
`xlim`	Numeric vector of length 2. Plot range for x. If NULL, it is chosen from the iterates.
`ylim`	Numeric vector of length 2. Plot range for y. If NULL, it is chosen from the iterates.
`n_grid`	Integer. Grid size per axis used to draw the zero level curves.
`frame_ms`	Integer. Frame duration in milliseconds.
`transition_ms`	Integer. Transition duration in milliseconds.
`title`	Character. Plot title. If NULL, a default title is used.
`safe_mode`	Logical. If TRUE, uses calmer animation defaults intended to reduce flicker and visual stress.

Details

The animation shows the two zero level curves

$$f_1(x, y) = 0 \quad \text{and} \quad f_2(x, y) = 0$$

together with the Newton iterates and the step segments

$$\mathbf{x}_n \to \mathbf{x}_{n+1}.$$

Value

A list with components:

plot A plotly object (htmlwidget) with animation frames.

iterates Data frame of iterates (n, x, y, f1, f2, norm_inf).

root Numeric vector c(x, y) with the last iterate.

converged Logical. TRUE if convergence was detected within max_iter.

Examples

```
library(plotly)

# Example system:
# f1(x,y)=x^2+y^2-1 (unit circle)
# f2(x,y)=x-y (line)
f <- function(x, y) c(x^2 + y^2 - 1, x - y)

out <- newton_raphson2d(f, x0 = c(0.8, 0.2), max_iter = 8)
out$plot
out$iterates
out$converged
out$root
```

`newton_raphson_anim` *Newton-Raphson root finding with tangent animation (Plotly)*

Description

Builds a Plotly animation of the Newton-Raphson method for finding a root of a real function. Each frame shows the tangent line at the current iterate and how its x-intercept defines the next iterate.

Usage

```
newton_raphson_anim(
  f,
  x0,
  df = NULL,
  h = 1e-04,
  max_iter = 10L,
  tol = 1e-08,
  xlim = NULL,
  n_curve = 600L,
  frame_ms = 600L,
  transition_ms = 400L,
  title = NULL,
  safe_mode = TRUE
)
```

Arguments

`f`	Function. A real-valued function f(x). Must accept a numeric vector and return a numeric vector of the same length.
`x0`	Numeric scalar. Initial guess.
`df`	Optional function. Derivative f'(x). If NULL, a numerical derivative is used.

`h`	Numeric scalar. Step size for numerical derivative (when df is NULL).
`max_iter`	Integer. Maximum number of iterations.
`tol`	Numeric scalar. Stopping tolerance based on |f(x_n)|.
`xlim`	Numeric vector of length 2. Plot range for x. If NULL, it is chosen around the iterates.
`n_curve`	Integer. Number of points used to draw the curve.
`frame_ms`	Integer. Frame duration in milliseconds.
`transition_ms`	Integer. Transition duration in milliseconds.
`title`	Character. Plot title. If NULL, a default title is used.
`safe_mode`	Logical. If TRUE, use calmer animation defaults intended to reduce flicker and visual stress.

Details

The Newton-Raphson update is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

If a derivative function is not provided, the derivative is approximated numerically by the central difference

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

Value

A list with components:

plot A plotly object (htmlwidget) with animation frames.

iterates Data frame with iterations (n, x, fx, dfx).

root Last iterate (approximate root).

converged Logical. TRUE if convergence was detected within max_iter.

Examples

```
library(plotly)

f <- function(x) x^3 - 2*x - 5
out <- newton_raphson_anim(f, x0 = 2)
out$plot
out$root

g <- function(x) cos(x) - x
newton_raphson_anim(g, x0 = 1)
```

normal3d *Principal normal vectors along a 3D curve*

Description

Computes numerical principal normal vectors of a three-dimensional parametric curve at several parameter values. The curve is described by three coordinate functions X, Y and Z. At each evaluation point, the function approximates the first and second derivatives of the curve, builds the unit tangent and binormal vectors, and then obtains the principal normal as the unit vector orthogonal to both of them.

Usage

```
normal3d(
  X,
  Y,
  Z,
  a,
  b,
  t_points,
  h = 1e-04,
  plot = FALSE,
  n_samples = 400,
  vec_scale = NULL,
  vec_factor = 1,
  curve_line = list(color = "blue", width = 2, dash = "solid"),
  N_line = list(color = "green", width = 5, dash = "solid"),
  show_curve = TRUE,
  show_points = TRUE,
  point_marker = list(color = "black", size = 3, symbol = "circle"),
 scene = list(aspectmode = "data", xaxis = list(title = "x(t)"), yaxis = list(title =
    "y(t)"), zaxis = list(title = "z(t)")),
  bg = list(paper = "white", plot = "white"),
  tol = 1e-10
)
```

Arguments

X	Function giving the x coordinate of the curve as a function of the parameter t.
Y	Function giving the y coordinate of the curve as a function of the parameter t.
Z	Function giving the z coordinate of the curve as a function of the parameter t.
a	Lower endpoint of the parameter interval.
b	Upper endpoint of the parameter interval.
t_points	Numeric vector of parameter values at which the principal normal is evaluated and, optionally, plotted.

`h`	Step size for the centered finite-difference approximations used to compute derivatives.
`plot`	Logical; if `TRUE`, displays a 3D plot of the curve and the corresponding normal segments using **plotly**.
`n_samples`	Number of points used to sample and draw the curve for plotting purposes.
`vec_scale`	Base length used for the normal segments. If `NULL`, it is estimated as a small fraction of the overall size of the sampled curve.
`vec_factor`	Multiplicative factor applied to `vec_scale` to control the visual length of the normal segments.
`curve_line`	List with **plotly** style options for the base curve.
`N_line`	List with **plotly** style options for the normal segments.
`show_curve`	Logical; if `TRUE`, the base curve is drawn.
`show_points`	Logical; if `TRUE`, the evaluation points are marked on the curve.
`point_marker`	List with **plotly** marker options for the evaluation points.
`scene`	List with 3D scene settings for **plotly**.
`bg`	Background colors for the figure, given as a list with entries such as `paper` and `plot`.
`tol`	Numeric tolerance used to detect singular or nearly singular situations in which the normal direction cannot be computed reliably.

Details

For every parameter value in `t_points`, the function:

- approximates the first derivative of the curve with respect to the parameter,
- normalizes this derivative to obtain a unit tangent direction,
- uses the first and second derivative vectors to construct a direction orthogonal to the tangent and interprets it as a binormal direction,
- builds the principal normal direction as a unit vector orthogonal to both the tangent and the binormal.

When the curvature of the curve at a given parameter value is extremely small, the normal direction becomes poorly defined from a numerical point of view. In such situations, the function marks the corresponding components of the normal vector as `NA`.

Optionally, the function can display the curve and the associated normal segments in a 3D interactive plot using **plotly**. The base curve, the evaluation points and the normal segments can be shown or hidden independently.

Value

A tibble with columns `t`, `x`, `y`, `z`, `Nx`, `Ny` and `Nz`, where the last three columns contain the components of the principal normal vector at each parameter value.

Examples

```
X <- function(t) t*cos(t)
Y <- function(t) t*sin(3*t)
Z <- function(t) t
normal3d(X, Y, Z, a = 0, b = 2*pi, t_points = c(pi/3, pi, 5*pi/3))
```

---

osculating_circle3d *Osculating discs and circles of a spatial curve*

Description

For a three-dimensional parametric curve, this function constructs numerical approximations to the osculating circles (and associated discs) at a set of parameter values. At each requested point on the curve, it approximates the Frenet frame and the curvature, and then uses this information to define the center and radius of the local osculating circle. Optionally, it can display these circles or discs in an interactive 3D visualization using **plotly**.

Usage

```
osculating_circle3d(
  X,
  Y,
  Z,
  a,
  b,
  t_points,
  h = 1e-04,
  plot = FALSE,
  n_samples = 400,
  fill = c("disk", "ring"),
  ru = 24,
  rv = 72,
  colorscale = "Reds",
  opacity = 0.6,
  ring_line = list(color = "red", width = 4, dash = "solid"),
  show_curve = TRUE,
  show_points = TRUE,
  curve_line = list(color = "blue", width = 2, dash = "solid"),
  point_marker = list(color = "black", size = 3, symbol = "circle"),
  show_radius = FALSE,
  radius_phase = 0,
  radius_line = list(color = "orange", width = 5, dash = "solid"),
 scene = list(aspectmode = "data", xaxis = list(title = "x(t)"), yaxis = list(title =
    "y(t)"), zaxis = list(title = "z(t)")),
  bg = list(paper = "white", plot = "white"),
 lighting = list(ambient = 1, diffuse = 0.15, specular = 0, roughness = 1, fresnel = 0),
  tol = 1e-10
)
```

Arguments

`X, Y, Z`	Functions of `t` returning the coordinate components of the curve.
`a, b`	Numeric endpoints of the parameter interval.
`t_points`	Numeric vector of parameter values at which osculating circles or discs are constructed.
`h`	Step size for centered finite-difference approximations.
`plot`	Logical; if `TRUE`, creates a 3D visualization using **plotly**.
`n_samples`	Number of sample points used to draw the base curve when `show_curve = TRUE`.
`fill`	Character; either `"disk"` for a filled surface or `"ring"` for the circumference only.
`ru`	Number of radial subdivisions when drawing a filled disc.
`rv`	Number of angular subdivisions; also used as the number of points on each ring.
`colorscale`	Character string giving the **plotly** colorscale used for the discs.
`opacity`	Numeric value between 0 and 1 controlling the opacity of the discs when `fill = "disk"`.
`ring_line`	List with style options for the ring when `fill = "ring"`.
`show_curve, show_points`	Logical values indicating whether the base curve and the corresponding points on the curve should be displayed.
`curve_line, point_marker`	Lists with **plotly** style options for the base curve and the points.
`show_radius`	Logical; if `TRUE`, draws a radius segment from the center of each osculating circle to its boundary.
`radius_phase`	Angle, in radians, that determines the direction of the displayed radius.
`radius_line`	List with **plotly** style options for the radius segment.
`scene`	List with 3D scene settings for the **plotly** figure.
`bg`	List defining background colors for the figure, typically with entries `paper` and `plot`.
`lighting`	List with lighting options for `add_surface` when `fill = "disk"`.
`tol`	Numeric tolerance used in derivative-based checks and to detect degenerate cases in which curvature or frame vectors cannot be computed reliably.

Details

For each parameter value in `t_points`, the function:

- evaluates the curve and approximates its first and second derivatives,
- constructs approximate tangent, normal and binormal directions,
- estimates the curvature from the derivative information,

- defines the center of the osculating circle by moving from the curve point along the normal direction by a distance equal to the reciprocal of the curvature,
- records the corresponding radius as that same reciprocal quantity.

Depending on the value of `fill`, the function either:

- builds a filled disc that lies in the osculating plane and is bounded by the osculating circle, or
- draws only the circumference corresponding to that circle.

A regular sampling of angles around the osculating circle is used to generate the discrete representation. For filled discs, radial subdivisions are added to obtain a surface mesh. The resulting objects can be combined with a sampled version of the base curve and additional elements such as radius segments.

