Package 'affiner'

October 14, 2024

Type Package

Title A Finer Way to Render 3D Illustrated Objects in 'grid' Using Affine Transformations

Version 0.1.1

Description Dilate, permute, project, reflect, rotate, shear, and translate 2D and 3D points. Supports parallel projections including oblique projections such as the cabinet projection as well as axonometric projections such as the isometric projection. Use 'grid's ``affine transformation'' feature to render illustrated flat surfaces.

URL https://trevorldavis.com/R/affiner/

BugReports https://github.com/trevorld/affiner/issues

License MIT + file LICENSE

Imports graphics, grDevices, grid, R6, utils

Suggests ggplot2, gridpattern, knitr, ragg (>= 1.3.3), rgl, rlang, rmarkdown, stats, testthat (>= 3.0.0), vdiffr, withr

VignetteBuilder knitr, ragg, rmarkdown

Encoding UTF-8

RoxygenNote 7.3.1

Config/testthat/edition 3

NeedsCompilation no

Author Trevor L. Davis [aut, cre] (<https://orcid.org/0000-0001-6341-4639>)

Maintainer Trevor L. Davis <trevor.l.davis@gmail.com>

Repository CRAN

Date/Publication 2024-10-14 12:00:43 UTC

Contents

`affiner-package` *affiner: A Finer Way to Render 3D Illustrated Objects in 'grid' Using Affine Transformations*

Description

Dilate, permute, project, reflect, rotate, shear, and translate 2D and 3D points. Supports parallel projections including oblique projections such as the cabinet projection as well as axonometric projections such as the isometric projection. Use 'grid's "affine transformation" feature to render illustrated flat surfaces.

Package options

The following affiner function arguments may be set globally via `base::options()`:

affiner_angular_unit The default for the `unit` argument used by `angle()` and `as_angle()`. The default for this option is "degrees".

affiner_grid_unit The default for the `unit` argument used by `affine_settings()`. The default for this option is "inches".

The following `cli` options may also be of interest:

cli.unicode Whether UTF-8 character support should be assumed. Along with `l10n_info()` used to determine the default of the `use_unicode` argument of `format.angle()` and `print.angle()`.

Author(s)

Maintainer: Trevor L. Davis <trevor.l.davis@gmail.com> (ORCID)

See Also

Useful links:

- `https://trevorldavis.com/R/affiner/`
- Report bugs at `https://github.com/trevorld/affiner/issues`

abs.Coord1D *Compute Euclidean norm*

Description

abs() computes the Euclidean norm for Coord2D class objects and Coord3D class objects.

Usage

```
## S3 method for class 'Coord1D'
abs(x)

## S3 method for class 'Coord2D'
abs(x)

## S3 method for class 'Coord3D'
abs(x)
```

Arguments

x	A Coord2D class object or Coord2D class object.

Value

A numeric vector

Examples

```
z <- complex(real = 1:4, imaginary = 1:4)
p <- as_coord2d(z)
abs(p) # Euclidean norm
# Less efficient ways to calculate same Euclidean norms
sqrt(p * p) # `*` dot product
distance2d(p, as_coord2d(0, 0, 0))

# In {base} R `abs()` calculates Euclidean norm of complex numbers
all.equal(abs(p), abs(z))
all.equal(Mod(p), Mod(z))

p3 <- as_coord3d(x = 1:4, y = 1:4, z = 1:4)
abs(p3)
```

`affineGrob` *Affine transformation grob*

Description

`affineGrob()` is a grid grob function to facilitate using the group affine transformation features introduced in R 4.2.

Usage

```
affineGrob(
  grob,
  vp_define = NULL,
  transform = NULL,
  vp_use = NULL,
  name = NULL,
  gp = grid::gpar(),
  vp = NULL
)

grid.affine(...)
```

Arguments

`grob`	A grid grob to perform affine transformations on. Passed to `grid::defineGrob()` as its `src` argument.
`vp_define`	`grid::viewport()` to define grid group in. Passed to `grid::defineGrob()` as its `vp` argument. This will cumulative with the current viewport and the `vp` argument (if any), if this cumulative viewport falls outside the graphics device drawing area this grob may be clipped on certain graphics devices.
`transform`	An affine transformation function. If `NULL` default to `grid::viewportTransform()`. Passed to `grid::useGrob()` as its `transform` argument.
`vp_use`	`grid::viewport()` passed to `grid::useGrob()` as its `vp` argument.
`name`	A character identifier (for grid).
`gp`	A `grid::gpar()` object.
`vp`	A `grid::viewport()` object (or `NULL`).
`...`	Passed to `affineGrob()`

Value

A `grid::gTree()` (grob) object of class "affine". As a side effect `grid.affine()` draws to the active graphics device.

See Also

See `affine_settings()` for computing good `transform` and `vp_use` settings. See https://www.stat.auckland.ac.nz/~paul/Reports/GraphicsEngine/groups/groups.html for more information about the group affine transformation feature. See `isocubeGrob()` which wraps this function to render isometric cubes.

Examples

```
if (require("grid")) {
  grob <- grobTree(rectGrob(gp = gpar(fill = "blue", col = NA)),
                   circleGrob(gp=gpar(fill="yellow", col = NA)),
                   textGrob("RSTATS", gp=gpar(fontsize=32)))
  grid.newpage()
  pushViewport(viewport(width=unit(4, "in"), height=unit(2, "in")))
  grid.draw(grob)
  popViewport()
}

if (require("grid") &&
    getRversion() >= "4.2.0" &&
    isTRUE(dev.capabilities()$transformations)) {
  # Only works if active graphics device supports affine transformations
  # such as `png(type="cairo")` on R 4.2+
  vp_define <- viewport(width=unit(2, "in"), height=unit(2, "in"))
  affine <- affineGrob(grob, vp_define=vp_define)
  grid.newpage()
  pushViewport(viewport(width=unit(4, "in"), height=unit(2, "in")))
  grid.draw(affine)
  popViewport()
}
if (require("grid") &&
    getRversion() >= "4.2.0" &&
    isTRUE(dev.capabilities()$transformations)) {
  # Only works if active graphics device supports affine transformations
  # such as `png(type="cairo")` on R 4.2+
  settings <- affine_settings(xy = list(x = c(3/3, 2/3, 0/3, 1/3),
                                        y = c(2/3, 1/3, 1/3, 2/3)),
                              unit = "snpc")
  affine <- affineGrob(grob,
                       vp_define = vp_define,
                       transform = settings$transform,
                       vp_use = settings$vp)
  grid.newpage()
  grid.draw(affine)
}
```

---

affiner_options *Get affiner options*

Description

`affiner_options()` returns the `affiner` package's global options.

Usage

```
affiner_options(..., default = FALSE)
```

Arguments

`...`	`affiner` package options using `name = value`. The return list will use any of these instead of the current/default values.
`default`	If `TRUE` return the default values instead of current values.

Value

A list of option values. Note this function **does not** set option values itself but this list can be passed to `options()`, `withr::local_options()`, or `withr::with_options()`.

See Also

affiner for a high-level description of relevant global options.

