

Package: guerrilla (via r-universe)

October 14, 2026

Version 0.4.0.9000

Title Illustrate Various Methods of Interpolation for Irregular Data

Description Examples of interpolating irregular data, to illustrate the mechanics of various methods and some easy tools to run them.

License GPL-3

Encoding UTF-8

LazyData true

ByteCompile true

URL <https://github.com/hypertidy/guerrilla>,
<https://hypertidy.github.io/guerrilla/>

BugReports <https://github.com/hypertidy/guerrilla/issues>

Depends R (>= 3.3.0)

Imports geometry (>= 0.4.0), geos, methods, vaster, wk

Suggests Rvcg, interp, raster, terra, gdalraster, sf, knitr,
spatstat.geom, rmarkdown, fields, gstat, palr, readxl, reproj,
covr, testthat, mgcv, maps

VignetteBuilder knitr

Roxygen list(markdown = TRUE)

Config/roxygen2/version 8.1.0

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

Date/Publication 2026-09-10 01:58:11 UTC

RemoteUrl <https://github.com/hypertidy/guerrilla>

RemoteRef HEAD

RemoteSha 2d7bfc010c743ffc1c7545bc5e8456483ab49069

Contents

as.data.frame.guerrilla_grid	2
as.matrix.guerrilla_grid	3
as_grid	4
bary_weights	4
bathy	5
collapse_duplicates	6
converters	7
find_triangle	8
grid_barycentric	8
grid_bin	10
grid_facet_lm	11
grid_gam	12
grid_gdal	13
grid_idw	15
grid_kriging	16
grid_ncell	17
grid_res	18
grid_smooth	18
grid_spec	19
grid_tps	20
grid_voronoi	22
grid_xy	23
is_grid	23
mesh_raster	24
plot.guerrilla_grid	25
xyz_input	26
Index	28

as.data.frame.guerrilla_grid
<i>Turn a grid into a data frame</i>

Description

One row per cell, with the cell centre coordinates and the value.

Usage

```
## S3 method for class 'guerrilla_grid'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

x a guerrilla_grid
 row.names, optional
 ignored, for consistency with the generic
 ... ignored

Value

A data frame of x, y and value.

Examples

```
g <- grid_spec(dimension = c(3, 2), extent = c(0, 3, 0, 2))
g$values <- 1:6
head(as.data.frame(g))
```

```
as.matrix.guerrilla_grid
```

Turn a grid into a matrix

Description

The values as a matrix, one row per grid row, with the top row of the grid first. This is the arrangement image() and raster both think in, and it is where most orientation bugs come from, so it gets its own function.

Usage

```
## S3 method for class 'guerrilla_grid'
as.matrix(x, ...)
```

Arguments

x a guerrilla_grid
 ... ignored

Value

A matrix with nrow rows and ncol columns.

Examples

```
g <- grid_spec(dimension = c(3, 2), extent = c(0, 3, 0, 2))
g$values <- 1:6
as.matrix(g)
```

`as_grid`*Read another package's raster as a grid*

Description

Take the dimension, extent and coordinate system off a RasterLayer or a terra SpatRaster and return them as a `grid_spec()`. Values are not carried over; this is for describing a target to interpolate onto.

Usage

```
as_grid(x)
```

Arguments

`x` a guerrilla_grid, a RasterLayer, or a SpatRaster

Details

It exists so that a grid you already have still works as a target, and so that the claim these classes are all the same few numbers can be checked rather than taken on trust.

Value

A guerrilla_grid.

Examples

```
if (requireNamespace("raster", quietly = TRUE)) {  
  as_grid(raster::raster(raster::extent(0, 8, 0, 6), ncols = 4, nrows = 3))  
}
```

`bary_weights`*Barycentric coordinates of points within a triangle*

Description

The weights that say how much of each corner a point is made of. They sum to one, and they are all non-negative exactly when the point is inside the triangle – which makes them a point-in-triangle test and an interpolation rule at the same time.

Usage

```
bary_weights(triangle, xy)
```

Arguments

triangle	three corners, as a 3 by 2 matrix
xy	points to place within it, a two column matrix

Details

This is the whole of what `geometry::tsearch(bary = TRUE)` computes in C, written out so it can be read. For a triangle with corners (x_1, y_1) , (x_2, y_2) , (x_3, y_3) and a point (x, y) , solve for the weights by Cramer's rule on

$$w_1(x_1, y_1) + w_2(x_2, y_2) + w_3(x_3, y_3) = (x, y), \quad w_1 + w_2 + w_3 = 1$$

Estimating a value is then one line: the weighted sum of the corner values.

Value

A matrix with one row per point and three columns, one weight per corner of triangle.

See Also

[find_triangle\(\)](#), which uses this to locate points, and [grid_barycentric\(\)](#), which uses it to interpolate.

Examples