Value

A list with two components:

`data` A tibble with columns `t`, `x`, `y`, `z`, `kappa`, `cx`, `cy`, `cz`, `radius`, `Tx`, `Ty`, `Tz`, `Nx`, `Ny`, `Nz`, `Bx`, `By`, `Bz`, containing the parameter values, the curve coordinates, the numerical curvature, the centers and radii of the osculating circles, and the associated Frenet frame vectors.

`plot` A **plotly** object when `plot = TRUE`, otherwise `NULL`.

Examples

```
X <- function(t) cos(t)
Y <- function(t) sin(t)
Z <- function(t) 0.2 * t
osculating_circle3d(
  X, Y, Z,
  a = 0, b = 6 * pi,
  t_points = c(pi, 2 * pi),
  plot = FALSE
)
```

`osculating_ribbon3d` *Osculating ribbon along a 3D parametric curve*

Description

Constructs a narrow ribbon that follows a three-dimensional parametric curve. The ribbon is based on the Frenet-Serret frame of the curve, computed numerically along a set of sample points. The ribbon extends a small distance in the normal direction of the curve, producing a thin band that helps visualize how the curve bends and twists in space.

Usage

```
osculating_ribbon3d(
  X,
  Y,
  Z,
  a,
  b,
  h = 1e-04,
  plot = FALSE,
  n_t = 400,
  n_u = 25,
  u_max = 1,
  colorscale = "Blues",
  opacity = 0.35,
  show_curve = TRUE,
  show_centers = TRUE,
  curve_line = list(color = "black", width = 2),
  centers_line = list(color = "red", width = 2),
  show_surface_grid = TRUE,
  surface_grid_color = "rgba(60,80,200,0.25)",
  surface_grid_width = 1,
  show_axis_grid = FALSE,
  scene = list(aspectmode = "data", xaxis = list(title = "x(t)"), yaxis = list(title =
    "y(t)"), zaxis = list(title = "z(t)")),
  bg = list(paper = "white", plot = "white"),
  lighting = list(ambient = 1, diffuse = 0.15, specular = 0, roughness = 1, fresnel = 0),
  tol = 1e-10
)
```

Arguments

`X, Y, Z`	Functions returning the coordinate components of the curve as functions of the parameter `t`.
`a, b`	Numeric values giving the endpoints of the parameter interval.
`h`	Step size used in the finite-difference approximations.
`plot`	Logical; if `TRUE`, produces a 3D visualization of the ribbon using **plotly**.
`n_t`	Number of sample points along the curve.
`n_u`	Number of subdivisions across the width of the ribbon.
`u_max`	Half-width of the ribbon, measured in units of the normal direction.
`colorscale`	Character string giving the **plotly** colorscale used for the ribbon surface.
`opacity`	Numeric value between 0 and 1 controlling the transparency of the ribbon.
`show_curve`	Logical; if `TRUE`, draws the base curve.
`show_centers`	Logical; if `TRUE`, draws the centerline joining the midpoints of the ribbon cross-sections.
`curve_line`	List with **plotly** style options for the base curve.

`centers_line`	List with **plotly** style options for the centerline.
`show_surface_grid`	Logical; if `TRUE`, draws a grid on the surface of the ribbon.
`surface_grid_color`	Character string giving the color of the grid lines on the ribbon.
`surface_grid_width`	Numeric value giving the width of the surface grid lines.
`show_axis_grid`	Logical; if `TRUE`, displays gridlines on the coordinate axes in the **plotly** scene.
`scene`	List with 3D scene settings for the **plotly** figure.
`bg`	List defining the background colors of the figure, typically with entries `paper` and `plot`.
`lighting`	List with lighting settings for the surface in **plotly**.
`tol`	Numeric tolerance used to detect numerical instabilities when computing the derivative-based frame vectors.

Details

The function samples the curve at `n_t` points and computes the numerical tangent, normal and binormal directions using finite-difference approximations of the derivatives. At each sampled point, a short segment is taken in the normal direction to define the width of the ribbon. These segments are interpolated across the curve and subdivided according to `n_u` to produce a mesh that represents the ribbon surface.

Optionally, the function can display the ribbon in an interactive 3D plot using **plotly**. The base curve, the centerline, and optional grid lines on the ribbon surface can be shown or hidden independently.

Value

A list with two components:

`data` A tibble containing the sampled parameter values, the coordinates of the curve, and the corresponding tangent, normal and binormal directions.

`plot` A **plotly** object if `plot = TRUE`, otherwise `NULL`.

Examples

```
X <- function(t) cos(t)
Y <- function(t) sin(t)
Z <- function(t) 0.2 * t
osculating_ribbon3d(X, Y, Z, a = 0, b = 4*pi, plot = FALSE)
```

`partial_derivatives_surface` *Partial derivatives of z = f(x, y) at a point with 3D visualization*

Description

Numerically approximates the partial derivatives $f_x(x_0, y_0)$ and $f_y(x_0, y_0)$ of a scalar field $z = f(x, y)$ at a given point (x_0, y_0) using central finite differences.

Usage

```
partial_derivatives_surface(
  f,
  x0,
  y0,
  h = NULL,
  xlim = NULL,
  ylim = NULL,
  nx = 60L,
  ny = 60L,
  plot = TRUE,
 scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
    zaxis = list(title = "z")),
  bg = list(paper = "white", plot = "white")
)
```

Arguments

`f`	Function `function(x, y)` returning a numeric scalar `f(x, y)`.
`x0`, `y0`	Numeric scalars; coordinates of the point where the partial derivatives are evaluated.
`h`	Numeric step for the central finite differences. If `NULL`, a default value is chosen as `1e-4 * (1 + max(abs(x0), abs(y0)))`.
`xlim`	Numeric length-2 vector `c(x_min, x_max)`. Range for the x-axis used to draw the surface. If `NULL`, a symmetric window around `x0` is used.
`ylim`	Numeric length-2 vector `c(y_min, y_max)`. Range for the y-axis used to draw the surface. If `NULL`, a symmetric window around `y0` is used.
`nx`, `ny`	Integer grid sizes (number of points) along `x` and `y` for the surface plot. Recommended values are at least 20.
`plot`	Logical; if `TRUE`, builds and returns a **plotly** surface plot. If `FALSE`, only the numeric derivatives are returned.
`scene`	List with **plotly** 3D scene options (axis titles, aspect mode, and so on) passed to `plotly::layout()`.
`bg`	List with background colors for **plotly**, with components `paper` and `plot`.

Details

Optionally, it builds a 3D **plotly** surface for $z = f(x, y)$ on a rectangular window around (x_0, y_0) and overlays:

- the intersection curve of the surface with the plane $y = y_0$ and its tangent line given by $f_x(x_0, y_0)$;
- the intersection curve with the plane $x = x_0$ and its tangent line given by $f_y(x_0, y_0)$;
- the base point $(x_0, y_0, f(x_0, y_0))$.

Value

A list with components:

- `fx`: numeric scalar, approximation of $f_x(x_0, y_0)$.
- `fy`: numeric scalar, approximation of $f_y(x_0, y_0)$.
- `f0`: numeric scalar, $f(x_0, y_0)$.
- `fig`: a **plotly** object if `plot = TRUE` and **plotly** is available; otherwise `NULL`.

Examples

```
f <- function(x, y) x^2 + 3 * x * y - y^2
res <- partial_derivatives_surface(
  f,
  x0 = 1, y0 = -1,
  xlim = c(-1, 3),
  ylim = c(-3, 1),
  nx = 60, ny = 60
)
res$fx
res$fy
```

`plot_curve3d` *Plot a 3D parametric curve with plotly*

Description

Creates an interactive 3D plot of a parametric curve given a tibble with columns `t`, `x`, `y`, `z` (typically produced by `curve_sample3d()`).

Usage

```
plot_curve3d(
  data,
  mode = "lines",
  line = list(color = "blue", width = 3, dash = "solid"),
  marker = NULL,
  title = NULL,
  scene = list(xaxis = list(title = "x(t)"), yaxis = list(title = "y(t)"), zaxis =
    list(title = "z(t)")),
  bg = list(paper = "white", plot = "white")
)
```

Arguments

data	Tibble with columns t, x, y, z.
mode	Character string. Plotly trace mode, e.g. "lines" or "lines+markers".
line	List with line styling options, such as list(color = "blue", width = 3, dash = "solid").
marker	Optional list with marker styling options, or NULL to omit markers.
title	Optional character string for the plot title.
scene	List specifying 3D axis titles and options, passed to plotly::layout(), typically including xaxis, yaxis, and zaxis.
bg	List with background colors, e.g. list(paper = "white", plot = "white").

Details

This function requires the **plotly** package to be installed.

Value

A **plotly** object, which is printed for interactive visualization.

See Also

curve_sample3d(), arc_length3d()

Examples

```
data <- curve_sample3d(
  function(t) 2 * cos(t),
  function(t) 3 * sin(t),
  function(t) t / 5,
  0, 2 * pi, 100
)

if (requireNamespace("plotly", quietly = TRUE)) {
  plot_curve3d(data, line = list(color = "red", width = 4))
}
```

`plot_surface_with_tangents` *Surface with tangent lines at a point*

Description

Draws the surface $z = f(x, y)$ on a rectangular domain and overlays two tangent line segments at a given point (x_0, y_0): one tangent in the direction of the x-axis and one tangent in the direction of the y-axis. The partial derivatives are approximated numerically by central finite differences.

Usage

```
plot_surface_with_tangents(
  f,
  x0,
  y0,
  xlim = c(-3, 3),
  ylim = c(-3, 3),
  n = 120,
  h = 1e-05,
  t_len = 0.75,
  title_prefix = "f"
)
```

Arguments

`f`	Scalar field, given as `function(x, y)` returning a numeric value.
`x0`, `y0`	Numeric scalars with the coordinates of the point where the tangent lines are drawn.
`xlim`	Numeric vector `c(x_min, x_max)` giving the range of the x-axis used to draw the surface.
`ylim`	Numeric vector `c(y_min, y_max)` giving the range of the y-axis used to draw the surface.
`n`	Integer number of grid points per axis used to discretize the surface. Must be at least 20.
`h`	Numeric step used for the central finite–difference approximation of the partial derivatives f_x and f_y.
`t_len`	Numeric scalar giving half the length of the tangent segments along the x and y directions.
`title_prefix`	Optional character string used as a prefix in the plot title (for example, the name of the function f).

Value

A **plotly** object representing the surface $z = f(x, y)$ together with the point $(x_0, y_0, f(x_0, y_0))$ and the two tangent line segments. The object can be further modified with usual **plotly** tools.

Examples

```
f <- function(x, y) sin(x) * cos(y)
p <- plot_surface_with_tangents(
  f,
  x0 = 1, y0 = 1,
  xlim = c(-3, 3),
  ylim = c(-3, 3),
  n = 80
)
# p
```

`region_xyz0` *Planar region $\{(x, y) : a \leq x \leq b, H1(x) \leq y \leq H2(x)\}$ drawn at height z0*

Description

Builds the planar region

$$\Omega = \{(x, y) :\ a \leq x \leq b,\ H_1(x) \leq y \leq H_2(x)\}$$

and renders it as a thin patch on the plane $z = z_0$. Optionally it draws the boundary curves, partition lines along x, and a surface patch.

Builds the planar region defined by $a \leq x \leq b$ and $H_1(x) \leq y \leq H_2(x)$, and renders it as a thin patch on the plane $z = z_0$. Optionally it draws the boundary curves, partition lines along x, and a surface patch.