Examples

```
affiner_options()

affiner_options(default = TRUE)

affiner_options(affiner_angular_unit = "pi-radians")
```

---

affine_settings — *Compute `grid` affine transformation feature viewports and transformation functions*

Description

`affine_settings()` computes `grid` group affine transformation feature viewport and transformation function settings given the (x,y) coordinates of the corners of the affine transformed "viewport" one wishes to draw in.

Usage

```
affine_settings(
  xy = data.frame(x = c(0, 0, 1, 1), y = c(1, 0, 0, 1)),
  unit = getOption("affiner_grid_unit", "inches")
)
```

Arguments

- `xy` — An R object with named elements `x` and `y` representing the (x,y) coordinates of the affine transformed "viewport" one wishes to draw in. The (x,y) coordinates of the "viewport" should be in "upper left", "lower left", "lower right", and "upper right" order (this ordering should be from the perspective of **before** the "affine transformation" of the "viewport").
- `unit` — Which `grid::unit()` to assume the `xy` "x" and "y" coordinates are expressed in.

Value

A named list with the following group affine transformation feature viewport and functions settings:

transform An affine transformation function to pass to `affineGrob()` or `useGrob()`. If `getRversion()` is less than `"4.2.0"` will instead be `NULL`.

vp A `grid::viewport()` object to pass to `affineGrob()` or `useGrob()`.

sx x-axis sx factor

flipX whether the affine transformed "viewport" is "flipped" horizontally

x x-coordinate for viewport

y y-coordinate for viewport

width Width of viewport

height Height of viewport

default.units Default `grid::unit()` for viewport

angle angle for viewport

Usage in other packages

To avoid taking a dependency on `affiner` you may copy the source of `affine_settings()` into your own package under the permissive Unlicense. Either use `usethis::use_standalone("trevorld/affiner", "standalone-affine-settings.r")` or copy the file `standalone-affine-settings.r` into your `R` directory and add `grid` to the `Imports` of your `DESCRIPTION` file.

See Also

Intended for use with `affineGrob()` and `grid::useGrob()`. See https://www.stat.auckland.ac.nz/~paul/Reports/GraphicsEngine/groups/groups.html for more information about the group affine transformation feature.

Examples

```
if (require("grid")) {
  grob <- grobTree(rectGrob(gp = gpar(fill = "blue", col = NA)),
                   circleGrob(gp=gpar(fill="yellow", col = NA)),
                   textGrob("RSTATS", gp=gpar(fontsize=32)))
  grid.newpage()
  pushViewport(viewport(width=unit(4, "in"), height=unit(2, "in")))
  grid.draw(grob)
```

```
  popViewport()
}
if (require("grid") &&
    getRversion() >= "4.2.0" &&
    isTRUE(dev.capabilities()$transformations)) {
  # Only works if active graphics device supports affine transformations
  # such as `png(type="cairo")` on R 4.2+
  vp_define <- viewport(width=unit(2, "in"), height=unit(2, "in"))
  settings <- affine_settings(xy = list(x = c(1/3, 0/3, 2/3, 3/3),
                                        y = c(2/3, 1/3, 1/3, 2/3)),
                              unit = "snpc")
  affine <- affineGrob(grob,
                       vp_define=vp_define,
                       transform = settings$transform,
                       vp_use = settings$vp)
  grid.newpage()
  grid.draw(affine)
}
if (require("grid") &&
    getRversion() >= "4.2.0" &&
    isTRUE(dev.capabilities()$transformations)) {
  # Only works if active graphics device supports affine transformations
  # such as `png(type="cairo")` on R 4.2+
  settings <- affine_settings(xy = list(x = c(3/3, 2/3, 0/3, 1/3),
                                        y = c(2/3, 1/3, 1/3, 2/3)),
                              unit = "snpc")
  affine <- affineGrob(grob,
                       vp_define=vp_define,
                       transform = settings$transform,
                       vp_use = settings$vp)
  grid.newpage()
  grid.draw(affine)
}
```

`angle` *Angle vectors*

Description

`angle()` creates angle vectors with user specified angular unit. around `as_angle()` for those angular units.

Usage

```
angle(x = numeric(), unit = getOption("affiner_angular_unit", "degrees"))

degrees(x)

gradians(x)
```

```
pi_radians(x)

radians(x)

turns(x)
```

Arguments

x	An angle vector or an object to convert to it (such as a numeric vector)
unit	A string of the desired angular unit. Supports the following strings (note we ignore any punctuation and space characters as well as any trailing s's e.g. "half turns" will be treated as equivalent to "halfturn"): • "deg" or "degree" • "half-revolution", "half-turn", or "pi-radian" • "gon", "grad", "grade", or "gradian" • "rad" or "radian" • "rev", "revolution", "tr", or "turn"

Value

A numeric vector of class "angle". Its "unit" attribute is a standardized string of the specified angular unit.

See Also

as_angle(), angular_unit(), and angle-methods. https://en.wikipedia.org/wiki/Angle#Units for more information about angular units.

Examples

```
# Different representations of the "same" angle
angle(180, "degrees")
angle(pi, "radians")
angle(0.5, "turns")
angle(200, "gradians")
pi_radians(1)

a1 <- angle(180, "degrees")
angular_unit(a1)
is_angle(a1)
as.numeric(a1, "radians")
cos(a1)

a2 <- as_angle(a1, "radians")
angular_unit(a2)
is_congruent(a1, a2)
```

`angle-methods` *Implemented base methods for angle vectors*

Description

We implemented methods for several base generics for the `angle()` vectors.

Usage

```
## S3 method for class 'angle'
as.double(x, unit = angular_unit(x), ...)

## S3 method for class 'angle'
as.complex(x, modulus = 1, ...)

## S3 method for class 'angle'
format(x, unit = angular_unit(x), ..., use_unicode = is_utf8_output())

## S3 method for class 'angle'
print(x, unit = angular_unit(x), ..., use_unicode = is_utf8_output())

## S3 method for class 'angle'
abs(x)
```

Arguments

`x`	`angle()` vector
`unit`	A string of the desired angular unit. Supports the following strings (note we ignore any punctuation and space characters as well as any trailing s's e.g. "half turns" will be treated as equivalent to "halfturn"): • "deg" or "degree" • "half-revolution", "half-turn", or "pi-radian" • "gon", "grad", "grade", or "gradian" • "rad" or "radian" • "rev", "revolution", "tr", or "turn"
`...`	Passed to `print.default()`
`modulus`	Numeric vector representing the complex numbers' modulus
`use_unicode`	If `TRUE` use Unicode symbols as appropriate.

Details

- Mathematical Ops (in particular `+` and `-`) for two angle vectors will (if necessary) set the second vector's `angular_unit()` to match the first.

- `as.numeric()` takes a `unit` argument which can be used to convert angles into other angular units e.g. `angle(x, "degrees") |> as.numeric("radians")` to cast a numeric vector `x` from degrees to radians.
- `abs()` will calculate the angle modulo full turns.
- Use `is_congruent()` to test if two angles are congruent instead of == or `all.equal()`.
- Not all implemented methods are documented here and since `angle()` is a `numeric()` class many other S3 generics besides the explicitly implemented ones should also work with it.