```
tri <- cbind(c(0, 1, 0), c(0, 0, 1))
## the corners themselves
bary_weights(tri, tri)

## the centroid is one third of each
bary_weights(tri, cbind(1/3, 1/3))

## outside the triangle, a weight goes negative
bary_weights(tri, cbind(1, 1))
```

bathy

Bathymetry and topography for the BROKE-West region

Description

A gridded elevation layer covering the Southern Ocean sector surveyed by the BROKE-West voyage, used as a backdrop for the interpolation examples.

Format

A RasterLayer of 300 rows by 1200 columns at 1/15 degree resolution, in longitude/latitude on WGS84. Values are integer metres, from -5574 to 3612, with 17280 of 360000 cells missing.

Details

Derived from ETOPO2, cropped to 15 to 95 degrees east and 75 to 55 degrees south, aggregated by a factor of 2, smoothed with a Gaussian focal weight, and rounded to whole metres. Values are elevation relative to sea level, so the ocean is negative and the Antarctic continent is positive. Cells around the edge are NA where the focal window ran off the grid.

Source

ETOPO2, via data-raw/bathy.R. See inst/examples/afternoon_exercise.Rmd for how it was built.

collapse_duplicates	<i>Collapse coordinates that repeat</i>
---------------------	---

Description

Triangulation has nothing to say about two values at the same place, and Qhull refuses outright. Real data does this constantly: a station occupied twice, a transect that crosses its own track, coordinates rounded to a printed precision.

Usage

```
collapse_duplicates(
  xy,
  value,
  fun = mean,
  quiet = FALSE,
  fun_label = deparse(substitute(fun))
)
```

Arguments

xy	coordinates, two columns
value	one value per row of xy
fun	function used to combine the values at a repeated coordinate
quiet	do not report how many coordinates were collapsed
fun_label	name of fun to use in that report

Details

So decide explicitly. `collapse_duplicates()` reduces repeated coordinates to one, combining their values with `fun`, and says how many it collapsed.

Value

A list of `xy` and `value`, with no repeated coordinate.

Examples

```
xy <- cbind(c(1, 2, 2, 3), c(1, 2, 2, 3))
collapse_duplicates(xy, c(10, 20, 30, 40))
```

converters

*Hand a grid to another package***Description**

Converters to the raster classes people actually have installed. Each one is a handful of lines, because there is nothing in a grid that these classes do not also hold: the same dimension, the same extent, the same values in the same order.

Usage

```
as_raster(grid)

as_terra(grid)

as_gdalraster(grid, filename = NULL)
```

Arguments

grid	a guerrilla_grid
filename	file to write; defaults to a path in GDAL's /vsimem

Details

as_gdalraster() has to write a file, since that is how GDAL works; by default it writes to GDAL's in-memory filesystem. It returns the open GDALRaster object, not the file name, and the caller has to \$close() it. The file name is \$getFilename() on the returned object.

Value

An object of the corresponding class: a RasterLayer, a SpatRaster, or an open GDALRaster.

Examples

```
xy <- cbind(runif(50), runif(50))
g <- grid_barycentric(xy, xy[, 1])
if (requireNamespace("terra", quietly = TRUE)) as_terra(g)
```

find_triangle	<i>Which triangle is each point in?</i>
---------------	---

Description

For every point, the row of triangles whose triangle contains it, or NA for a point in none of them.

Usage

```
find_triangle(xy, triangles, points)
```

Arguments

xy	vertex coordinates, a two column matrix
triangles	one triangle per row, as three indices into xy
points	points to locate, a two column matrix

Details

This is what the old `tri_pip()` did by building one **sp** polygon per triangle and calling `over()`. GEOS does the same job with a spatial index it builds for you: `geos::geos_intersects_matrix()` puts the triangles in an STRtree, so each point is only tested against the few whose bounding boxes it falls in. The exact containment test is then `bary_weights()`, in R.

Value

An integer vector with one element per row of points.

Examples

```
xy <- cbind(c(0, 1, 1, 0), c(0, 0, 1, 1))
triangles <- rbind(c(1, 2, 3), c(1, 3, 4))
find_triangle(xy, triangles, cbind(c(0.9, 0.1, 5), c(0.5, 0.9, 5)))
```

grid_barycentric	<i>Interpolate to a grid across a triangulation</i>
------------------	---

Description

Triangulate the input coordinates, then estimate a value at every cell of a target grid from the barycentric coordinates of the cell centre within the triangle that contains it.

Usage