Usage

```
region_xyz0(
  H1,
  H2,
  a,
  b,
  z0,
  D,
  plot = TRUE,
  n_curve = 800,
  show_surface = FALSE,
  surface_colorscale = "Blues",
  surface_opacity = 0.3,
  show_surface_grid = TRUE,
  surface_grid_color = "rgba(60,80,200,0.25)",
  surface_grid_width = 1,
  boundary_line = list(color = "blue", width = 2),
  partition_line = list(color = "blue", width = 1),
  show_end_edges = TRUE,
 scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
    zaxis = list(title = "z")),
  bg = list(paper = "white", plot = "white")
)

region_xyz0(
  H1,
  H2,
  a,
  b,
  z0,
  D,
  plot = TRUE,
  n_curve = 800,
  show_surface = FALSE,
  surface_colorscale = "Blues",
  surface_opacity = 0.3,
  show_surface_grid = TRUE,
  surface_grid_color = "rgba(60,80,200,0.25)",
  surface_grid_width = 1,
  boundary_line = list(color = "blue", width = 2),
  partition_line = list(color = "blue", width = 1),
  show_end_edges = TRUE,
 scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
    zaxis = list(title = "z")),
  bg = list(paper = "white", plot = "white")
)
```

Arguments

`H1, H2`	Functions `H_i(x)`; lower and upper y-boundaries.
a, b	Numeric scalars; x-interval endpoints with $a < b$.
`z0`	Numeric scalar; height of the display plane $z = z_0$.
`D`	Integer > 0; number of x-partitions (controls vertical slices and grid density).
`plot`	Logical; if `TRUE`, draw the region with **plotly**.
`n_curve`	Integer; number of samples used to trace each boundary curve $y = H_i(x)$.
`show_surface`	Logical; if `TRUE`, draw a thin surface patch of the region at $z = z_0$.
`surface_colorscale`	Character; Plotly colorscale for the surface patch (for example, `"Blues"`).
`surface_opacity`	Numeric in $[0, 1]$; opacity of the surface patch.

`show_surface_grid`
: Logical; show grid/contours on the surface patch.

`surface_grid_color`
: Character; color for surface grid lines.

`surface_grid_width`
: Numeric; width for surface grid lines.

`boundary_line`
: List; style for the boundary polylines (for example, `list(color = "blue", width = 2)`).

`partition_line`
: List; style for the partition lines at $x = a + k(b - a)/D$.

`show_end_edges`
: Logical; draw edges at $x = a$ and $x = b$.

`scene`
: List; Plotly 3D scene options.

`bg`
: List with `paper` and `plot` background colors.

Value

A list with components:

`data` List (or tibble, depending on the implementation) with the sampled curves and/or the grid used.

`plot` A **plotly** object when `plot = TRUE`, otherwise `NULL`.

A list with components:

`data` List (or tibble, depending on the implementation) with the sampled curves and/or the grid used.

`plot` A **plotly** object when `plot = TRUE`, otherwise `NULL`.

Examples

```
H1 <- function(x) 0
H2 <- function(x) 1 - x
# Region under H2 and above H1 in [0,1], drawn at z = 0
# region_xyz0(H1, H2, a = 0, b = 1, z0 = 0,
#             D = 20, plot = TRUE, show_surface = TRUE)
```

```
H1 <- function(x) 0
H2 <- function(x) 1 - x
# Region under H2 and above H1 in [0,1], drawn at z = 0
# region_xyz0(H1, H2, a = 0, b = 1, z0 = 0,
#             D = 20, plot = TRUE, show_surface = TRUE)
```

`related_rates_grad` *Related rates via the gradient (implicit constraint)*

Description

Computes related rates for an implicit constraint using the gradient. Let

$$g(\mathbf{x}) = 0, \quad \mathbf{x} = \mathbf{x}(t) \in \mathbb{R}^k.$$

Differentiating with respect to time yields

$$\nabla g(\mathbf{x}(t)) \cdot \mathbf{x}'(t) = 0.$$

If all components of $\mathbf{x}'(t)$ are known except one, the missing rate is determined by this orthogonality condition (velocity tangent to the level set).

Usage

```
related_rates_grad(g, x, known_rates, solve_for, var_names = NULL, h = 1e-06)
```

Arguments

`g`	Function. A scalar function g(x1, x2, ..., xk) defining the implicit constraint g = 0. It must accept k numeric arguments and return a numeric scalar.
`x`	Numeric vector of length k. Point where rates are evaluated. This point should satisfy g(x) = 0 (approximately).
`known_rates`	Numeric vector of known rates. It can be: (1) a named numeric vector with names matching var_names (e.g. "x","y"), (2) a named numeric vector with names matching paste0("d", var_names) (e.g. "dx","dy"), or (3) an unnamed numeric vector of length k with NA for the unknown component.
`solve_for`	Integer or character. Which rate to solve for. If integer, it is the position in var_names (1..k). If character, it may be either a variable name (e.g. "y") or a rate name (e.g. "dy").
`var_names`	Character vector of length k. Variable names. If NULL, defaults to c("x1","x2",...,"xk").
`h`	Numeric scalar. Step size for numerical partial derivatives.

Value

A list with components:

rate Numeric scalar. The solved rate (the requested component of x'(t)).

rates Named numeric vector length k. Full rate vector x'(t). Names are paste0("d", var_names).

grad Numeric vector length k. Gradient of g at x.

dot Numeric scalar. Dot product grad . rates (should be near 0).

gx Numeric scalar. g(x) value (should be near 0 if x is on the constraint).

Examples

```
# Ladder (implicit circle): x^2 + y^2 = L^2
# Suppose L = 5, at the instant (x,y) = (4,3) and dx/dt = -1.
# Then dy/dt = -(x/y) dx/dt = (4/3).
g <- function(x, y) x^2 + y^2 - 25

out <- related_rates_grad(
  g = g,
  x = c(4, 3),
  known_rates = c(dx = -1, dy = NA),
  solve_for = "dy",
  var_names = c("x", "y")
)
out$rate
out$rates
```

`riemann_prisms3d` *Riemann rectangular prisms over a planar region*

Description

Approximates the double integral of a scalar function over a planar region using an N-by-M rectangular partition and rectangular prisms with constant height on each cell. The region is defined by an x-interval and two functions giving the lower and upper y-limits. Each valid cell produces a prism whose height corresponds to a chosen estimate of the function on that cell: lower value, upper value, or mean value.

When `plot = TRUE`, a 3D visualization of the prisms is produced using **plotly**. Optionally, the actual surface `z = F(x, y)` can also be drawn over the rectangular bounding box that contains the region.

Usage

```
riemann_prisms3d(
  f,
  f1,
  f2,
  a,
  b,
  N,
  M,
  plot = TRUE,
  estimate = c("lower", "upper", "mean", "all"),
  sample_n = 6,
  show_surface = FALSE,
  surface_colorscale = "Viridis",
  surface_opacity = 0.35,
  show_surface_grid = TRUE,
  surface_grid_color = "rgba(60,80,200,0.25)",
  surface_grid_width = 1,
  color_by = c("mean", "lower", "upper"),
  top_colorscale = "YlOrBr",
  top_opacity = 0.85,
  side_color = "rgba(60,60,60,0.25)",
  side_opacity = 0.35,
  frame_color = "rgba(0,0,0,0.55)",
  frame_width = 1.5,
 scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
    zaxis = list(title = "z")),
  bg = list(paper = "white", plot = "white")
)
```

Arguments

`f`	A function `f(x, y)` returning a numeric scalar.
`f1, f2`	Functions returning the lower and upper y-boundaries for each x. The valid y-range at each x is the interval between the minimum and maximum of these two functions.
`a, b`	Numeric endpoints of the x-interval. Must satisfy `a < b`.
`N, M`	Integer numbers of subdivisions in x and y for the rectangular partition.
`plot`	Logical. If `TRUE`, the 3D visualization is generated.
`estimate`	Character. One of `"lower"`, `"upper"`, `"mean"`, or `"all"`, indicating which estimate to highlight.
`sample_n`	Integer. Number of evaluation points per direction inside each cell when computing lower, upper, and mean values.
`show_surface`	Logical. If `TRUE`, draws the surface `z = F(x, y)` over the entire rectangular bounding box.
`surface_colorscale`	Colorscale used for the surface.
`surface_opacity`	Opacity for the surface.
`show_surface_grid`	Logical. If `TRUE`, draws a grid on the surface.
`surface_grid_color`	Color of the grid lines.
`surface_grid_width`	Width of the grid lines.
`color_by`	Character. Determines the value used to color the top of each prism: `"mean"`, `"lower"`, or `"upper"`.
`top_colorscale`	Colorscale for the prism tops.
`top_opacity`	Opacity of the prism tops.
`side_color`	Color for the vertical faces of the prisms.

side_opacity	Opacity of the prism sides.
frame_color	Color for the prism frame lines.
frame_width	Width of the frame lines.
scene	A list with plotly 3D scene settings.
bg	A list with background colors for the figure.

Value

A list containing:

`sum_lower` Lower Riemann sum for the chosen partition.

`sum_upper` Upper Riemann sum.

`sum_mean` Mean-value Riemann sum.

`cells` A tibble describing all valid cells.

`fig` A **plotly** object if `plot = TRUE`, otherwise `NULL`.

`estimate` When `estimate != "all"`, the selected Riemann sum (lower/upper/mean) is repeated here for convenience.

Examples

```
f  <- function(x, y) x * y
f1 <- function(x) 0
f2 <- function(x) 1 - x
riemann_prisms3d(f, f1, f2, 0, 1, N = 10, M = 10, plot = FALSE)
```

---

riemann_rectangles2d *Animate Riemann rectangles under a curve (2D)*

Description

Builds an interactive Plotly animation of Riemann sums approximating the area under a function on a closed interval.

Usage

```
riemann_rectangles2d(
  f,
  a,
  b,
  n_vals = NULL,
  method = c("midpoint", "left", "right"),
  n_curve = 400L,
  frame_ms = 900L,
  transition_ms = 0L,
  title = NULL,
  show_sum = TRUE,
  y0 = 0
)
```

Arguments

`f`	Function. A real-valued function. It must accept a numeric vector and return a numeric vector of the same length.
`a`	Numeric scalar. Left endpoint.
`b`	Numeric scalar. Right endpoint. Must satisfy b > a.
`n_vals`	Integer vector. Values of the number of subintervals used as animation frames. If NULL, a default increasing sequence is used.
`method`	Character. Rule used for rectangle heights: "midpoint" (default), "left", or "right".
`n_curve`	Integer. Number of points used to draw the base curve.
`frame_ms`	Integer. Frame duration in milliseconds.
`transition_ms`	Integer. Transition duration in milliseconds.
`title`	Character. Plot title. If NULL, a default title is used.
`show_sum`	Logical. If TRUE, show n and the value of the Riemann sum in hover text.
`y0`	Numeric scalar. Baseline for rectangles (default 0).

Value

A plotly object (htmlwidget) with animation frames.

Examples

```
library(plotly)
f <- function(x) x^2
riemann_rectangles2d(f, 0, 1)
```

`riemann_sum_1d_plot` *1D Riemann sums with optional plot*

Description

Computes lower, upper, and midpoint Riemann sums for a scalar function `f(x)` on an interval `[xmin, xmax]`. Optionally draws a 2D plot with rectangles and, if requested, the true curve.