Value

Typical values as usually returned by these base generics.

Examples

```
# Two "congruent" angles
a1 <- angle(180, "degrees")
a2 <- angle(pi, "radians")

print(a1)
print(a1, unit = "radians")
print(a1, unit = "pi-radians")

cos(a1)
sin(a1)
tan(a1)

# mathematical operations will coerce second `angle()` object to
# same `angular_unit()` as the first one
a1 + a2
a1 - a2

as.numeric(a1)
as.numeric(a1, "radians")
as.numeric(a1, "turns")

# Use `is_congruent()` to check if two angles are "congruent"
a1 == a2
isTRUE(all.equal(a1, a2))
is_congruent(a1, a2)
is_congruent(a1, a2, mod_turns = FALSE)
a3 <- angle(-180, "degrees") # Only congruent modulus full turns
a1 == a3
isTRUE(all.equal(a1, a2))
is_congruent(a1, a3)
is_congruent(a1, a3, mod_turns = FALSE)
```

angular_unit — *Get/set angular unit of angle vectors*

Description

`angular_unit()` gets/sets the angular unit of `angle()` vectors.

Usage

```
angular_unit(x)

angular_unit(x) <- value
```

Arguments

x	An `angle()` vector
value	A string of the desired angular unit. See `angle()` for supported strings.

Value

`angular_unit()` returns a string of x's angular unit.

Examples

```
a <- angle(seq(0, 360, by = 90), "degrees")
angular_unit(a)
print(a)
angular_unit(a) <- "turns"
angular_unit(a)
print(a)
```

as_angle — *Cast to angle vector*

Description

`as_angle()` casts to an `angle()` vector

Usage

```
as_angle(x, unit = getOption("affiner_angular_unit", "degrees"), ...)

## S3 method for class 'angle'
as_angle(x, unit = getOption("affiner_angular_unit", "degrees"), ...)

## S3 method for class 'character'
```

```
as_angle(x, unit = getOption("affiner_angular_unit", "degrees"), ...)

## S3 method for class 'complex'
as_angle(x, unit = getOption("affiner_angular_unit", "degrees"), ...)

## S3 method for class 'Coord2D'
as_angle(x, unit = getOption("affiner_angular_unit", "degrees"), ...)

## S3 method for class 'Coord3D'
as_angle(
  x,
  unit = getOption("affiner_angular_unit", "degrees"),
  type = c("azimuth", "inclination"),
  ...
)

## S3 method for class 'Line2D'
as_angle(x, unit = getOption("affiner_angular_unit", "degrees"), ...)

## S3 method for class 'Plane3D'
as_angle(
  x,
  unit = getOption("affiner_angular_unit", "degrees"),
  type = c("azimuth", "inclination"),
  ...
)

## S3 method for class 'numeric'
as_angle(x, unit = getOption("affiner_angular_unit", "degrees"), ...)
```

Arguments

`x`	An R object to convert to a `angle()` vector
`unit`	A string of the desired angular unit. Supports the following strings (note we ignore any punctuation and space characters as well as any trailing s's e.g. "half turns" will be treated as equivalent to "halfturn"): • "deg" or "degree" • "half-revolution", "half-turn", or "pi-radian" • "gon", "grad", "grade", or "gradian" • "rad" or "radian" • "rev", "revolution", "tr", or "turn"
`...`	Further arguments passed to or from other methods
`type`	Use "azimuth" to calculate the azimuthal angle and "inclination" to calculate the inclination angle aka polar angle.

Value

An `angle()` vector

Examples

```
as_angle(angle(pi, "radians"), "pi-radians")
as_angle(complex(real = 0, imaginary = 1), "degrees")
as_angle(as_coord2d(x = 0, y = 1), "turns")
as_angle(200, "gradians")
```

---

as_coord1d *Cast to coord1d object*

Description

`as_coord1d()` casts to a Coord1D class object

Usage

```
as_coord1d(x, ...)

## S3 method for class 'character'
as_coord1d(x, ...)

## S3 method for class 'Coord2D'
as_coord1d(
  x,
  permutation = c("xy", "yx"),
  ...,
  line = as_line2d("x-axis"),
  scale = 0
)

## S3 method for class 'data.frame'
as_coord1d(x, ...)

## S3 method for class 'list'
as_coord1d(x, ...)

## S3 method for class 'matrix'
as_coord1d(x, ...)

## S3 method for class 'numeric'
as_coord1d(x, ...)

## S3 method for class 'Coord1D'
as_coord1d(x, ...)

## S3 method for class 'Point1D'
as_coord1d(x, ...)
```

Arguments

`x`	An object that can be cast to a Coord1D class object such as a numeric vector of x-coordinates.
`...`	Further arguments passed to or from other methods
`permutation`	Either "xy" (no permutation) or "yx" (permute x and y axes)
`line`	A Line2D object of length one representing the line you with to reflect across or project to or an object coercible to one by `as_line2d(line, ...)` such as "x-axis" or "y-axis".
`scale`	Oblique projection scale factor. A degenerate `0` value indicates an orthogonal projection.

Value

A Coord1D class object

Examples

```
as_coord1d(x = rnorm(10))
```

---

as_coord2d *Cast to coord2d object*

Description

`as_coord2d()` casts to a Coord2D class object

Usage

```
as_coord2d(x, ...)

## S3 method for class 'angle'
as_coord2d(x, radius = 1, ...)

## S3 method for class 'character'
as_coord2d(x, ...)

## S3 method for class 'complex'
as_coord2d(x, ...)

## S3 method for class 'Coord3D'
as_coord2d(
  x,
  permutation = c("xyz", "xzy", "yxz", "yzx", "zyx", "zxy"),
  ...,
  plane = as_plane3d("xy-plane"),
  scale = 0,
```

```
  alpha = angle(45, "degrees")
)

## S3 method for class 'data.frame'
as_coord2d(x, ...)

## S3 method for class 'list'
as_coord2d(x, ...)

## S3 method for class 'matrix'
as_coord2d(x, ...)

## S3 method for class 'numeric'
as_coord2d(x, y = rep_len(0, length(x)), ...)

## S3 method for class 'Coord2D'
as_coord2d(x, ...)
```

Arguments

`x`	An object that can be cast to a Coord2D class object such as a matrix or data frame of coordinates.
`...`	Further arguments passed to or from other methods
`radius`	A numeric vector of radial distances.
`permutation`	Either "xyz" (no permutation), "xzy" (permute y and z axes), "yxz" (permute x and y axes), "yzx" (x becomes z, y becomes x, z becomes y), "zxy" (x becomes y, y becomes z, z becomes x), "zyx" (permute x and z axes). This permutation is applied before the (oblique) projection.
`plane`	A Plane3D class object representing the plane you wish to project to or an object coercible to one using `as_plane3d(plane, ...)` such as "xy-plane", "xz-plane", or "yz-plane".
`scale`	Oblique projection foreshortening scale factor. A (degenerate) `0` value indicates an orthographic projection. A value of `0.5` is used by a “cabinet projection” while a value of `1.0` is used by a “cavalier projection”.
`alpha`	Oblique projection angle (the angle the third axis is projected going off at). An `angle()` object or one coercible to one with `as_angle(alpha, ...)`. Popular angles are 45 degrees, 60 degrees, and `arctangent(2)` degrees.
`y`	Numeric vector of y-coordinates to be used.