```
grid_barycentric(
  x,
  value = NULL,
  grid = NULL,
  duplicates = mean,
  engine = c("geometry", "R"),
  ...
)
```

Arguments

x	coordinates, or coordinates carrying their value as z; see xyz_input()
value	one value per coordinate, or NULL to use the z of x
grid	a grid_spec() to interpolate onto, or NULL for a default one
duplicates	function used to combine values at repeated coordinates, or NULL to leave them alone
engine	"geometry" for the fast path, "R" for the readable one
...	ignored

Details

Within a triangle the surface is the plane through its three corner values, so the result is continuous, passes exactly through the input values, and invents nothing beyond them. This is what MATLAB calls `griddata(method = "linear")` and what GDAL's `gdal_grid` calls `linear`.

Cells outside the convex hull of the points are in no triangle, and are left NA. Points sharing a coordinate are combined first, because a triangulation cannot hold two values in one place; see [collapse_duplicates\(\)](#).

Value

A `guerrilla_grid` with values.

Two engines

`engine = "geometry"` locates the cells and computes their weights in one pass of `geometry::tsearch()`, in C. It is the one to use.

`engine = "R"` does the same work in the open: [find_triangle\(\)](#) to locate each cell, [bary_weights\(\)](#) to weight it. It is slower, and it is the point of this package – the two agree to floating point, and the second one can be read.

Examples

```
xy <- cbind(runif(100), runif(100))
grid_barycentric(xy, xy[, 1] + xy[, 2])

## the value can be the z of the coordinates
```

```

grid_barycentric(cbind(xy, xy[, 1]))

## the two engines agree
g1 <- grid_barycentric(xy, xy[, 1], grid_spec(xy, dimension = c(20, 20)))
g2 <- grid_barycentric(xy, xy[, 1], grid_spec(xy, dimension = c(20, 20)),
                      engine = "R")
max(abs(g1$values - g2$values), na.rm = TRUE)

```

grid_bin

*Bin values into grid cells***Description**

Assign each point to the cell that contains it, and combine the values that land in the same cell.

Usage

```
grid_bin(x, value = NULL, grid = NULL, fun = mean, ...)
```

Arguments

x	coordinates, or coordinates carrying their value as z; see xyz_input()
value	one value per coordinate, or NULL to use the z of x
grid	a grid_spec() to interpolate onto, or NULL for a default one
fun	function used to combine the values in one cell
...	ignored

Details

This is the simplest thing that can be called interpolation, and it is not really interpolation at all: nothing is estimated between the points, cells with no point in them stay NA, and the answer depends entirely on how big the cells are. It is here because it is the honest baseline, and because the question it raises – what should happen when two points land in one cell – is a question every other method answers silently.

fun is that answer, made explicit. mean averages them, length counts them, function(x) x[1] keeps the first, and looking at grid_bin(fun = length) next to any other method is the quickest way to see where a surface is supported by data and where it is supported by the algorithm.

Points on the outer edge of the extent are outside the last cell and are dropped, which is one reason to pad a grid; see [grid_spec\(\)](#).

The whole implementation is [vaster::cell_from_xy\(\)](#) and [tapply\(\)](#). No raster class is involved, because none is needed.

Value

A guerrilla_grid with values.

Examples

```
xy <- cbind(runif(300), runif(300))
g <- grid_spec(xy, dimension = c(20, 20))
plot(grid_bin(xy, xy[, 1], g))

## how many points went into each cell
plot(grid_bin(xy, xy[, 1], g, fun = length))
```

grid_facet_lm	<i>Fit a plane within each Delaunay triangle</i>
---------------	--

Description

Triangulate the points, fit value $\sim x + y$ by least squares to the three corners of each triangle, and evaluate that fit at the cells inside it.

Usage

```
grid_facet_lm(x, value = NULL, grid = NULL, duplicates = mean, ...)
```

Arguments

x	coordinates, or coordinates carrying their value as z; see xyz_input()
value	one value per coordinate, or NULL to use the z of x
grid	a grid_spec() to interpolate onto, or NULL for a default one
duplicates	function used to combine values at repeated coordinates, or NULL to leave them alone
...	ignored

Details

This is the long way round to [grid_barycentric\(\)](#), and the two agree to floating point. Three points determine a plane, least squares through three points is that plane, and barycentric interpolation evaluates that same plane – so the expensive per-triangle `lm()` and the one vectorised `tsearch()` call compute the same surface. Worth seeing once.

It replaces the Delaunay half of `facets()`, which did this through **spatstat** and returned a point pattern rather than a grid.

The Voronoi half of `facets()` is not here, because it was not doing what it looked like it was doing. A Voronoi tile contains exactly one point by construction, so `lm(value ~ x + y)` on it fits an intercept and nothing else, and predicts that one point's value across the whole tile. That is nearest neighbour, computed the slowest way available. [grid_voronoi\(\)](#) is the same answer under its right name.

Value

A `guerrilla_grid` with values.

See Also

[grid_barycentric\(\)](#) for the fast form of this, [grid_voronoi\(\)](#) for what the Voronoi case really was.

Examples