Usage

```
riemann_sum_1d_plot(
  f,
  xlim,
  n = 12L,
  methods = c("lower", "upper", "mid"),
  show_curve = TRUE,
  curve_res = 400L,
  colors = list(lower = "#a1d99b", upper = "#fc9272", mid = "#9ecae1"),
  alpha = 0.8,
  edge_color = "black",
  edge_width = 1.2,
  curve_color = "black",
  curve_width = 2,
  show_baseline = TRUE,
  baseline = 0,
  baseline_color = "gray50",
  baseline_width = 1,
  warn_heavy = TRUE
)
```

Arguments

`f`	Function `function(x)` returning numeric values.
`xlim`	Numeric vector `c(xmin, xmax)` with `xmax` > `xmin`.
`n`	Integer. Number of subintervals.
`methods`	Character vector with any of `"lower"`, `"upper"`, `"mid"` indicating which rectangle types to draw.
`show_curve`	Logical. If `TRUE`, overlays the curve `f(x)`.
`curve_res`	Integer. Number of points used to draw the curve.
`colors`	Named list specifying fill colors for `list(lower=..., upper=..., mid=...)`.
`alpha`	Numeric in `[0,1]`. Fill opacity for rectangles.
`edge_color`	Color for rectangle borders.
`edge_width`	Border width.
`curve_color`	Color for the curve.
`curve_width`	Line width for the curve.
`show_baseline`	Logical. If `TRUE`, draws a horizontal baseline.
`baseline`	Numeric. Y-value for the baseline.
`baseline_color`	Baseline color.
`baseline_width`	Baseline width.
`warn_heavy`	Logical. If `TRUE`, warns when `n` is very large.

Value

A list with components:

- `lower_sum` Lower Riemann sum.
- `upper_sum` Upper Riemann sum.
- `mid_sum` Midpoint Riemann sum.
- `dx` Subinterval width.
- `x_breaks` Partition points.
- `figure` A plotly object, or `NULL` if not available.

Examples

```
f <- function(x) sin(2*x)
out <- riemann_sum_1d_plot(
  f, xlim = c(0, pi), n = 10,
  methods = c("lower","upper","mid"),
  show_curve = TRUE
)
out$mid_sum
```

`riemann_sum_2d_plot` *2D Riemann sums (upper, lower, midpoint) with a 3D plot*

Description

Visualize 2D Riemann sums for a scalar field $f(x, y)$ over a rectangular domain $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$. The function computes upper, lower, and midpoint sums and renders a 3D figure showing step tiles for the selected methods. Optionally overlays the true surface $z = f(x, y)$ and a base grid on the xy-plane.

Usage

```
riemann_sum_2d_plot(
  f,
  xlim,
  ylim,
  nx = 8,
  ny = 8,
  methods = c("lower", "upper", "mid"),
  show_surface = TRUE,
  surface_res = c(60L, 60L),
  surface_colorscale = "Viridis",
  surface_opacity = 0.5,
  base_plane = TRUE,
  z0 = 0,
  base_opacity = 0.15,
  base_color = "lightgray",
  show_base_grid = TRUE,
  grid_color = "gray50",
  grid_width = 1,
  tile_opacity = 0.92,
  colors = list(lower = "#a1d99b", upper = "#fc9272", mid = "#9ecae1"),
  edge_color = "black",
  edge_width = 1.2,
  warn_heavy = TRUE
)
```

Arguments

`f`	`function(x, y)` returning a numeric scalar $f(x, y)$.
`xlim`, `ylim`	Numeric length-2 vectors `c(min, max)` for the domain.
`nx`, `ny`	Positive integers: number of subintervals along x and y.
`methods`	Character vector with any of `c("lower","upper","mid")`. Controls which Riemann tiles are drawn. (All estimates are returned.)
`show_surface`	Logical; if `TRUE` overlays the true surface $z = f(x, y)$.
`surface_res`	Integer vector `c(nx_s, ny_s)` for the surface mesh.
`surface_colorscale`	Plotly colorscale for the true surface (e.g. `"Viridis"`).
`surface_opacity`	Opacity of the true surface (0–1).
`base_plane`	Logical; if `TRUE` draws a faint base plane at `z0`.
`z0`	Numeric; height of the base plane (typically 0).
`base_opacity`	Opacity of the base plane (0–1).
`base_color`	Color of the base plane.
`show_base_grid`	Logical; if `TRUE` draws partition grid lines on the base plane.
`grid_color`	Color of the base grid lines.
`grid_width`	Line width of the base grid.
`tile_opacity`	Opacity of Riemann tiles (0–1).
`colors`	Named list for tile colors (hex or rgba), e.g.: `list(lower="#a1d99b", upper="#fc9272", mid="#9ecae1")`.
`edge_color`	Edge color for vertical edges of tiles.
`edge_width`	Line width for tile edges.
`warn_heavy`	Logical; if `TRUE`, warns when `nx*ny` is large.

Details

Upper/lower tiles use corner samples (minimum/maximum of the four corners), and the midpoint tiles sample at the cell center.

Value

A list with:

- `lower_sum`, `upper_sum`, `mid_sum`: numeric estimates,
- `dx`, `dy`: partition widths,
- `breaks`: list with `x_breaks`, `y_breaks`,
- `figure`: the **plotly** object (or `NULL` if **plotly** is not available).

Examples

```
f <- function(x, y) exp(-(x^2 + y^2)) * (1 + 0.3 * cos(3*x) * sin(2*y))
out <- riemann_sum_2d_plot(
  f, xlim = c(-2, 2), ylim = c(-2, 2),
  nx = 8, ny = 7, methods = c("lower","mid","upper"),
  show_surface = TRUE, surface_res = c(80, 80),
  surface_colorscale = "YlGnBu", surface_opacity = 0.45
)
out$lower_sum; out$mid_sum; out$upper_sum
```

`secant_tangent` *Secant lines converge to the tangent line (Plotly)*

Description

Approximates the derivative of a function at a point numerically and builds an interactive Plotly animation showing how secant (incremental quotient) lines converge to the tangent line as the step size decreases. The secant point(s) used for the slope computation are also animated.

Usage

```
secant_tangent(
  f,
  x0,
  h_vals = NULL,
  method = c("forward", "central"),
  xlim = NULL,
  n_curve = 400L,
  frame_ms = 220L,
  transition_ms = 220L,
  title = NULL,
  safe_mode = TRUE
)
```

Arguments

`f`	Function. A real-valued function f(x). It must accept a numeric vector and return a numeric vector of the same length.
`x0`	Numeric scalar. Point where the derivative is approximated.
`h_vals`	Numeric vector. Positive step sizes used as animation frames. If NULL, a default decreasing sequence is used.
`method`	Character. Derivative approximation method: "forward" (default) or "central".
`xlim`	Numeric vector of length 2. Plot range for x. If NULL, it is chosen automatically from x0 and h_vals.
`n_curve`	Integer. Number of points used to draw the curve and lines.
`frame_ms`	Integer. Frame duration in milliseconds.
`transition_ms`	Integer. Transition duration in milliseconds.
`title`	Character. Plot title. If NULL, a default title is used.
`safe_mode`	Logical. If TRUE, use calmer animation defaults intended to reduce flicker and visual stress.

Details

The forward incremental quotient is

$$m_h = \frac{f(x_0 + h) - f(x_0)}{h}.$$

The central difference approximation is

$$m_h = \frac{f(x_0 + h) - f(x_0 - h)}{2h}.$$

The tangent line model at x0 is

$$y = f(x_0) + f'(x_0)\,(x - x_0).$$

Value

A list with components:

plot A plotly object (htmlwidget) with animation frames.

derivative Numeric scalar. Derivative estimate using the smallest h.

data Data frame used for the animated secant lines (useful for debugging).

Examples

```
library(plotly)

f <- function(x) x^2
out <- secant_tangent(f, x0 = 1)
out$plot
out$derivative
```

```
g <- function(x) sin(x)
secant_tangent(g, x0 = 0.7, method = "central", h_vals = 2^(-(1:7)))
```

`solid_cylindrical3d` *Cylindrical solid defined by radial and vertical bounds (with optional plot)*

Description

Builds and optionally plots, using **plotly**, a three-dimensional solid described in cylindrical coordinates. The solid is defined by:

- an angular variable `theta` in the interval `[th_min, th_max]`,
- a radial variable `r` between `R1(theta)` and `R2(theta)`,
- and a vertical coordinate `z` between `Z1(r, theta)` and `Z2(r, theta)`.

The surface is rendered by sampling a curvilinear grid in the parameters `(theta, u, v)`, where `u` and `v` vary in `[0, 1]` and are used as linear blending variables along the radial and vertical directions, respectively.

When volume computation is requested, the function numerically approximates the triple integral of the form `integral theta` from `th_min` to `th_max` of `integral r` from `R1(theta)` to `R2(theta)` of `integral z` from `Z1(r, theta)` to `Z2(r, theta)` of `r dz dr dtheta`, which is the standard volume element in cylindrical coordinates.

Usage

```
solid_cylindrical3d(
  R1,
  R2,
  Z1,
  Z2,
  th_min,
  th_max,
  plot = TRUE,
  n_theta = 160,
  n_u = 70,
  n_v = 70,
  mode = c("faces", "wireframe", "both"),
  colorscale = "Blues",
  opacity = 0.35,
  show_surface_grid = TRUE,
  surface_grid_color = "rgba(60,80,200,0.25)",
  surface_grid_width = 1,
  edge_line = list(color = "black", width = 2),
  wire_line = list(color = "rgba(0,0,0,0.35)", width = 1),
 scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
   zaxis = list(title = "z")),
  bg = list(paper = "white", plot = "white"),
  compute_volume = FALSE,
  vol_method = c("adaptive", "grid"),
  ntheta_vol = 400,
  nr_vol = 400
)
```

Arguments

R1, R2	Functions `function(theta)` giving the inner and outer radius bounds, respectively.
Z1, Z2	Functions `function(r, theta)` giving the lower and upper z bounds.
th_min, th_max	Angular limits (numeric scalars) with `th_max > th_min`.
plot	Logical. If `TRUE`, the solid is plotted using **plotly**.
n_theta, n_u, n_v	Mesh resolution in `theta` (angle), `u` (radial blend) and `v` (vertical blend).
mode	Character string, one of `"faces"`, `"wireframe"` or `"both"`, indicating whether to draw only the surface, only a wireframe or both.
colorscale	Plotly colorscale for the surface. It can be a named scale, a single color, or a character vector of colors interpreted as a gradient.
opacity	Surface opacity, a numeric value between 0 and 1.
show_surface_grid	Logical. If `TRUE`, draws a grid over the surface.
surface_grid_color, surface_grid_width	Color and line width used for the surface grid.
edge_line, wire_line	Line style lists used for the edges and the wireframe lines when those are drawn.
scene, bg	Plotly 3D scene configuration and background colors. The background list typically has entries `paper` and `plot`.
compute_volume	Logical. If `TRUE`, the volume of the solid is approximated numerically.
vol_method	Character string. Either `"adaptive"`, which uses nested `stats::integrate`, or `"grid"`, which uses the trapezoidal rule over a regular mesh.
ntheta_vol, nr_vol	Mesh sizes in the angular and radial directions used when `vol_method = "grid"`.