Value

A Coord2D class object

Examples

```
df <- data.frame(x = sample.int(10, 3),
                 y = sample.int(10, 3))
```

```
as_coord2d(df)
as_coord2d(complex(real = 3, imaginary = 2))
as_coord2d(angle(90, "degrees"), radius = 2)
as_coord2d(as_coord3d(1, 2, 2), alpha = degrees(90), scale = 0.5)
```

---

as_coord3d *Cast to coord3d object*

Description

as_coord3d() casts to a Coord3D class object

Usage

```
as_coord3d(x, ...)

## S3 method for class 'angle'
as_coord3d(x, radius = 1, inclination = NULL, z = NULL, ...)

## S3 method for class 'character'
as_coord3d(x, ...)

## S3 method for class 'data.frame'
as_coord3d(x, ..., z = NULL)

## S3 method for class 'list'
as_coord3d(x, ..., z = NULL)

## S3 method for class 'matrix'
as_coord3d(x, ...)

## S3 method for class 'numeric'
as_coord3d(x, y = rep_len(0, length(x)), z = rep_len(0, length(x)), ...)

## S3 method for class 'Coord3D'
as_coord3d(x, ...)

## S3 method for class 'Coord2D'
as_coord3d(x, z = rep_len(0, length(x)), ...)
```

Arguments

x	An object that can be cast to a Coord3D class object such as a matrix or data frame of coordinates.
...	Further arguments passed to or from other methods
radius	A numeric vector. If inclination is not NULL represents spherical distances of spherical coordinates and if z is not NULL represents radial distances of cylindrical coordinates.

inclination	Spherical coordinates inclination angle aka polar angle. x represents the azimuth aka azimuthal angle.
z	Numeric vector of z-coordinates to be used
y	Numeric vector of y-coordinates to be used if hasName(x, "z") is FALSE.

Value

A Coord3D class object

Examples

```
as_coord3d(x = 1, y = 2, z = 3)
df <- data.frame(x = sample.int(10, 3),
                 y = sample.int(10, 3),
                 z = sample.int(10, 3))
as_coord3d(df)
# Cylindrical coordinates
as_coord3d(degrees(90), z = 1, radius = 1)
# Spherical coordinates
as_coord3d(degrees(90), inclination = degrees(90), radius = 1)
```

---

as_line2d *Cast to Line2D object*

Description

as_line2d() casts to a Line2D object.

Usage

```
as_line2d(...)

## S3 method for class 'numeric'
as_line2d(a, b, c, ...)

## S3 method for class 'angle'
as_line2d(theta, p1 = as_coord2d("origin"), ...)

## S3 method for class 'character'
as_line2d(x, ...)

## S3 method for class 'Coord2D'
as_line2d(normal, p1 = as_coord3d("origin"), p2, ...)

## S3 method for class 'Line2D'
as_line2d(line, ...)

## S3 method for class 'Point1D'
as_line2d(point, b = 0, ...)
```

Arguments

`...`	Passed to other function such as `as_coord2d()`.
`a, b, c`	Numeric vectors that parameterize the line via the equation `a * x + b * y + c = 0`. Note if `y = m * x + b` then `m * x + 1 * y + -b = 0`.
`theta`	Angle of the line represented by an `angle()` vector.
`p1`	Point on the line represented by a Coord2D class object.
`x`	A (character) vector to be cast to a Line2D object
`normal`	Normal vector to the line represented by a Coord2D class object. `p2` should be missing.
`p2`	Another point on the line represented by a Coord2D class object.
`line`	A Line2D object
`point`	A Point1D object

Examples

```
p1 <- as_coord2d(x = 5, y = 10)
p2 <- as_coord2d(x = 7, y = 12)
theta <- degrees(45)
as_line2d(theta, p1)
as_line2d(p1, p2)
```

---

as_plane3d *Cast to Plane3D object*

Description

`as_plane3d()` casts to a Plane3D object.

Usage

```
as_plane3d(...)

## S3 method for class 'numeric'
as_plane3d(a, b, c, d, ...)

## S3 method for class 'character'
as_plane3d(x, ...)

## S3 method for class 'Coord3D'
as_plane3d(normal, p1 = as_coord3d("origin"), p2, p3, ...)

## S3 method for class 'Plane3D'
as_plane3d(plane, ...)
```

```
## S3 method for class 'Point1D'
as_plane3d(point, b = 0, c = 0, ...)

## S3 method for class 'Line2D'
as_plane3d(line, c = 0, ...)
```

Arguments

`...`	Passed to other function such as `as_coord2d()`.
`a, b, c, d`	Numeric vectors that parameterize the plane via the equation `a * x + b * y + c * z + d = 0`.
`x`	A (character) vector to be cast to a Plane3D object
`normal`	Normal vector to the plane represented by a Coord3D class object. `p2` and `p3` should be missing.
`p1`	Point on the plane represented by a Coord3D class object.
`p2, p3`	Points on the plane represented by Coord3D class objects. `normal` should be missing.
`plane`	A Plane3D object
`point`	A Point1D object
`line`	A Line2D object

`as_point1d` *Cast to Point1D object*

Description

`as_point1d()` casts to a Point1D object.

Usage

```
as_point1d(...)

## S3 method for class 'numeric'
as_point1d(a, b, ...)

## S3 method for class 'character'
as_point1d(x, ...)

## S3 method for class 'Coord1D'
as_point1d(normal, ...)

## S3 method for class 'Point1D'
as_point1d(point, ...)
```

Arguments

`...`	Passed to other function such as `as_coord2d()`.
`a, b`	Numeric vectors that parameterize the point via the equation `a * x + b = 0`. Note this means that `x = -b / a`.
`x`	A (character) vector to be cast to a Point1D object
`normal`	Coord1D class object.
`point`	A Point1D object

Examples

```
p1 <- as_point1d(a = 1, b = 0)
```

`as_transform1d` *Cast to 1D affine transformation matrix*

Description

`as_transform1d()` casts to a `transform1d()` affine transformation matrix

Usage

```
as_transform1d(x, ...)

## S3 method for class 'transform1d'
as_transform1d(x, ...)

## Default S3 method:
as_transform1d(x, ...)
```

Arguments

`x`	An object that can be cast to a
`...`	Further arguments passed to or from other methods

Value

A `transform1d()` object

Examples

```
m <- diag(2L)
as_transform1d(m)
```

`as_transform2d` *Cast to 2D affine transformation matrix*

Description

`as_transform2d()` casts to a `transform2d()` affine transformation matrix

Usage

```
as_transform2d(x, ...)

## S3 method for class 'transform2d'
as_transform2d(x, ...)

## Default S3 method:
as_transform2d(x, ...)
```

Arguments

`x`	An object that can be cast to a
`...`	Further arguments passed to or from other methods

Value

A `transform2d()` object

Examples

```
m <- diag(3L)
as_transform2d(m)
```

`as_transform3d` *Cast to 3D affine transformation matrix*

Description

`as_transform3d()` casts to a `transform3d()` affine transformation matrix

Usage

```
as_transform3d(x, ...)

## S3 method for class 'transform3d'
as_transform3d(x, ...)

## Default S3 method:
as_transform3d(x, ...)
```

Arguments

x	An object that can be cast to a
...	Further arguments passed to or from other methods

Value

A transform3d() object

Examples

```
m <- diag(4L)
as_transform3d(m)
```

---

bounding_ranges *Compute axis-aligned ranges*

Description

range() computes axis-aligned ranges for Coord1D, Coord2D, and Coord3D class objects.