```
xy <- cbind(runif(40), runif(40))
v <- xy[, 1] + xy[, 2]
plot(grid_facet_lm(xy, v))
```

grid_gam

Generalized additive model on the coordinates

Description

Fit a smooth of x and y with [mgcv::gam\(\)](#) and predict it at every cell.

Usage

```
grid_gam(
  x,
  value = NULL,
  grid = NULL,
  formula = value ~ s(x, y),
  statistic = c("prediction", "se"),
  ...
)
```

Arguments

x	coordinates, or coordinates carrying their value as z; see xyz_input()
value	one value per coordinate, or NULL to use the z of x
grid	a grid_spec() to interpolate onto, or NULL for a default one
formula	a model formula in value, x and y
statistic	"prediction" for the fitted surface, "se" for its standard error
...	passed to mgcv::gam()

Details

The default formula is `value ~ s(x, y)`, one isotropic two dimensional smooth. That matters: `mgcv`'s default basis for a two dimensional `s()` is a thin plate regression spline, so this is [grid_tps\(\)](#) with the number of basis functions reduced and the smoothing parameter chosen by REML. The two give visibly similar surfaces on the same data, and seeing that is worth more than either surface on its own.

`value ~ s(x) + s(y)` is the wrong model for a surface and a useful thing to try: additive in x and y means every column of the grid has the same shape as every other, so it cannot represent a feature that sits in one place. Pass it as formula and look.

Value

A guerrilla_grid with values.

Examples

```
xy <- cbind(runif(200), runif(200))
v <- exp(-20 * ((xy[, 1] - 0.5)^2 + (xy[, 2] - 0.5)^2))
if (requireNamespace("mgcv", quietly = TRUE)) {
  op <- par(mfrow = c(1, 2))
  plot(grid_gam(xy, v), main = "s(x, y)")
  ## additive in x and y cannot put a bump in one place
  plot(grid_gam(xy, v, formula = value ~ s(x) + s(y)), main = "s(x) + s(y)")
  par(op)
}
```

grid_gdal

Interpolate with GDAL's own gridder

Description

Hand the points to GDAL and let gdal_grid do the interpolation, returning the result as a grid like everything else here.

Usage

```
grid_gdal(
  x,
  value = NULL,
  grid = NULL,
  algorithm = "linear",
  nodata = NaN,
  ...
)
```

Arguments

x	coordinates, or coordinates carrying their value as z; see xyz_input()
value	one value per coordinate, or NULL to use the z of x
grid	a grid_spec() to interpolate onto, or NULL for a default one
algorithm	a gdal_grid algorithm string, passed to -a
nodata	value GDAL should write for empty cells, read back as NA; ignored if algorithm already sets one
...	further arguments appended to the gdal_grid command line

Details

Four of the methods in this package have a `gdal_grid` twin, and running them side by side is the best available check that any of them is right. They are independent implementations in different languages by different people, so agreement is evidence and disagreement is a bug in one of them:

"linear:radius=0.0"	<code>grid_barycentric()</code>
"nearest"	<code>grid_voronoi()</code>
"invdist:power=2.0"	<code>grid_idw()</code>
"average:radius1=,radius2="	<code>grid_bin()</code> , with a search radius

Value

A `guerrilla_grid` with values.

Two GDAL defaults worth knowing

`algorithm = "linear"` on its own does not stop at the convex hull. GDAL's default is `radius=-1`, an infinite search, so a cell in no triangle takes the value of the nearest point instead of being left out. That is extrapolation, it is silent, and it is why `"linear:radius=0.0"` rather than `"linear"` is the one that matches `grid_barycentric()`.

Second, `gdal_grid` fills cells it could not estimate with 0 and does not record that 0 as the band's no data value, so nothing downstream can tell it apart from a real measurement – and 0 is a real temperature, a real elevation and a real anomaly. This function therefore appends `nodata=nan` unless the algorithm string already sets one. That both fills the empty cells with NaN and tags the band, and **gdalraster** turns the tagged value into NA on the way back, so an explicit `nodata=-9999` arrives here as NA just the same.

Why sf

`gdal_grid` is reachable from R only through `sf::gdal_utils()`. **gdalraster** wraps `warp`, `translate` and `rasterize` but not `GDALGrid`, and the unified `gdal` command line added in GDAL 3.11 has no `grid` subcommand either, so there is nothing for it to wrap yet.

See Also

`grid_barycentric()`, `grid_voronoi()`, `grid_idw()`

Examples