Value

A list with components:

- `theta_seq`, `u_seq`, `v_seq`: the parameter sequences used for sampling the surface,
- `fig`: a **plotly** object when `plot = TRUE`, or `NULL` otherwise,
- `volume`: `NULL` if `compute_volume = FALSE`, or a list containing the numeric volume estimate and metadata (method and grid parameters) when `compute_volume = TRUE`.

Examples

```
# Example: a quarter-twisted cup
# R in [0, 1 + 0.2*cos(theta)],  z in [0, 1 + 0.5*r]
R1 <- function(theta) 0
R2 <- function(theta) 1 + 0.2*cos(theta)
Z1 <- function(r, theta) 0
Z2 <- function(r, theta) 1 + 0.5*r
solid_cylindrical3d(
  R1, R2, Z1, Z2, th_min = 0, th_max = pi/2,
  plot = FALSE, mode = "both",
  colorscale = c("white", "#2a9d8f"), opacity = 0.35,
  show_surface_grid = TRUE,
  compute_volume = TRUE, vol_method = "adaptive"
)$volume
```

---

solid_of_revolution_y *Solid of revolution around a horizontal line*

Description

Construct a three-dimensional surface for the solid obtained by rotating the graph of a function `f(x)` around the line `y = a` on a finite interval, and compute its volume and surface areas.

Usage

```
solid_of_revolution_y(
  f,
  xlim,
  a,
  nx = 120L,
  nt = 120L,
  deriv = NULL,
  h = NULL,
  include_end_caps = FALSE,
  plot = TRUE,
  colors = list(surface = "steelblue", axis = "black", curve = "firebrick"),
  opacity = 0.9,
  show_axis = TRUE,
  show_profile_curve = TRUE,
  curve_thetas = 0,
  curve_width = 4,
  curve_opacity = 1,
 scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
    zaxis = list(title = "z"))
)
```

Arguments

`f`	Function of one numeric argument `x` that returns a numeric value.
`xlim`	Numeric vector of length two with the limits of the `x` interval `c(xmin, xmax)`, with `xmax` > `xmin`.
`a`	Numeric scalar that gives the horizontal axis of rotation.
`nx`	Integer number of grid points along the `x` direction (for plotting).
`nt`	Integer number of grid points along the angular direction (for plotting).
`deriv`	Optional function that returns the derivative of `f`. If `NULL` a numeric derivative is used.
`h`	Optional numeric step for the numeric derivative.
`include_end_caps`	Logical value. If `TRUE` the area of the circular cross sections at the ends of the interval is added to the total area.
`plot`	Logical; if `TRUE` and **plotly** is available, the solid is drawn.
`colors`	List with optional entries `surface`, `axis` and `curve` that control the colours used in the plot.
`opacity`	Numeric value between 0 and 1 that controls the surface opacity in the plot.
`show_axis`	Logical value indicating whether the axis of rotation is drawn.
`show_profile_curve`	Logical value indicating whether the generating curve is drawn on the surface.
`curve_thetas`	Numeric vector of angles (in radians) where profile curves are drawn.
`curve_width`	Numeric line width for the profile curves.
`curve_opacity`	Numeric value between 0 and 1 for the profile curves.
`scene`	List of **plotly** scene options used in `plotly::layout()`.

Value

A list with components:

- `volume`: numeric value of the volume.
- `surface_area_lateral`: numeric value of the lateral area.
- `surface_area_total`: numeric value of the total area.
- `figure`: **plotly** object with the three-dimensional plot if `plot` = `TRUE` and **plotly** is available; otherwise `NULL`.

Examples

```
f <- function(x) sqrt(x)
solid_of_revolution_y(f, xlim = c(0, 4), a = 0, plot = FALSE)
```

solid_spherical3d *Solid in spherical coordinates with Plotly visualization and volume*

Description

Draws a three-dimensional solid described in spherical coordinates by:

- a radial variable r between R1(theta, phi) and R2(theta, phi),
- an azimuthal angle theta in the interval [theta_range[1], theta_range[2]] (in radians),
- a polar angle phi in the interval [phi_range[1], phi_range[2]] (in radians).

The function uses the standard convention for spherical coordinates: theta is the azimuth (angle in the xy-plane) and phi is the polar angle measured from the positive z-axis.

Optionally, the volume of the solid is computed using the spherical volume element. The exact integral has the form:

- inner integral: from r = R1(theta, phi) to r = R2(theta, phi) of r^2 * sin(phi) dr,
- outer integrals: over phi in [phi_min, phi_max] and theta in [theta_min, theta_max].

Equivalently, for each pair (theta, phi) one integrates (R2(theta, phi)^3 - R1(theta, phi)^3) / 3 * sin(phi) over the angular rectangle.

Usage

```
solid_spherical3d(
  R1,
  R2,
  theta_range = c(0, 2 * pi),
  phi_range = c(0, pi),
  n_theta = 160,
  n_phi = 120,
  plot = TRUE,
  show_surfaces = c(TRUE, TRUE),
  colorscales = list("Blues", "Reds"),
  opacities = c(0.3, 0.35),
  show_surface_grid = TRUE,
  surface_grid_color = "rgba(60,80,200,0.25)",
  surface_grid_width = 1,
 scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
    zaxis = list(title = "z")),
  bg = list(paper = "white", plot = "white"),
  compute_volume = FALSE,
  vol_method = c("adaptive", "grid"),
  n_th_vol = 600,
  n_ph_vol = 400
)
```

Arguments

`R1, R2`	Functions `function(theta, phi)` giving the inner and outer radius, respectively, as numeric scalars.
`theta_range`	Numeric vector of length 2, `c(theta_min, theta_max)`, giving the azimuth interval in radians.
`phi_range`	Numeric vector of length 2, `c(phi_min, phi_max)`, giving the polar angle interval in radians.
`n_theta, n_phi`	Mesh resolution for the two boundary surfaces. Each surface is sampled on an `n_phi x n_theta` grid.
`plot`	Logical. If `TRUE`, the solid boundaries are drawn with **plotly**.
`show_surfaces`	Logical vector of length 2 indicating which spherical shells to show in the plot, in the order `c(r = R1, r = R2)`.
`colorscales`	Colorscales for the two surfaces. You can pass: • a single Plotly colorscale (string, single color, or vector of colors) applied to both surfaces, or • a list of length 2, with one colorscale per surface. Flat colors such as `"#2a9d8f"` or `"rgba(0,0,0,0.6)"` are also accepted.
`opacities`	Numeric vector of length 1 or 2 giving the opacity of the two surfaces.
`show_surface_grid`	Logical. If `TRUE`, draws grid lines on the surfaces.
`surface_grid_color, surface_grid_width`	Color and width for the surface grid lines.
`scene`	Plotly 3D scene settings. By default the aspect mode is `"data"` and each axis has a title.
`bg`	Background colors, typically a list of the form `list(paper = "white", plot = "white")`.
`compute_volume`	Logical. If `TRUE`, the volume of the solid is approximated numerically using the spherical volume integral.
`vol_method`	Character string indicating the integration method for the volume: `"adaptive"` uses nested `stats::integrate`, while `"grid"` uses the trapezoidal rule on a regular grid.
`n_th_vol, n_ph_vol`	Integer resolutions for the grid method in the azimuth and polar directions, respectively (ignored when `vol_method = "adaptive"`).

Value

A list with:

- `theta_seq`, `phi_seq`: the parameter sequences used for plotting,
- `R1_surf`, `R2_surf`: lists containing the matrices `X`, `Y`, `Z` for the two boundary surfaces (or `NULL` if the corresponding surface is not shown),
- `fig`: a **plotly** figure when `plot = TRUE`, otherwise `NULL`,
- `volume`: `NULL` if `compute_volume = FALSE`, or a list with the numeric volume estimate, the method used and additional metadata when `compute_volume = TRUE`.

Examples

```
# Example 1: Spherical shell: a <= r <= b, independent of angles
R1 <- function(th, ph) 0.8
R2 <- function(th, ph) 1.2
out <- solid_spherical3d(
  R1, R2,
  theta_range = c(0, 2*pi),
  phi_range   = c(0, pi),
  plot = TRUE,
  colorscales = list("Blues", "Reds"),
  opacities   = c(0.25, 0.35),
  compute_volume = TRUE
)
out$volume$estimate       # approximately 4/3 * pi * (1.2^3 - 0.8^3)

# Example 2: Spherical cap: 0 <= r <= 1, phi in [0, pi/3]
R1 <- function(th, ph) 0
R2 <- function(th, ph) 1
out2 <- solid_spherical3d(
  R1, R2,
  theta_range = c(0, 2*pi),
  phi_range   = c(0, pi/3),
  plot = TRUE,
  compute_volume = TRUE
)
out2$volume$estimate      # analytic value matches the standard spherical cap formula
```

`solid_xyz3d` *Solid defined by bounds in x, y and z*

Description

Constructs a three-dimensional solid defined by bounds in the variables `x`, `y` y `z`, and optionally renders it using **plotly**. The solid is described by:

- an interval for `x` between `a` y `b`,
- lower and upper functions in the `y` direction,
- lower and upper functions in the `z` direction that may depend on both `x` y `y`.

The function uses a curvilinear-prism parametrization to build meshes for the six faces of the solid. It supports different display modes (faces, wireframe, or both), optional numerical volume computation, and internal slices on coordinate planes (slice arguments are reserved for future extensions and are currently ignored).

Usage

```
solid_xyz3d(
  H1,
  H2,
  G1,
  G2,
  a,
  b,
  plot = TRUE,
  n_x = 120,
  n_u = 60,
  n_v = 60,
  mode = c("faces", "wireframe", "both"),
  show_faces = c(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE),
  colorscales = c("Blues", "Blues", "Greens", "Greens", "Reds", "Reds"),
  opacities = 0.35,
  show_surface_grid = TRUE,
  surface_grid_color = "rgba(60,80,200,0.25)",
  surface_grid_width = 1,
  show_edges = TRUE,
  edge_line = list(color = "black", width = 2),
  wire_step = 6,
  wire_line = list(color = "black", width = 1),
 scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
    zaxis = list(title = "z")),
  bg = list(paper = "white", plot = "white"),
  compute_volume = FALSE,
  vol_method = c("adaptive", "grid"),
  nx_vol = 300,
  ny_vol = 300,
  slice = list(x = NULL, y = NULL, z = NULL),
  slice_mode = c("surface", "wireframe", "both"),
  slice_nx = 200,
  slice_nu = 120,
  slice_nv = 120,
  slice_colorscales = list(x = "Oranges", y = "Purples", z = "Greens"),
  slice_opacity = 0.55,
  slice_show_grid = TRUE,
  slice_grid_color = "rgba(80,80,80,0.25)",
  slice_grid_width = 1,
  slice_wire_step = 8,
  slice_wire_line = list(color = "black", width = 2, dash = "dot")
)
```