Usage

```
## S3 method for class 'Coord1D'
range(..., na.rm = FALSE)

## S3 method for class 'Coord2D'
range(..., na.rm = FALSE)

## S3 method for class 'Coord3D'
range(..., na.rm = FALSE)
```

Arguments

...	Coord1D, Coord2D, or Coord3D object(s)
na.rm	logical, indicating if NA's should be omitted

Value

Either a Coord1D, Coord2D, or Coord3D object of length two. The first element will have the minimum x/y(/z) coordinates and the second element will have the maximum x/y(/z) coordinates of the axis-aligned ranges.

Examples

```
range(as_coord2d(rnorm(5), rnorm(5)))
range(as_coord3d(rnorm(5), rnorm(5), rnorm(5)))
```

`centroid` *Compute centroids of coordinates*

Description

`mean()`computes centroids for Coord1D, Coord2D, and Coord3D class objects

Usage

```
## S3 method for class 'Coord1D'
mean(x, ...)

## S3 method for class 'Coord2D'
mean(x, ...)

## S3 method for class 'Coord3D'
mean(x, ...)
```

Arguments

`x`	A Coord1D, Coord2D, or Coord3D object
`...`	Passed to `base::mean()`

Value

A Coord1D, Coord2D, or Coord3D class object of length one

Examples

```
p <- as_coord2d(x = 1:4, y = 1:4)
print(mean(p))
print(sum(p) / length(p)) # less efficient alternative

p <- as_coord3d(x = 1:4, y = 1:4, z = 1:4)
print(mean(p))
```

`convex_hull2d` *Compute 2D convex hulls*

Description

`convex_hull2d()` is a S3 generic for computing the convex hull of an object. There is an implemented method supporting Coord2D class objects using `grDevices::chull()` to compute the convex hull.

Usage

```
convex_hull2d(x, ...)

## S3 method for class 'Coord2D'
convex_hull2d(x, ...)
```

Arguments

`x`	An object representing object to compute convex hull of such as a Coord2D class object.
`...`	Further arguments passed to or from other methods.

Value

An object of same class as `x` representing just the subset of points on the convex hull. The method for Coord2D class objects returns these points in counter-clockwise order.

Examples

```
p <- as_coord2d(x = rnorm(25), y = rnorm(25))
print(convex_hull2d(p))

# Equivalent to following caculation using `grDevices::chull()`
all.equal(convex_hull2d(p),
          p[rev(grDevices::chull(as.list(p)))])
```

`Coord1D` *1D coordinate vector R6 Class*

Description

`Coord1D` is an `R6::R6Class()` object representing two-dimensional points represented by Cartesian Coordinates.

Active bindings

`xw` A two-column matrix representing the homogeneous coordinates. The first column is the "x" coordinates and the second column is all ones.

`x` A numeric vector of x-coordinates.

Methods

Public methods:

- `Coord1D$new()`
- `Coord1D$print()`
- `Coord1D$project()`
- `Coord1D$reflect()`

- `Coord1D$scale()`
- `Coord1D$translate()`
- `Coord1D$transform()`
- `Coord1D$clone()`

Method `new()`:

Usage:

```
Coord1D$new(xw)
```

Arguments:

`xw` A matrix with three columns representing (homogeneous) coordinates. The first column represents x coordinates and the last column is all ones. Column names should be "x" and "w".

Method `print()`:

Usage:

```
Coord1D$print(n = NULL, ...)
```

Arguments:

`n` Number of coordinates to print. If `NULL` print all of them.

`...` Passed to `format.default()`.

Method `project()`:

Usage:

```
Coord1D$project(point = as_point1d("origin"), ...)
```

Arguments:

`point` A Point1D object of length one representing the point you with to reflect across or project to or an object coercible to one by `as_point1d(point, ...)` such as "origin".

`...` Passed to `project1d()`.

Method `reflect()`:

Usage:

```
Coord1D$reflect(point = as_point1d("origin"), ...)
```

Arguments:

`point` A Point1D object of length one representing the point you with to reflect across or project to or an object coercible to one by `as_point1d(point, ...)` such as "origin".

`...` Passed to `reflect1d()`.

Method `scale()`:

Usage:

```
Coord1D$scale(x_scale = 1)
```

Arguments:

`x_scale` Scaling factor to apply to x coordinates

Method `translate()`:

Usage:

```
Coord1D$translate(x = as_coord1d(0), ...)
```

Arguments:

`x` A Coord1D object of length one or an object coercible to one by `as_coord1d(x, ...)`.

`...` Passed to `as_coord1d(x, ...)` if x is not a Coord1D object

Method `transform()`**:**

Usage:

```
Coord1D$transform(mat = transform1d())
```

Arguments:

`mat` A 2x2 matrix representing a post-multiplied affine transformation matrix. The last **column** must be equal to `c(0, 1)`. If the last **row** is `c(0, 1)` you may need to transpose it to convert it from a pre-multiplied affine transformation matrix to a post-multiplied one. If a 1x1 matrix we'll quietly add a final column/row equal to `c(0, 1)`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Coord1D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
p <- as_coord1d(x = rnorm(100, 2))
print(p, n = 10L)
pc <- mean(p) # Centroid
# method chained affine transformation matrices are auto-pre-multiplied
p$
  translate(-pc)$
  reflect("origin")$
  print(n = 10L)
```

`Coord2D` *2D coordinate vector R6 Class*

Description

`Coord2D` is an `R6::R6Class()` object representing two-dimensional points represented by Cartesian Coordinates.

Active bindings

`xyw` A three-column matrix representing the homogeneous coordinates. The first two columns are "x" and "y" coordinates and the third column is all ones.

`x` A numeric vector of x-coordinates.

`y` A numeric vector of y-coordinates.

Methods

Public methods:

- `Coord2D$new()`
- `Coord2D$permute()`
- `Coord2D$print()`
- `Coord2D$project()`
- `Coord2D$reflect()`
- `Coord2D$rotate()`
- `Coord2D$scale()`
- `Coord2D$shear()`
- `Coord2D$translate()`
- `Coord2D$transform()`
- `Coord2D$clone()`

Method `new()`**:**

Usage:

```
Coord2D$new(xyw)
```

Arguments:

`xyw` A matrix with three columns representing (homogeneous) coordinates. The first two columns represent x and y coordinates and the last column is all ones. Column names should be "x", "y", and "w".

Method `permute()`**:**

Usage:

```
Coord2D$permute(permutation = c("xy", "yx"))
```

Arguments:

`permutation` Either "xy" (no permutation) or "yx" (permute x and y axes)

Method `print()`**:**

Usage:

```
Coord2D$print(n = NULL, ...)
```

Arguments:

`n` Number of coordinates to print. If `NULL` print all of them.

`...` Passed to `format.default()`.

Method `project()`**:**

Usage:

```
Coord2D$project(line = as_line2d("x-axis"), ..., scale = 0)
```

Arguments:

`line` A Line2D object of length one representing the line you with to reflect across or project to or an object coercible to one by `as_line2d(line, ...)` such as "x-axis" or "y-axis".

`...` Passed to `project2d()`

`scale` Oblique projection scale factor. A degenerate `0` value indicates an orthogonal projection.

Method `reflect():`

Usage:

```
Coord2D$reflect(line = as_line2d("x-axis"), ...)
```

Arguments:

`line` A Line2D object of length one representing the line you with to reflect across or project to or an object coercible to one by `as_line2d(line, ...)` such as "x-axis" or "y-axis".

`...` Passed to `reflect2d()`.

Method `rotate():`

Usage:

```
Coord2D$rotate(theta = angle(0), ...)
```

Arguments:

`theta` An `angle()` object of length one or an object coercible to one by `as_angle(theta, ...)`.

`...` Passed to `as_angle()`.

Method `scale():`

Usage:

```
Coord2D$scale(x_scale = 1, y_scale = x_scale)
```

Arguments:

`x_scale` Scaling factor to apply to x coordinates

`y_scale` Scaling factor to apply to y coordinates

Method `shear():`

Usage:

```
Coord2D$shear(xy_shear = 0, yx_shear = 0)
```

Arguments:

`xy_shear` Horizontal shear factor: `x = x + xy_shear * y`

`yx_shear` Vertical shear factor: `y = yx_shear * x + y`

Method `translate():`

Usage:

```
Coord2D$translate(x = as_coord2d(0, 0), ...)
```

Arguments:

`x` A Coord2D object of length one or an object coercible to one by `as_coord2d(x, ...)`.

`...` Passed to `as_coord2d(x, ...)` if `x` is not a Coord2D object

Method `transform():`

Usage:

```
Coord2D$transform(mat = transform2d())
```

Arguments:

mat A 3x3 matrix representing a post-multiplied affine transformation matrix. The last **column** must be equal to `c(0, 0, 1)`. If the last **row** is `c(0, 0, 1)` you may need to transpose it to convert it from a pre-multiplied affine transformation matrix to a post-multiplied one. If a 2x2 matrix (such as a 2x2 post-multiplied 2D rotation matrix) we'll quietly add a final column/row equal to `c(0, 0, 1)`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Coord2D$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
p <- as_coord2d(x = rnorm(100, 2), y = rnorm(100, 2))
print(p, n = 10)
pc <- mean(p) # Centroid
# method chained affine transformation matrices are auto-pre-multiplied
p$
  translate(-pc)$
  shear(x = 1, y = 0)$
  reflect("x-axis")$
  rotate(90, "degrees")$
  print(n = 10)
```

`Coord3D` *3D coordinate vector R6 Class*

Description

`Coord3D` is an `R6::R6Class()` object representing three-dimensional points represented by Cartesian Coordinates.

Active bindings

xyzw A four-column matrix representing the homogeneous coordinates. The first three columns are "x", "y", and "z" coordinates and the fourth column is all ones.

x A numeric vector of x-coordinates.

y A numeric vector of y-coordinates.

z A numeric vector of z-coordinates.

Methods

Public methods:

- `Coord3D$new()`
- `Coord3D$permute()`
- `Coord3D$print()`
- `Coord3D$project()`
- `Coord3D$reflect()`
- `Coord3D$rotate()`
- `Coord3D$scale()`
- `Coord3D$shear()`
- `Coord3D$translate()`
- `Coord3D$transform()`
- `Coord3D$clone()`

Method `new()`:

Usage:

```
Coord3D$new(xyzw)
```

Arguments:

`xyzw` A matrix with four columns representing (homogeneous) coordinates. The first three columns represent x, y, and z coordinates and the last column is all ones. Column names should be "x", "y", "z", and "w".

Method `permute()`:

Usage:

```
Coord3D$permute(permutation = c("xyz", "xzy", "yxz", "yzx", "zyx", "zxy"))
```

Arguments:

`permutation` Either "xyz" (no permutation), "xzy" (permute y and z axes), "yxz" (permute x and y axes), "yzx" (x becomes z, y becomes x, z becomes y), "zxy" (x becomes y, y becomes z, z becomes x), "zyx" (permute x and z axes)

Method `print()`:

Usage:

```
Coord3D$print(n = NULL, ...)
```

Arguments:

`n` Number of coordinates to print. If `NULL` print all of them.

`...` Passed to `format.default()`.

Method `project()`:

Usage:

```
Coord3D$project(
  plane = as_plane3d("xy-plane"),
  ...,
  scale = 0,
  alpha = angle(45, "degrees")
)
```

Arguments:

`plane` A Plane3D object of length one representing the plane you wish to reflect across or project to or an object coercible to one using `as_plane3d(plane, ...)` such as "xy-plane", "xz-plane", or "yz-plane".

`...` Passed to `project3d()`.

`scale` Oblique projection foreshortening scale factor. A (degenerate) `0` value indicates an orthographic projection. A value of `0.5` is used by a "cabinet projection" while a value of `1.0` is used by a "cavalier projection".

`alpha` Oblique projection angle (the angle the third axis is projected going off at). An `angle()` object or one coercible to one with `as_angle(alpha, ...)`. Popular angles are 45 degrees, 60 degrees, and `arctangent(2)` degrees.

Method `reflect()`**:**

Usage:

```
Coord3D$reflect(plane = as_plane3d("xy-plane"), ...)
```

Arguments:

`plane` A Plane3D object of length one representing the plane you wish to reflect across or project to or an object coercible to one using `as_plane3d(plane, ...)` such as "xy-plane", "xz-plane", or "yz-plane".

`...` Passed to `reflect3d()`.

Method `rotate()`**:**

Usage:

```
Coord3D$rotate(axis = as_coord3d("z-axis"), theta = angle(0), ...)
```

Arguments:

`axis` A Coord3D class object or one that can coerced to one by `as_coord3d(axis, ...)`. The `axis` represents the axis to be rotated around.

`theta` An `angle()` object of length one or an object coercible to one by `as_angle(theta, ...)`.

`...` Passed to `rotate3d()`.