```
xy <- cbind(runif(50), runif(50))
if (requireNamespace("sf", quietly = TRUE) &&
    requireNamespace("gdalraster", quietly = TRUE)) {
  g <- grid_spec(xy, dimension = c(40, 40))
  gdal <- grid_gdal(xy, xy[, 1], g, algorithm = "linear:radius=0.0")
  ours <- grid_barycentric(xy, xy[, 1], g)
  max(abs(gdal$values - ours$values), na.rm = TRUE)
}
```

grid_idw	<i>Inverse distance weighted interpolation</i>
----------	--

Description

Estimate each cell as a weighted average of the input values, weighted by one over distance to the power `idp`.

Usage

```
grid_idw(x, value = NULL, grid = NULL, idp = 2, nmax = Inf, ...)
```

Arguments

<code>x</code>	coordinates, or coordinates carrying their value as <code>z</code> ; see xyz_input()
<code>value</code>	one value per coordinate, or <code>NULL</code> to use the <code>z</code> of <code>x</code>
<code>grid</code>	a grid_spec() to interpolate onto, or <code>NULL</code> for a default one
<code>idp</code>	the inverse distance power
<code>nmax</code>	use at most this many of the nearest points per cell
<code>...</code>	passed to gstat::idw()

Details

There is no model here, only a rule, and that is the point of having it next to the others. Raising `idp` makes the nearest point dominate, so the surface tends to nearest neighbour ([grid_voronoi\(\)](#)); lowering it flattens towards the overall mean. Neither end is more correct than the other, and nothing in the data tells you where to sit between them.

The consequence is that this method cannot report a standard error, because it never claimed to be estimating anything. [grid_kriging\(\)](#) is the same idea taken seriously: weights chosen from a fitted model of how the values vary with distance, which then also says how uncertain each cell is.

Value

A `guerrilla_grid` with values.

Examples

```
xy <- cbind(runif(50), runif(50))
if (requireNamespace("gstat", quietly = TRUE)) {
  op <- par(mfrow = c(1, 2))
  plot(grid_idw(xy, xy[, 1], idp = 0.5), main = "idp = 0.5")
  plot(grid_idw(xy, xy[, 1], idp = 8), main = "idp = 8")
  par(op)
}
```

grid_kriging

*Kriging***Description**

Fit a variogram to the data, then use it to weight the input values for every cell.

Usage

```
grid_kriging(
  x,
  value = NULL,
  grid = NULL,
  model = NULL,
  statistic = c("prediction", "se"),
  ...
)
```

Arguments

<code>x</code>	coordinates, or coordinates carrying their value as <code>z</code> ; see xyz_input()
<code>value</code>	one value per coordinate, or <code>NULL</code> to use the <code>z</code> of <code>x</code>
<code>grid</code>	a grid_spec() to interpolate onto, or <code>NULL</code> for a default one
<code>model</code>	a <code>variogramModel</code> from gstat::vgm() to fit, or <code>NULL</code> for a spherical starting guess
<code>statistic</code>	"prediction" for the kriged surface, "se" for the square root of the kriging variance
<code>...</code>	passed to gstat::krige()

Details

The variogram is a picture of how much two values differ, as a function of how far apart they are. Kriging fits a curve to that picture and uses it to choose the weights, so unlike [grid_idw\(\)](#) the weighting is estimated from the data rather than chosen.

That also makes this the one method here that can refuse. If the values have no spatial structure at these distances there is no curve to fit, the fit comes back with a negative range, and this stops rather than kriging from nonsense. Every other method in the package will happily interpolate pure noise and hand you a picture of it.

Because there is a fitted model, there is also a variance: `statistic = "se"` returns the square root of the kriging variance, which is small near the data and rises to the sill away from it. That surface is usually more informative than the prediction, and it is the reason to reach for this method at all.

`model` is passed to [gstat::fit.variogram\(\)](#) as the starting point. The default is a spherical model with the sill at the overall variance and the range at a third of the distances the variogram covers. That is a starting guess, not an analysis: a real one plots [gstat::variogram\(\)](#) first and chooses. If the fit does not converge, that is what it is telling you.

Value

A guerrilla_grid with values.

Examples

```
xy <- cbind(runif(80), runif(80))
v <- xy[, 1] * 3 + rnorm(80, sd = 0.1)
if (requireNamespace("gstat", quietly = TRUE)) {
  plot(grid_kriging(xy, v))
  plot(grid_kriging(xy, v, statistic = "se"))
}
```

grid_ncell	<i>Number of cells in a grid</i>
------------	----------------------------------

Description

Number of cells in a grid

Usage

```
grid_ncell(grid)
```

Arguments

grid a guerrilla_grid

Value

A single integer, the product of the two dimensions.

Examples

```
grid_ncell(grid_spec(dimension = c(3, 2), extent = c(0, 6, 0, 4)))
```

grid_res	<i>Cell size in each direction</i>
----------	------------------------------------

Description

Cell size in each direction

Usage

```
grid_res(grid)
```

Arguments

grid	a guerrilla_grid
------	------------------

Value

Two numbers, the x and y size of one cell.

Examples