Arguments

`H1, H2`	Functions of one variable x giving the lower and upper bounds in the y direction.
`G1, G2`	Functions of two variables x y y giving the lower and upper bounds in the z

	direction.
`a`, `b`	Numeric endpoints of the interval for `x`. It is assumed that `b > a`.
`plot`	Logical; if `TRUE`, the solid is rendered with **plotly**.
`n_x`, `n_u`, `n_v`	Integers giving the mesh resolution in the principal parameter along `x` and in the two internal parameters of the face meshes.
`mode`	Character string; one of `"faces"`, `"wireframe"` or `"both"`, indicating whether to draw surfaces, wireframe, or a combination of both.
`show_faces`	Logical vector indicating which of the six faces to display. The order is `c("x=a","x=b","y=H1","y=H2",` A single logical value is also allowed and will be recycled.
`colorscales`	Color specification for faces. It can be: • a single **plotly** colorscale name applied to all faces, • a single flat color (R color name, hexadecimal code, or `"rgba(...)"`), • a vector of colors that define a gradient, • or a list or vector of length six, assigning a scale or color specification to each face separately.
`opacities`	Numeric values controlling face opacity; may be a single value or a vector of length six.
`show_surface_grid`	Logical; if `TRUE`, draws grid lines on the faces.
`surface_grid_color`, `surface_grid_width`	Color and width for surface grid lines.
`show_edges`	Logical; if `TRUE`, draws the edges of each face.
`edge_line`	List with style options for edges (for example, color, width and dash pattern).
`wire_step`	Integer greater or equal to one; controls how many mesh lines are skipped between wireframe lines.
`wire_line`	List with style options for wireframe lines.
`scene`	List with 3D scene options for **plotly**. By default, an aspect ratio based on the data is used.
`bg`	List specifying background colors for the figure, typically with entries `paper` and `plot`.
`compute_volume`	Logical; if `TRUE`, computes an approximate volume of the solid.
`vol_method`	Character string selecting the volume integration method: `"adaptive"` for nested calls to `stats::integrate`, or `"grid"` for a trapezoidal rule on a regular grid.
`nx_vol`, `ny_vol`	Integer grid sizes used when `vol_method = "grid"`.
`slice`	List describing slices to be drawn, with components `x`, `y` `y` `z`. Each component can be `NULL`, a single numeric value or a numeric vector of slice positions. (Reserved for future use; currently ignored.)
`slice_mode`	Character string indicating how to render slices: `"surface"`, `"wireframe"` or `"both"`. (Reserved for future use; currently ignored.)
`slice_nx`, `slice_nu`, `slice_nv`	Mesh resolutions used to build the slices. (Reserved for future use; currently ignored.)

`slice_colorscales`
: List with color scales for slices in the `x`, `y` y `z` directions, in the same formats accepted by `colorscales`. (Reserved for future use; currently ignored.)

`slice_opacity`
: Numeric opacity for slices, between 0 and 1. (Reserved for future use; currently ignored.)

`slice_show_grid`
: Logical; if `TRUE`, draws grid lines on the slices. (Reserved for future use; currently ignored.)

`slice_grid_color`, `slice_grid_width`
: Color and width for slice grid lines. (Reserved for future use; currently ignored.)

`slice_wire_step`
: Integer controlling the spacing of wireframe lines on slices. (Reserved for future use; currently ignored.)

`slice_wire_line`
: List with style options for slice wireframe lines. (Reserved for future use; currently ignored.)

Details

The solid is sampled on a three-parameter grid. Two of the parameters describe the position on the base region in the `x`-`y` plane, and the third parameter interpolates between the lower and upper `z` bounds. From this parametrization the function constructs the six bounding faces, corresponding to the two extreme values of `x`, the two extreme values of `y`, and the two extreme values of `z`.

Rendering options allow:

- drawing only the faces of the solid,
- drawing only a wireframe of the mesh,
- combining both faces and wireframe,
- assigning individual color scales and opacities to each face,
- showing or hiding surface grids and edges.

When internal slices are requested, the intention is to intersect the solid with planes of the form `x = constant`, `y = constant` o `z = constant`. The corresponding slice arguments are reserved for future versions of the function and are not yet implemented.

If `compute_volume = TRUE`, the function also computes an approximate volume of the solid using either:

- a nested adaptive integration based on `stats::integrate`,
- or a trapezoidal rule on a regular grid in the `x` y `y` directions.

Value

A list with:

- `x_seq`, `u_seq`, `v_seq`: the parameter sequences used to build the mesh,
- `fig`: a **plotly** object when `plot = TRUE`, otherwise `NULL`,
- `volume`: either `NULL` or a list with an approximate volume estimate and related metadata when `compute_volume = TRUE`.

Examples

```
# Note: examples avoid plotting for CRAN checks
H1 <- function(x) -1 - x
H2 <- function(x)  1 - x^2
G1 <- function(x, y) y
G2 <- function(x, y) y + 1
s <- solid_xyz3d(
  H1, H2, G1, G2,
  a = -1, b = 1,
  plot = FALSE,
  compute_volume = TRUE,
  vol_method = "grid",
  nx_vol = 50, ny_vol = 50
)
s$volume$estimate
```

`streamline_and_field3d` *Vector field and streamline in 3D (single combined figure)*

Description

Draws a three-dimensional vector field inside a curvilinear volume and overlays a streamline that follows the field, all in a single **plotly** figure. The streamline is obtained by integrating an ordinary differential equation using a fixed-step Runge-Kutta method of order four (RK4), starting from an initial point.

Usage

```
streamline_and_field3d(
  field,
  H1,
  H2,
  G1,
  G2,
  a,
  b,
  NX = 8,
  NY = 6,
  NZ = 6,
  p,
  t_final,
  step,
  arrows = c("both", "line", "cone", "none"),
  arrow_scale = 0.08,
  normalize_bias = 1,
  normal_color = "rgba(0,0,0,0.55)",
  normal_width = 2,
  arrow_color = "#1d3557",
  arrow_opacity = 0.95,
  arrow_size = 0.35,
  traj_color = "#e63946",
  traj_width = 5,
  traj_markers = TRUE,
  traj_marker_size = 2,
 scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
    zaxis = list(title = "z")),
  bg = list(paper = "white", plot = "white"),
  ...
)
```

Arguments

`field`	A function representing the vector field. It can be given as `function(x, y, z)` or `function(x, y, z, t)`, and must return a numeric vector of length three `c(Fx, Fy, Fz)`.
`H1, H2`	Functions of one variable `x` giving the lower and upper bounds in the `y` direction.
`G1, G2`	Functions of two variables `x` and `y` giving the lower and upper bounds in the `z` direction.
`a, b`	Numeric endpoints of the interval for `x`. It is assumed that `b > a`.
`NX, NY, NZ`	Integers greater than or equal to one specifying the sampling density of the field in the three parameter directions.
`p`	Numeric vector of length three giving the initial point of the streamline, in the form `c(x0, y0, z0)`.
`t_final`	Final integration time for the streamline. A negative value integrates backward in time.
`step`	Step size for the fixed-step RK4 integration. Must be strictly positive.
`arrows`	Character string indicating the arrow mode. Allowed values are `"none"`, `"line"`, `"cone"` and `"both"`.
`arrow_scale`	Global arrow length scale, expressed as a fraction of the largest span of the bounding box.
`normalize_bias`	Numeric saturation bias used in the scaling of the vector norm. Larger values make arrow lengths saturate earlier.
`normal_color`	Color of the arrow shafts (line segments).
`normal_width`	Numeric width of the arrow shafts.
`arrow_color`	Color of the arrow heads (cones or chevrons).
`arrow_opacity`	Opacity of the arrow heads.
`arrow_size`	Relative size of the arrow heads with respect to `arrow_scale`.
`traj_color`	Color of the streamline.
`traj_width`	Width of the streamline.

`traj_markers`	Logical; if `TRUE`, draws markers along the streamline.
`traj_marker_size`	Size of the markers drawn on the streamline.
`scene`	List with 3D scene settings for **plotly**, such as aspect mode and axis titles.
`bg`	List with background colors for the figure, typically with entries `paper` and `plot`.
`...`	Reserved for backward compatibility. Do not use.

Details

The volume is defined by:

- an interval for `x` between `a` and `b`,
- lower and upper functions for `y` that depend on `x`,
- lower and upper functions for `z` that may depend on both `x` and `y`.

The function builds a regular grid in three parameters and maps each grid point to the physical coordinates `(x, y, z)` using linear blends between the corresponding bounds. The vector field is then evaluated at each of these points to obtain the base positions and vectors used to draw the arrows.

Arrow lengths are scaled using a saturated version of the vector norm. This means that small magnitudes produce short arrows, whereas very large magnitudes are limited by a bias parameter so that they do not dominate the entire plot. A global scale factor controls the typical arrow length relative to the size of the domain.

The streamline is defined as the trajectory of a particle whose velocity at each point is given by the vector field evaluated along the path. The field may optionally depend on time; if the function `field` has a time argument, it is used during integration. The integration runs from time zero up to a final time, with positive or negative direction depending on the sign of the final time.

The resulting figure combines:

- a set of arrows representing the vector field,
- a space curve representing the streamline,
- optional markers along the streamline and highlighted start and end points.

Value

A list with:

- `field_points`: a data frame with base positions `x`, `y`, `z` and the magnitude of the field at each point,
- `field_segments`: a data frame with columns `x0`, `y0`, `z0`, `x1`, `y1`, `z1` describing the arrow shafts,
- `traj`: a data frame with the streamline data, including time, coordinates and local speed,
- `fig`: a **plotly** object containing the combined field and streamline visualization.

Examples

```
H1 <- function(x) -1
H2 <- function(x)  1
G1 <- function(x, y) -0.5
G2 <- function(x, y)  0.5
field <- function(x, y, z) c(-y, x, 0.6)

out <- streamline_and_field3d(
  field, H1, H2, G1, G2,
  a = -2, b = 2, NX = 10, NY = 8, NZ = 5,
  p = c(1, 0, 0), t_final = 2, step = 0.05,
  arrows = "both", arrow_scale = 0.12, normalize_bias = 1,
  normal_color = "rgba(0,0,0,0.55)", normal_width = 2,
  arrow_color  = "#1d3557", arrow_opacity = 0.95, arrow_size = 0.4,
  traj_color   = "#e63946", traj_width = 5, traj_markers = TRUE
)
```

`surface_integral_z` *Surface integral over a graph z = g(x, y)*

Description

Computes numeric approximations of surface integrals over a surface given in graph form `z = g(x, y)` on a rectangular domain in the x-y plane.

Usage

```
surface_integral_z(
  gfun,
  xlim,
  ylim,
  nx = 160,
  ny = 160,
  scalar_phi = NULL,
  vector_F = NULL,
  orientation = c("up", "down"),
  plot = TRUE,
  title = "Surface integral over z = g(x,y)"
)
```

Arguments

gfun	Function of two variables `function(x, y)` returning the height `z = g(x, y)` of the surface.
xlim	Numeric vector of length 2 giving the range for x, `c(x_min, x_max)` with `x_max` > `x_min`.

`ylim`	Numeric vector of length 2 giving the range for y, `c(y_min, y_max)` with `y_max > y_min`.
`nx`	Integer, number of grid points in the x direction (recommended: at least 20).
`ny`	Integer, number of grid points in the y direction (recommended: at least 20).
`scalar_phi`	Optional scalar field `function(x, y, z)`. If provided, the function computes the scalar surface integral of `scalar_phi` over the surface.
`vector_F`	Optional vector field `function(x, y, z)` that returns a numeric vector of length 3 `c(Fx, Fy, Fz)`. If provided, the function computes the flux integral of `vector_F` across the oriented surface.
`orientation`	Character string indicating the orientation of the normal vector, either `"up"` or `"down"`. This affects the sign of the flux integral.
`plot`	Logical. If `TRUE`, returns a **plotly** surface plot colored by the available integrand (scalar field times area density or flux density).
`title`	Character string used as the base title for the plot.