Method `scale()`**:**

Usage:

```
Coord3D$scale(x_scale = 1, y_scale = x_scale, z_scale = x_scale)
```

Arguments:

`x_scale` Scaling factor to apply to x coordinates

`y_scale` Scaling factor to apply to y coordinates

`z_scale` Scaling factor to apply to z coordinates

Method `shear()`**:**

Usage:

```
Coord3D$shear(
  xy_shear = 0,
  xz_shear = 0,
  yx_shear = 0,
  yz_shear = 0,
  zx_shear = 0,
  zy_shear = 0
)
```

Arguments:

`xy_shear` Shear factor: `x` = `x` + `xy_shear` * `y` + `xz_shear` * `z`

`xz_shear` Shear factor: `x` = `x` + `xy_shear` * `y` + `xz_shear` * `z`

`yx_shear` Shear factor: `y` = `yx_shear` * `x` + `y` + `yz_shear` * `z`

`yz_shear` Shear factor: `y` = `yx_shear` * `x` + `y` + `yz_shear` * `z`

`zx_shear` Shear factor: `z` = `zx_shear` * `x` + `zy_shear` * `y` + `z`

`zy_shear` Shear factor: `z` = `zx_shear` * `x` + `zy_shear` * `y` + `z`

Method `translate()`**:**

Usage:

```
Coord3D$translate(x = as_coord3d(0, 0, 0), ...)
```

Arguments:

`x` A Coord3D object of length one or an object coercible to one by `as_coord3d(x, ...)`.

`...` Passed to `as_coord3d(x, ...)` if `x` is not a Coord3D object

Method `transform()`**:**

Usage:

```
Coord3D$transform(mat = transform3d())
```

Arguments:

`mat` A 4x4 matrix representing a post-multiplied affine transformation matrix. The last **column** must be equal to `c(0, 0, 0, 1)`. If the last **row** is `c(0, 0, 0, 1)` you may need to transpose it to convert it from a pre-multiplied affine transformation matrix to a post-multiplied one. If a 3x3 matrix (such as a 3x3 post-multiplied 3D rotation matrix) we'll quietly add a final column/row equal to `c(0, 0, 0, 1)`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Coord3D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
p <- as_coord3d(x = rnorm(100, 2), y = rnorm(100, 2), z = rnorm(100, 2))
print(p, n = 10)
pc <- mean(p) # Centroid
# method chained affine transformation matrices are auto-pre-multiplied
```

```
p$
  translate(-pc)$
  reflect("xy-plane")$
  rotate("z-axis", degrees(90))$
  print(n = 10)
```

`cross_product3d` *Compute 3D vector cross product*

Description

`cross_product3d()` computes the cross product of two Coord3D class vectors.

Usage

```
cross_product3d(x, y)
```

Arguments

x	A Coord3D class vector.
y	A Coord3D class vector.

Value

A Coord3D class vector

Examples

```
x <- as_coord3d(2, 3, 4)
y <- as_coord3d(5, 6, 7)
cross_product3d(x, y)
```

`distance1d` *1D Euclidean distances*

Description

`distance1d()` computes 1D Euclidean distances.

Usage

```
distance1d(x, y)
```

Arguments

x	Either a Coord1D or Point1D class object
y	Either a Coord1D or Point1D class object

Examples

```
p <- as_coord1d(x = 1:4)
distance1d(p, as_coord1d(0))
```

---

distance2d *2D Euclidean distances*

Description

`distance2d()` computes 2D Euclidean distances.

Usage

```
distance2d(x, y)
```

Arguments

x	Either a Coord2D or Line2D class object
y	Either a Coord2D or Line2D class object

Examples

```
p <- as_coord2d(x = 1:4, y = 1:4)
distance2d(p, as_coord2d(0, 0))
```

---

distance3d *3D Euclidean distances*

Description

`distance3d()` computes 3D Euclidean distances.

Usage

```
distance3d(x, y)
```

Arguments

x	Either a Coord3D or Plane3D class object
y	Either a Coord3D or Plane3D class object

Examples

```
p <- as_coord3d(x = 1:4, y = 1:4, z = 1:4)
distance3d(p, as_coord3d("origin"))
```

`graphics` *Plot coordinates, points, lines, and planes*

Description

`plot()` plots Coord1D and Coord2D class objects while `points()` draws Coord1D and Coord2D class objects and `lines()` draws Point1D and Line2D class objects to an existing plot. If the suggested ggplot2 and rgl packages are available we also register `ggplot2::autolayer()` methods for Coord1D, Coord2D, Point1D, and Line2D class objects and a `rgl::plot3d()` method for Coord3D class objects.

Usage

```
## S3 method for class 'Coord1D'
plot(x, ...)

## S3 method for class 'Coord1D'
points(x, ...)

## S3 method for class 'Point1D'
lines(x, ...)

## S3 method for class 'Coord2D'
plot(x, ...)

## S3 method for class 'Coord2D'
points(x, ...)

## S3 method for class 'Line2D'
lines(x, ...)
```

Arguments

`x`	A supported object to plot.
`...`	Passed to the underlying plot method.

Value

Used for its side effect of drawing to the graphics device.

Examples

```
c1 <- as_coord2d(x = 0, y = 1:10)
l <- as_line2d(a = 1, b = -1, c = 0) # y = x
c2 <- c1$clone()$reflect(l)
plot(c1, xlim = c(-1, 11), ylim = c(-1, 11),
     main = "2D reflection across a line")
lines(l)
```

```
points(c2, col = "red")

c1 <- as_coord2d(x = 1:10, y = 1:10)
l <- as_line2d(a = -1, b = 0, c = 0) # x = 0
c2 <- c1$clone()$project(l)
if (require("ggplot2", quietly = TRUE,
            include.only = c("ggplot", "autolayer", "labs"))) {
  ggplot() +
      autolayer(c1) +
      autolayer(l) +
      autolayer(c2, color = "red") +
      labs(title = "2D projection onto a line")
}

c1 <- as_coord1d(x = seq.int(-4, -1))
pt <- as_point1d(a = 1, b = 0) # x = 0
c2 <- c1$clone()$reflect(pt)
plot(c1, xlim = c(-5, 5), main = "1D reflection across a point")
lines(pt)
points(c2, col = "red")

# 3D reflection across a plane
c1 <- as_coord3d(x = 1:10, y = 1:10, z = 1:10)
pl <- as_plane3d(a = 0, b = 0, c = -1, d = 2) # z = 2
c2 <- c1$clone()$reflect(pl)
if (require("rgl", quietly = TRUE,
            include.only = c("plot3d", "planes3d", "points3d"))) {
  plot3d(c1, size = 8)
  planes3d(as.data.frame(pl), d =  pl$d, color = "grey", alpha = 0.6)
  points3d(as.data.frame(c2), col = "red", size = 8)
}
```

`inverse-trigonometric-functions`

Angle vector aware inverse trigonometric functions

Description

`arcsine()`, `arccosine()`, `arctangent()`, `arcsecant()`, `arccosecant()`, and `arccotangent()` are inverse trigonometric functions that return `angle()` vectors with a user chosen angular unit.

Usage

```
arcsine(
  x,
  unit = getOption("affiner_angular_unit", "degrees"),
  tolerance = sqrt(.Machine$double.eps)
)
```