```
grid_res(grid_spec(dimension = c(4, 2), extent = c(0, 8, 0, 4)))
```

grid_smooth	<i>Kernel smoothing on a grid</i>
-------------	-----------------------------------

Description

Bin the points to a grid, then smooth that grid with a Gaussian kernel, using `fields::image.smooth()`.

Usage

```
grid_smooth(x, value = NULL, grid = NULL, theta = NULL, ...)
```

Arguments

x	coordinates, or coordinates carrying their value as z; see <code>xyz_input()</code>
value	one value per coordinate, or NULL to use the z of x
grid	a <code>grid_spec()</code> to interpolate onto, or NULL for a default one
theta	kernel bandwidth, in the units of the coordinates
...	passed to <code>fields::image.smooth()</code>

Details

This is the two step method: rasterize first (`grid_bin()`), estimate second. Everything else here works from the points directly, so this one is worth having as the contrast – the binning throws away where inside its cell each point was, and no amount of smoothing afterwards gets that back.

theta is the kernel bandwidth in coordinate units, and it does all the work. There is no standard error, because there is no model.

`fields::image.smooth()` wants a square grid, so this uses one of `max(dimension)` cells a side and returns it; the result covers the same extent but need not have the dimension you asked for.

Value

A `guerrilla_grid` with values.

Examples

```
xy <- cbind(runif(200), runif(200))
if (requireNamespace("fields", quietly = TRUE)) {
  plot(grid_smooth(xy, xy[, 1] + xy[, 2], theta = 0.1))
}
```

grid_spec

A grid, as a plain list

Description

Describe a regular grid: how many cells across and down, what region of the plane it covers, and optionally what coordinate system that plane is in.

Usage

```
grid_spec(
  x = NULL,
  dimension = c(60L, 50L),
  extent = NULL,
  crs = NULL,
  pad = 0
)
```

Arguments

x	coordinates to cover: anything with two or more columns, such as a matrix or data frame. Ignored when extent is given.
dimension	number of columns and rows, in that order
extent	outer edges <code>c(xmin, xmax, ymin, ymax)</code> ; taken from x if not given
crs	coordinate reference system, or NULL (the default) for none
pad	fraction of each axis range to add on every side, so the outermost input coordinates do not sit exactly on the boundary. 0 by default.

Details

The value returned is a list. That is deliberate. Every raster in every package is some version of these few numbers, and a reader who can print the whole thing has actually been told what a raster is:

- dimension, the number of columns and rows, in that order;
- extent, the outer edges as xmin, xmax, ymin, ymax;
- crs, a coordinate reference system, or NULL for none;
- values, one number per cell, or NULL for a grid that is only a target.

dimension and extent are exactly the first two arguments of the **vaster** functions, so `grid$dimension` and `grid$extent` can be handed straight to `vaster::xy_from_cell()` and friends. Cells are numbered left to right along the top row first, the same order raster and GDAL use.

Nothing here assumes the plane is the Earth. crs stays NULL unless you say otherwise.

Value

A guerrilla_grid, which is a list of dimension, extent, crs and values.

Examples

```
xy <- cbind(c(0, 4, 10), c(0, 2, 5))
grid_spec(xy)

## it really is just a list
str(unclass(grid_spec(xy)))

## give the points some room
grid_spec(xy, pad = 0.05)$extent

## or skip the points entirely
grid_spec(dimension = c(4, 3), extent = c(0, 8, 0, 6))
```

grid_tps

Thin plate spline interpolation

Description

Fit a thin plate spline to the points with `fields::Tps()` and evaluate it at every cell.

Usage

```
grid_tps(
  x,
  value = NULL,
  grid = NULL,
  statistic = c("prediction", "se"),
```

```

    lon.lat = FALSE,
    ...
)

```

Arguments

<code>x</code>	coordinates, or coordinates carrying their value as <code>z</code> ; see xyz_input()
<code>value</code>	one value per coordinate, or <code>NULL</code> to use the <code>z</code> of <code>x</code>
<code>grid</code>	a grid_spec() to interpolate onto, or <code>NULL</code> for a default one
<code>statistic</code>	"prediction" for the fitted surface, "se" for its standard error
<code>lon.lat</code>	the coordinates are longitude and latitude, so distances should be great circle ones; passed to fields::Tps()
<code>...</code>	passed to fields::Tps() , for instance <code>lambda</code> or <code>m</code>

Details

A thin plate spline is the surface that passes near the data while bending as little as possible, with a smoothing parameter deciding the trade between those two. It is the engine behind GDAL's `-tps` warping, so this is also what happens when you georeference an image from ground control points.

Unlike [grid_barycentric\(\)](#) it fills the whole grid, including cells far outside the data, where the answer is extrapolation and should be read as such. `statistic = "se"` is how it tells you that: the standard error grows with distance from the data, so the two surfaces together say both what the method thinks and where it is guessing.

`lon.lat` is the one place in this package where what the coordinates mean changes the arithmetic. A spline bends in the plane of its coordinates, and a degree of longitude is not a degree of latitude anywhere but the equator. It is not guessed from the `crs`, because the guess would be wrong exactly when it mattered; say what you have.

Value

A `guerrilla_grid` with values.

Examples