Details

Two types of integrals can be computed:

1. A scalar surface integral of the form `Integral_S phi dS`, where `phi(x, y, z)` is a scalar field evaluated on the surface.
2. A flux (vector surface integral) of the form `Integral_S F dot n dS`, where `F(x, y, z)` is a vector field and `n` is the chosen unit normal direction.

The surface is parametrized by `(x, y) -> (x, y, g(x, y))` over a rectangular domain given by `xlim` and `ylim`. Partial derivatives of g with respect to x and y are approximated by finite differences on a uniform grid, and the integrals are computed using a composite trapezoid rule on that grid.

Value

A list with components:

- `area_density_integral`: numeric scalar with the value of the scalar surface integral (or `NA` if `scalar_phi` is not supplied).
- `flux_integral`: numeric scalar with the value of the flux integral (or `NA` if `vector_F` is not supplied).
- `plot`: a **plotly** surface object if `plot = TRUE`, otherwise `NULL`.
- `grids`: list with matrices and grid information, including `X`, `Y`, `Z`, partial derivatives, and weights.
- `fields`: list with the scalar and/or flux integrand evaluated on the grid.

Examples

```
# Surface z = x^2 + y^2 on [-1,1] x [-1,1]
gfun <- function(x, y) x^2 + y^2

# Scalar field phi(x,y,z) = 1 (surface area of the patch)
```

```
phi <- function(x, y, z) 1

# Vector field F = (0, 0, 1), flux through the surface
Fvec <- function(x, y, z) c(0, 0, 1)

res <- surface_integral_z(
  gfun,
  xlim = c(-1, 1),
  ylim = c(-1, 1),
  nx = 60, ny = 60,
  scalar_phi = phi,
  vector_F = Fvec,
  orientation = "up",
  plot = FALSE
)
res$area_density_integral
res$flux_integral
```

surface_parametric_area *Plot a parametric surface and estimate its area*

Description

This function plots a smooth parametric surface defined by functions x(u,v), y(u,v), and z(u,v). It also estimates the surface area using a trapezoidal approximation based on the magnitudes of partial-derivative cross products.

Usage

```
surface_parametric_area(
  xfun,
  yfun,
  zfun,
  urange = c(0, 2 * pi),
  vrange = c(0, 2 * pi),
  nu = 160,
  nv = 160,
  h_u = NULL,
  h_v = NULL,
  title_prefix = "r(u,v)"
)
```

Arguments

xfun	Function of two arguments (u, v) returning the x-coordinate of the surface.

`yfun`	Function of two arguments (u, v) returning the y-coordinate of the surface.
`zfun`	Function of two arguments (u, v) returning the z-coordinate of the surface.
`urange`	Numeric vector of length 2 giving the interval for the parameter u, `c(u_min, u_max)` with `u_max` > `u_min`.
`vrange`	Numeric vector of length 2 giving the interval for the parameter v, `c(v_min, v_max)` with `v_max` > `v_min`.
`nu`	Integer, number of grid points along u (recommended: at least 20 for a reasonable surface).
`nv`	Integer, number of grid points along v (recommended: at least 20).
`h_u`	Numeric step size for finite differences in u. If `NULL`, a default based on the grid spacing is used.
`h_v`	Numeric step size for finite differences in v. If `NULL`, a default based on the grid spacing is used.
`title_prefix`	Character string used in the plot title.

Details

The parametric domain is given by ranges for u and v, and the surface is evaluated on a regular grid with sizes specified by `nu` and `nv`. Finite differences are used to approximate partial derivatives with respect to u and v.

Value

A list with:

- `plot`: a **plotly** surface object showing the parametric surface.
- `area`: numeric estimate of the surface area.
- `grid`: list with elements `U`, `V`, `X`, `Y`, and `Z`, representing the parameter values and evaluated surface.

Examples

```
# Example: torus-like parametric surface
xfun <- function(u, v) (2 + cos(v)) * cos(u)
yfun <- function(u, v) (2 + cos(v)) * sin(u)
zfun <- function(u, v) sin(v)
result <- surface_parametric_area(
  xfun, yfun, zfun,
  urange = c(0, 2*pi),
  vrange = c(0, 2*pi),
  nu = 80, nv = 80
)
result$area
```

tangent3d — *Unit tangent vectors along a 3D parametric curve*

Description

Computes numerical unit tangent vectors of a three-dimensional parametric curve at selected values of the parameter. The curve is defined by three functions that give its coordinate components. For each evaluation point, the first derivative of the curve is approximated numerically and then normalized to obtain a unit tangent direction.

Usage

```
tangent3d(
  X,
  Y,
  Z,
  a,
  b,
  t_points,
  h = 1e-04,
  plot = FALSE,
  n_samples = 400,
  vec_scale = NULL,
  vec_factor = 1,
  curve_line = list(color = "blue", width = 2, dash = "solid"),
  T_line = list(color = "red", width = 5, dash = "solid"),
  show_curve = TRUE,
  show_points = TRUE,
  point_marker = list(color = "black", size = 3, symbol = "circle"),
 scene = list(aspectmode = "data", xaxis = list(title = "x(t)"), yaxis = list(title =
    "y(t)"), zaxis = list(title = "z(t)")),
  bg = list(paper = "white", plot = "white"),
  tol = 1e-10
)
```

Arguments

X	Function returning the x coordinate of the curve as a function of the parameter t.
Y	Function returning the y coordinate of the curve as a function of the parameter t.
Z	Function returning the z coordinate of the curve as a function of the parameter t.
a	Lower endpoint of the parameter interval.
b	Upper endpoint of the parameter interval.

`t_points`	Numeric vector of parameter values at which the tangent direction is evaluated and optionally plotted.
`h`	Step size for centered finite-difference approximations.
`plot`	Logical; if `TRUE`, shows a 3D **plotly** visualization of the curve and tangent segments.
`n_samples`	Number of points used to sample and display the curve in the 3D plot.
`vec_scale`	Base length used for the tangent segments. If `NULL`, it is estimated as a small fraction of the overall size of the sampled curve.
`vec_factor`	Multiplicative factor applied to `vec_scale` to control the visual size of the tangent segments.
`curve_line`	List with **plotly** style options for drawing the base curve.
`T_line`	List with **plotly** style options for the tangent segments.
`show_curve`	Logical; if `TRUE`, the base curve is included in the plot.
`show_points`	Logical; if `TRUE`, the evaluation points are marked in the plot.
`point_marker`	List with **plotly** marker options for the evaluation points.
`scene`	List with 3D scene settings for **plotly**.
`bg`	Background colors for the figure, usually a list with entries such as `paper` and `plot`.
`tol`	Numeric tolerance for detecting situations in which the first derivative is too small to define a stable tangent direction.

Details

For every element of `t_points`, the function:

- computes a centered finite-difference approximation of the first derivative of the curve,
- evaluates the magnitude of that derivative,
- divides the derivative by its magnitude to obtain a unit vector pointing in the direction of motion of the curve at that point.

If the magnitude of the first derivative is extremely small at a given parameter value, the tangent direction becomes numerically unstable; in such cases, the function returns `NA` for the corresponding components and may emit a diagnostic message.

Optionally, the curve and the associated tangent directions can be shown in an interactive 3D plot using **plotly**. Short line segments representing the tangent direction can be anchored at each evaluation point. The sampled curve, the reference points and the tangent segments may be displayed or hidden independently.

Value

A tibble with columns `t`, `x`, `y`, `z`, `Tx`, `Ty` and `Tz`, where the last three columns contain the components of the unit tangent vector at each evaluation point.

Examples

```
X <- function(t) t*cos(t)
Y <- function(t) t*sin(3*t)
Z <- function(t) t
tangent3d(X, Y, Z, a = 0, b = 2*pi, t_points = c(pi/3, pi, 5*pi/3))
```

`tangent_plane3d` *Tangent plane and normal vector to a surface z = f(x, y)*

Description

Computes the tangent plane to the graph of a scalar field `z = f(x, y)` at a given point `(x0, y0)`, together with the associated normal vector. Optionally, it displays the surface, the tangent plane, two orthogonal cross-sections and the normal vector using **plotly**.

Usage

```
tangent_plane3d(
  f,
  point,
  h = 1e-04,
  plot = FALSE,
  x_window = 4,
  y_window = 4,
  z_window = 4,
  grid = 50,
  plane_window = 1,
  vec_N_factor = 1,
  surface_opacity = 0.85,
  plane_opacity = 0.7,
  colors = list(surface = "Viridis", plane = "Reds", xcut = "black", ycut = "black",
    point = "blue", normal = "red"),
  show_surface_grid = FALSE,
  surface_grid_color = "rgba(0,0,0,0.35)",
  surface_grid_width = 1,
  show_axis_grid = TRUE,
  axis_grid_color = "#e5e5e5",
  axis_grid_width = 1,
  scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
    zaxis = list(title = "z")),
  bg = list(paper = "white", plot = "white")
)
```

Arguments

`f`	Function of two variables `f(x, y)` returning a numeric scalar, representing the height `z`.
`point`	Numeric vector of length 2 giving the point of tangency `c(x0, y0)`.
`h`	Numeric step used in the centered finite-difference approximation of the partial derivatives. Must be strictly positive.
`plot`	Logical; if `TRUE`, constructs a **plotly** figure with the surface, tangent plane, cross-sections and normal vector.
`x_window`	Numeric half-width of the window in the `x` direction used to draw the surface patch.
`y_window`	Numeric half-width of the window in the `y` direction used to draw the surface patch.
`z_window`	Numeric half-height for the visible `z` range around `f(x0, y0)`.
`grid`	Integer number of grid points used to sample the surface in each horizontal direction.
`plane_window`	Numeric half-width of the square patch of the tangent plane drawn around `(x0, y0)`.
`vec_N_factor`	Numeric scale factor applied to the unit normal vector when drawing the segment that represents the normal.
`surface_opacity`	Numeric value between 0 and 1 controlling the opacity of the surface patch.
`plane_opacity`	Numeric value between 0 and 1 controlling the opacity of the tangent-plane patch.
`colors`	List with named entries that control colors in the plot: • `surface`: colorscale for the original surface, • `plane`: colorscale for the tangent plane, • `xcut`: color for the intersection curve along `y = y0`, • `ycut`: color for the intersection curve along `x = x0`, • `point`: color for the tangency point marker, • `normal`: color for the normal vector segment.
`show_surface_grid`	Logical; if `TRUE`, draws a grid on the surface patch.
`surface_grid_color`, `surface_grid_width`	Color and width of the surface grid lines.
`show_axis_grid`	Logical; if `TRUE`, draws grid lines on the coordinate axes in the 3D scene.
`axis_grid_color`, `axis_grid_width`	Color and width of the axis grid lines.
`scene`	List with 3D scene options passed to `plotly::layout`, typically including axis titles and `aspectmode`.
`bg`	List with background colors for the figure, with fields `paper` and `plot`.

Details

Given a differentiable function `f(x, y)` and a point `(x0, y0)`, the function:

- approximates the partial derivatives `f_x(x0, y0)` and `f_y(x0, y0)` using centered finite differences with step `h`,
- builds the tangent plane `g(x, y) = f(x0, y0) + f_x(x0, y0) * (x - x0) + f_y(x0, y0) * (y - y0)`,
- constructs a normal vector `n = (-f_x, -f_y, 1)` and its unit version,
- encodes the plane in the form `a x + b y + c z + d = 0`, with the coefficients returned in `plane_coeff`.

When `plot = TRUE`, the function produces an interactive figure containing:

- a patch of the original surface,
- a patch of the tangent plane centered at `(x0, y0)`,
- two intersection curves of the surface along `x = x0` and `y = y0`,
- the point of tangency and a segment in the direction of the normal vector.

Value

A list with components:

- `fx`, `fy`: approximations of the partial derivatives `f_x(x0, y0)` and `f_y(x0, y0)`,
- `f0`: value `f(x0, y0)`,
- `normal_unit`: unit normal vector at the point of tangency,
- `normal_raw`: non-normalized normal vector `(-fx, -fy, 1)`,
- `plane_fun`: function `g(x, y)` for the tangent plane,
- `plane_coeff`: numeric vector `c(a, b, c, d)` such that `a x + b y + c z + d = 0` is the tangent-plane equation,
- `fig`: a **plotly** figure when `plot = TRUE`, otherwise `NULL`.

Examples

```
f <- function(x, y) x^2 + y^2
tp <- tangent_plane3d(
  f,
  point = c(1, 1),
  plot = FALSE
)
tp$plane_coeff
```

`total_differential_nd` *Total differential of a scalar field in R^n*

Description

Computes the gradient of a scalar field `f(x)` at a point `x0` using central finite differences. It also returns the total differential, understood as the linear map `v` -> `grad` `f(x0)` `%*%` `v`.

Usage

```
total_differential_nd(f, x0, h = NULL)
```

Arguments

`f`	Function of one argument `x` (numeric vector) returning a numeric scalar.
`x0`	Numeric vector giving the evaluation point.
`h`	Step size for finite differences. Can be `NULL` (automatic), a scalar, or a vector of the same length as `x0`.

Details

The function `f` must take a single numeric vector `x` and return a single numeric value.

Value

A list with components:

- `point`: the numeric vector `x0`,
- `value`: the numeric value `f(x0)`,
- `gradient`: numeric vector with the gradient at `x0`,
- `differential`: a function `d(v)` that returns `sum(gradient * v)`.

Examples

```
f <- function(x) x[1]^2 + 3 * x[2]^2
out <- total_differential_nd(f, c(1, 2))
out$gradient
out$differential(c(1, 0))  # directional derivative in direction (1,0)
```

vector_field3d *3D vector field in a curvilinear prism*

Description

Displays a three-dimensional vector field inside a solid region whose bounds in the variables x, y and z are defined by user-supplied functions. The region is described by an interval for x, lower and upper bounds in the y direction depending on x, and lower and upper bounds in the z direction that may depend on both x and y. The function samples this volume on a regular grid and draws arrows representing the vector field using **plotly**.

Usage

```
vector_field3d(
  F,
  H1,
  H2,
  G1,
  G2,
  a,
  b,
  NX = 8,
  NY = 6,
  NZ = 6,
  plot = TRUE,
  arrows = c("both", "line", "cone", "none"),
  arrow_scale = 0.08,
  normalize_bias = 1,
  normal_color = "black",
  normal_width = 1.5,
  arrow_color = "black",
  arrow_opacity = 0.9,
  arrow_size = 0.35,
 scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
    zaxis = list(title = "z")),
  bg = list(paper = "white", plot = "white")
)
```

Arguments

F	A function function(x, y, z) returning a numeric vector of length three, interpreted as c(Fx, Fy, Fz).
H1, H2	Functions of one variable x giving the lower and upper bounds in the y direction.
G1, G2	Functions of two variables x and y giving the lower and upper bounds in the z direction.
a, b	Numeric endpoints of the interval for x. It is assumed that b > a.

`NX, NY, NZ`	Integers greater than or equal to one giving the grid density in the three parameter directions. Each parameter is sampled using a regular sequence between zero and one with `N + 1` points.
`plot`	Logical; if `TRUE`, the vector field is drawn using **plotly**.
`arrows`	Character string indicating the arrow mode. Allowed values are `"both"`, `"line"`, `"cone"` and `"none"`.
`arrow_scale`	Global length scale for arrows, expressed as a fraction of the largest span of the bounding box.
`normalize_bias`	Numeric saturation bias used in the scaling of the vector norm. Larger values make the arrow lengths saturate earlier.
`normal_color`	Color for the arrow shafts (line segments).
`normal_width`	Numeric width of the arrow shafts.
`arrow_color`	Color for the arrow heads (cones or chevrons).
`arrow_opacity`	Numeric opacity for arrow heads when cones are available.
`arrow_size`	Relative size of arrow heads with respect to `arrow_scale`.
`scene`	List with 3D scene options for **plotly**.
`bg`	List with background colors for the figure, typically with entries `paper` and `plot`.

Details

The domain is parameterized by three normalized parameters, one for each direction. For each grid point, the corresponding physical coordinates in `x`, `y` and `z` are obtained by linear interpolation between the lower and upper bounds. The vector field is evaluated at each of these points.

Arrow lengths are scaled using a saturated version of the vector norm. This avoids extremely long arrows when the magnitude of the field varies strongly across the region. A bias parameter controls how quickly the lengths approach saturation: small magnitudes produce short arrows and large magnitudes are capped so that they remain visible without dominating the picture.

Depending on the selected mode, the function can:

- draw only line segments representing the arrow shafts,
- draw only arrow heads (cones or chevrons),
- or combine both shafts and heads.

The plotted figure can be customized through colors, opacity settings, line widths and standard **plotly** scene options. If plotting is disabled, the function still returns the sampled data for further processing.

Value

A list with:

- `points`: a data frame with base positions `x, y, z` and the magnitude of the field at each point,
- `segments`: a data frame with columns `x0, y0, z0, x1, y1, z1` describing the arrow shafts,
- `fig`: a **plotly** object when `plot = TRUE`, otherwise `NULL`.

Examples

```
H1 <- function(x) -1 - x
H2 <- function(x)  1 - x^2
G1 <- function(x, y) y
G2 <- function(x, y) y + 1

F <- function(x, y, z) c(-y, x, 1)

vector_field3d(
  F, H1 = H1, H2 = H2, G1 = G1, G2 = G2,
  a = -1, b = 1, NX = 8, NY = 6, NZ = 6,
  plot = TRUE, arrows = "both",
  arrow_scale = 0.08, normalize_bias = 1,
  normal_color = "rgba(0,0,0,0.6)", normal_width = 2,
  arrow_color  = "#1d3557", arrow_opacity = 0.95, arrow_size = 0.35
)
```

`xy_region` *Planar region between two curves y = H1(x) and y = H2(x)*

Description

Constructs a numerical representation of the planar region bounded by two functions of one variable. The region consists of all points whose horizontal coordinate lies between `a` and `b`, and whose vertical coordinate lies between the values returned by `H1(x)` and `H2(x)`. Optionally, the region can be displayed either as a two-dimensional filled subset of the plane or as a thin surface in three dimensions using **plotly**.

Usage

```
xy_region(
  H1,
  H2,
  a,
  b,
  D,
  plot = TRUE,
  n_curve = 800,
  fill = FALSE,
  fillcolor = "rgba(49,130,189,0.25)",
  boundary_line = list(color = "blue", width = 2),
  partition_line = list(color = "blue", width = 1),
  show_end_edges = TRUE,
  axis_equal = TRUE,
  as_3d = FALSE,
  plane_z = 0,
  n_u = 30,
  surface_colorscale = "Blues",
  surface_opacity = 0.3,
  show_surface_grid = TRUE,
  surface_grid_color = "rgba(60,80,200,0.25)",
  surface_grid_width = 1,
 scene = list(aspectmode = "data", xaxis = list(title = "x"), yaxis = list(title = "y"),
    zaxis = list(title = "z")),
  bg = list(paper = "white", plot = "white")
)
```

Arguments

`H1, H2`	Functions of one variable returning the lower and upper vertical bounds at each value of `x`.
`a, b`	Numeric endpoints of the interval for `x`. It is assumed that `a < b`.
`D`	Integer giving the number of subdivisions in the horizontal direction (number of vertical strips).
`plot`	Logical; if `TRUE`, produces a visualization using **plotly**.
`n_curve`	Integer giving the number of points for sampling the boundary curves.
`fill`	Logical; if `TRUE`, fills the two-dimensional region.
`fillcolor`	Character string defining the fill color in 2D mode.
`boundary_line`	List with **plotly** style options for drawing the boundary curves.
`partition_line`	List with style parameters for vertical partition lines.
`show_end_edges`	Logical; if `TRUE`, draws boundary segments at the endpoints `x = a` and `x = b`.
`axis_equal`	Logical; if `TRUE`, enforces equal scaling on both axes in two dimensions.
`as_3d`	Logical; if `TRUE`, draws the region as a thin three-dimensional plate.
`plane_z`	Numeric height at which to draw the region when `as_3d = TRUE`.
`n_u`	Integer number of internal subdivisions used for discretization of the region in the cross-section (between lower and upper boundary).
`surface_colorscale`	Character string specifying a **plotly** colorscale for the three-dimensional mode.
`surface_opacity`	Numeric value between 0 and 1 controlling the transparency of the surface in 3D mode.
`show_surface_grid`	Logical; if `TRUE`, overlays grid lines on the plotted surface in 3D mode.
`surface_grid_color`	Character string giving the color of the grid lines in 3D mode.
`surface_grid_width`	Numeric width of the grid lines in 3D mode.
`scene`	Optional list of **plotly** scene parameters for three-dimensional rendering.
`bg`	Optional list defining the background colors of the figure, typically with components `paper` and `plot`.

Details

The function samples the interval `[a, b]` at `n_curve` points to represent the boundary curves. The interval `[a, b]` is also subdivided into `D` vertical strips. For each strip, the values `H1(x)` and `H2(x)` are evaluated to define the vertical bounds of the region.

Depending on the arguments, the function can:

- build a data grid suitable for numerical integration or visualization,
- draw a two-dimensional depiction of the region, possibly filled with a selected color,
- generate a simple three-dimensional visualization where the region is drawn as a thin plate at a chosen height.

Additional options allow drawing grid lines, showing the boundary curves, controlling colors and transparency, and adjusting the aspect ratio.

Value

A list containing:

- `x`: the sample points along the horizontal axis,
- `y1`, `y2`: the sampled boundary values,
- `y_low`, `y_high`: the lower and upper envelopes `pmin(H1,H2)` and `pmax(H1,H2)`,
- `x_part`: the partition points used in the horizontal direction,
- `fig`: a **plotly** object for visualization if `plot = TRUE` and **plotly** is available; otherwise `NULL`.

Examples

```
H1 <- function(x) 0
H2 <- function(x) 1 - x
xy_region(H1, H2, a = 0, b = 1, D = 20, plot = FALSE)
```

Index