```

xy <- cbind(runif(60), runif(60))
v <- sin(xy[, 1] * 6) + xy[, 2]
if (requireNamespace("fields", quietly = TRUE)) {
  plot(grid_tps(xy, v))
  ## and where it is guessing
  plot(grid_tps(xy, v, statistic = "se"))
}

```

grid_voronoi

Nearest neighbour interpolation, by Voronoi tessellation

Description

Give every cell the value of the closest input point.

Usage

```
grid_voronoi(x, value = NULL, grid = NULL, duplicates = mean, ...)
```

Arguments

x	coordinates, or coordinates carrying their value as z; see xyz_input()
value	one value per coordinate, or NULL to use the z of x
grid	a grid_spec() to interpolate onto, or NULL for a default one
duplicates	function used to combine values at repeated coordinates, or NULL to leave them alone
...	ignored

Details

The tessellation is the picture of that rule. A Voronoi tile is exactly the region closer to its own point than to any other, so assigning each cell to the tile it falls in is nearest neighbour, drawn rather than computed.

Unlike [grid_barycentric\(\)](#), this fills the whole grid. Voronoi tiles cover the plane, so there is no convex hull to fall outside of – which is convenient and is also the method’s main way of lying to you, since a cell far beyond the data gets a confident answer from one distant point.

Value

A guerrilla_grid with values.

Examples

```
xy <- cbind(runif(30), runif(30))
plot(grid_voronoi(xy, xy[, 1]))
points(xy, pch = 16, cex = 0.5)
```

grid_xy	<i>Coordinates of the cell centres</i>
---------	--

Description

The centre of every cell of grid, in cell order: left to right along the top row, then the next row down.

Usage

```
grid_xy(grid)
```

Arguments

grid	a guerrilla_grid
------	------------------

Details

This is one call to `vaster::xy_from_cell()` with the grid's own first two elements. It is the only thing the interpolation functions need to know about the grid, which is why the grid can be a list.

Value

A two column matrix with one row per cell.

Examples

```
grid_xy(grid_spec(dimension = c(3, 2), extent = c(0, 6, 0, 4)))
```

is_grid	<i>Is this a guerrilla grid?</i>
---------	----------------------------------

Description

Is this a guerrilla grid?

Usage

```
is_grid(x)
```

Arguments

x	any object
---	------------

Value

TRUE or FALSE

Examples

```
is_grid(grid_spec(cbind(1:3, 1:3)))
is_grid(1:10)
```

 mesh_raster

Mesh raster

Description

Create a grid by interpolating across triangles.

Usage

```
mesh_raster(x, grid = NULL, n = 128, ...)

## S3 method for class 'mesh3d'
mesh_raster(x, grid = NULL, n = 128, ...)

## S3 method for class 'matrix'
mesh_raster(x, grid = NULL, n = 128, ...)

## S3 method for class 'data.frame'
mesh_raster(x, grid = NULL, n = 128, ...)
```

Arguments

x	a matrix or data frame of three columns, or a mesh3d
grid	a grid_spec() to interpolate onto, or NULL for a default one
n	cells across, when grid is not supplied
...	passed to methods, and on to grid_barycentric()

Details

For x-y-z input this is [grid_barycentric\(\)](#) with the arguments arranged differently: triangulate, then estimate each cell from the triangle containing it. The difference is that `mesh_raster()` also takes a mesh3d, where the triangles already exist and no triangulation is needed.

That case is worth having because it runs the argument backwards. A grid is one particular mesh – a regular one, with the values at the vertices – so turning an arbitrary mesh into a grid is resampling one mesh onto another, and barycentric interpolation is how you do it either way.

Value

A `guerrilla_grid` with values.

Examples

```
## interpolate from raw points
xyz <- quakes[c("long", "lat", "depth")]
xyz$depth <- -xyz$depth
gx <- mesh_raster(xyz)
rat <- 1/cos(mean(xyz[["lat"]]) * pi/180)
plot(gx, asp = rat, col = hcl.colors(12, "YlOrRd"))
points(xyz, pch = "+", cex = 0.3)

## add some dummy points (we aren't modelling the world)
xex <- cbind(expand.grid(long = range(xyz$long),
                        lat = range(xyz$lat)), depth = 0)
g2 <- mesh_raster(rbind(xex, xyz))
plot(g2, asp = rat)
points(xyz, pch = "+", cex = 0.3)

if (requireNamespace("maps", quietly = TRUE)) {
  maps::map(add = TRUE)
}

## a mesh of triangles, rather than raw points
if (requireNamespace("Rvcg", quietly = TRUE)) {
  data("humface", package = "Rvcg")
  grid <- mesh_raster(humface, n = 256)
  plot(grid, col = grey.colors(24))
}

## for the other direction, a grid drawn as a 3D mesh, see
## vignette("triangulation")
```

plot.guerrilla_grid *Plot a grid*

Description

Draw the grid with `graphics::image()`. No raster class is involved, which is the point: a grid is a dimension, an extent and a vector of values, and that is enough to draw it.

Usage

```
## S3 method for class 'guerrilla_grid'
plot(x, col = grDevices::hcl.colors(24, "YlGnBu"), asp = 1, ...)
```

Arguments

<code>x</code>	a <code>guerrilla_grid</code> with values
<code>col</code>	colours
<code>asp</code>	aspect ratio; 1 by default, or use NA to fill the device
<code>...</code>	passed to <code>graphics::image()</code>

Value

x, invisibly.

Examples

```
xy <- cbind(runif(50), runif(50))
g <- grid_barycentric(xy, xy[, 1] + xy[, 2])
plot(g)
```

xyz_input	<i>Coordinates and values, from whatever you have</i>
-----------	---

Description

Every interpolation function here takes the same thing: places, and a value at each place. This is what turns the various ways of expressing that into one matrix of coordinates and one vector of values.

Usage

```
xyz_input(x, value = NULL)
```

Arguments

x	coordinates: a matrix, a data frame, or anything wk handles
value	one value per coordinate, or NULL to use the z of x

Details

A value can arrive two ways, and they are the same thing said differently: as a separate value argument, or as the z of three dimensional coordinates. Barycentric interpolation really is treating the value as a height above the plane, so `wk:xyz()` input is not an abuse of notation here, it is the notation. For kriging or a GAM it would be, which is why value exists too.

Objects from **wk** and **sf** go through `wk:wk_coords()`, which also carries their coordinate reference system across. A plain matrix or data frame is read by position: first column x, second y, third z if present. `wk:as_xy()` is deliberately not used for those, because it refuses to guess at column names, and Lon/Lat is exactly what real data is called.

Value

A list of xy (a two column matrix), value, and crs.

Examples

```
xyz_input(cbind(1:3, 4:6), c(10, 20, 30))
```

```
## z is the value
```

```
xyz_input(cbind(1:3, 4:6, c(10, 20, 30)))
```

```
## and the crs comes along
```

```
xyz_input(wk::xyz(1:3, 4:6, c(10, 20, 30), crs = "EPSG:4326"))$crs
```

Index

`as.data.frame.guerrilla_grid`, 2
`as.matrix.guerrilla_grid`, 3
`as_gdalraster` (converters), 7
`as_grid`, 4
`as_raster` (converters), 7
`as_terra` (converters), 7

`bary_weights`, 4
`bary_weights()`, 8, 9
`bathy`, 5

`collapse_duplicates`, 6
`collapse_duplicates()`, 9
`converters`, 7

`fields::image.smooth()`, 18
`fields::Tps()`, 20, 21
`find_triangle`, 8
`find_triangle()`, 5, 9

`geos::geos_intersects_matrix()`, 8
`graphics::image()`, 25
`grid_barycentric`, 8
`grid_barycentric()`, 5, 11, 12, 14, 21, 22, 24
`grid_bin`, 10
`grid_bin()`, 14, 19
`grid_facet_lm`, 11
`grid_gam`, 12
`grid_gdal`, 13
`grid_idw`, 15
`grid_idw()`, 14, 16
`grid_kriging`, 16
`grid_kriging()`, 15
`grid_ncell`, 17
`grid_res`, 18
`grid_smooth`, 18
`grid_spec`, 19
`grid_spec()`, 4, 9–13, 15, 16, 18, 21, 22, 24
`grid_tps`, 20
`grid_tps()`, 12

`grid_voronoi`, 22
`grid_voronoi()`, 11, 12, 14, 15
`grid_xy`, 23
`gstat::fit.variogram()`, 16
`gstat::idw()`, 15
`gstat::krige()`, 16
`gstat::vgm()`, 16

`is_grid`, 23

`mesh_raster`, 24
`mgcv::gam()`, 12

`plot.guerrilla_grid`, 25

`sf::gdal_utils()`, 14

`tapply()`, 10

`vaster::cell_from_xy()`, 10
`vaster::xy_from_cell()`, 20, 23

`wk::wk_coords()`, 26

`xyz_input`, 26
`xyz_input()`, 9–13, 15, 16, 18, 21, 22