```
arccosine(
  x,
  unit = getOption("affiner_angular_unit", "degrees"),
  tolerance = sqrt(.Machine$double.eps)
)

arctangent(x, unit = getOption("affiner_angular_unit", "degrees"), y = NULL)

arcsecant(x, unit = getOption("affiner_angular_unit", "degrees"))

arccosecant(x, unit = getOption("affiner_angular_unit", "degrees"))

arccotangent(x, unit = getOption("affiner_angular_unit", "degrees"))
```

Arguments

`x`	A numeric vector
`unit`	A string of the desired angular unit. Supports the following strings (note we ignore any punctuation and space characters as well as any trailing s's e.g. "half turns" will be treated as equivalent to "halfturn"): • "deg" or "degree" • "half-revolution", "half-turn", or "pi-radian" • "gon", "grad", "grade", or "gradian" • "rad" or "radian" • "rev", "revolution", "tr", or "turn"
`tolerance`	If `x` greater than 1 (or less than -1) but is within a `tolerance` of 1 (or -1) then it will be treated as 1 (or -1)
`y`	A numeric vector or `NULL`. If `NULL` (default) we compute the 1-argument arctangent else we compute the 2-argument arctangent. For positive coordinates `(x, y)` then `arctangent(x = y/x) == arctangent(x = x, y = y)`.

Value

An `angle()` vector

Examples

```
arccosine(-1, "degrees")
arcsine(0, "turns")
arctangent(0, "gradians")
arccosecant(-1, "degrees")
arcsecant(1, "degrees")
arccotangent(1, "half-turns")

# `base::atan2(y, x)` computes the angle of the vector from origin to (x, y)
as_angle(as_coord2d(x = 1, y = 1), "degrees")
```

isocubeGrob *Isometric cube grob*

Description

`isometricCube()` is a grid grob function to render isometric cube faces by automatically wrapping around `affineGrob()`.

Usage

```
isocubeGrob(
  top,
  right,
  left,
  gp_border = grid::gpar(fill = NA, col = "black", lwd = 12),
  name = NULL,
  gp = grid::gpar(),
  vp = NULL
)

grid.isocube(...)
```

Arguments

`top`	A grid grob object to use as the top side of the cube. ggplot2 objects will be coerced by `ggplot2::ggplotGrob()`.
`right`	A grid grob object to use as the right side of the cube. ggplot2 objects will be coerced by `ggplot2::ggplotGrob()`.
`left`	A grid grob object to use as the left side of the cube. ggplot2 objects will be coerced by `ggplot2::ggplotGrob()`.
`gp_border`	A `grid::gpar()` object for the `polygonGrob()` used to draw borders around the cube faces.
`name`	A character identifier (for grid).
`gp`	A `grid::gpar()` object.
`vp`	A `grid::viewport()` object (or `NULL`).
`...`	Passed to `isocubeGrob()`

Details

Any `ggplot2` objects are coerced to grobs by `ggplot2::ggplotGrob()`. Depending on what you'd like to do you may want to instead manually convert a ggplot2 object gg to a grob with `gtable::gtable_filter(ggplot2::ggplotGrob(gg), "panel")`.

Value

A `grid::gTree()` (grob) object of class "isocube". As a side effect `grid.isocube()` draws to the active graphics device.

Examples

```
if (require("grid") &&
    getRversion() >= "4.2.0" &&
    isTRUE(dev.capabilities()$transformations)) {
  # Only works if active graphics device supports affine transformations
  # such as `png(type="cairo")` on R 4.2+
  grid.newpage()
  gp_text <- gpar(fontsize = 72)
  grid.isocube(top = textGrob("top", gp = gp_text),
               right = textGrob("right", gp = gp_text),
               left = textGrob("left", gp = gp_text))
}
if (require("grid") &&
    getRversion() >= "4.2.0" &&
    isTRUE(dev.capabilities()$transformations)) {
    colors <- c("#D55E00", "#009E73", "#56B4E9")
    spacings <- c(0.25, 0.2, 0.25)
    texts <- c("pkgname", "left\nface", "right\nface")
    rots <- c(45, 0, 0)
    fontsizes <- c(52, 80, 80)
    sides <- c("top", "left", "right")
    types <- gridpattern::names_polygon_tiling[c(5, 7, 9)]
    l_grobs <- list()
    grid.newpage()
    for (i in 1:3) {
        if (requireNamespace("gridpattern", quietly = TRUE)) {
            bg <- gridpattern::grid.pattern_polygon_tiling(
                       colour = "grey80",
                       fill = c(colors[i], "white"),
                       type = types[i],
                       spacing = spacings[i],
                       draw = FALSE)
        } else {
            bg <- rectGrob(gp = gpar(col = NA, fill = colors[i]))
        }
        text <- textGrob(texts[i], rot = rots[i],
                         gp = gpar(fontsize = fontsizes[i]))
        l_grobs[[sides[i]]] <- grobTree(bg, text)
    }
  grid.newpage()
  grid.isocube(top = l_grobs$top,
               right = l_grobs$right,
               left = l_grobs$left)
}
```

is_angle — *Test whether an object is an angle vector*

Description

`is_angle()` tests whether an object is an angle vector

Usage

```
is_angle(x)
```

Arguments

x	An object

Value

A logical value

Examples

```
a <- angle(180, "degrees")
is_angle(a)
is_angle(pi)
```

is_congruent — *Test whether two objects are congruent*

Description

`is_congruent()` is a S3 generic that tests whether two different objects are "congruent". The `is_congruent()` method for `angle()` classes tests whether two angles are congruent.

Usage

```
is_congruent(x, y, ...)

## S3 method for class 'numeric'
is_congruent(x, y, ..., tolerance = sqrt(.Machine$double.eps))

## S3 method for class 'angle'
is_congruent(
  x,
  y,
  ...,
  mod_turns = TRUE,
  tolerance = sqrt(.Machine$double.eps)
)
```

Arguments

x, y	Two objects to test whether they are “"congruent"”.
...	Further arguments passed to or from other methods.
tolerance	Angles (coerced to half-turns) or numerics with differences smaller than `tolerance` will be considered “congruent”.
mod_turns	If `TRUE` angles that are congruent modulo full turns will be considered “congruent”.

Value

A logical vector

Examples

```
# Use `is_congruent()` to check if two angles are "congruent"
a1 <- angle(180, "degrees")
a2 <- angle(pi, "radians")
a3 <- angle(-180, "degrees") # Only congruent modulus full turns
a1 == a2
isTRUE(all.equal(a1, a2))
is_congruent(a1, a2)
is_congruent(a1, a2, mod_turns = FALSE)
a1 == a3
isTRUE(all.equal(a1, a3))
is_congruent(a1, a3)
is_congruent(a1, a3, mod_turns = FALSE)
```

---

is_coord1d — *Test whether an object has a Coord1D class*

Description

`is_coord1d()` tests whether an object has a "Coord1D" class

Usage

```
is_coord1d(x)
```

Arguments

x	An object

Value

A logical value

Examples

```
p <- as_coord1d(x = sample.int(10, 3))
is_coord1d(p)
```

---

is_coord2d *Test whether an object has a Coord2D class*

Description

`is_coord2d()` tests whether an object has a "Coord2D" class

Usage

```
is_coord2d(x)
```

Arguments

x	An object

Value

A logical value

Examples

```
p <- as_coord2d(x = sample.int(10, 3), y = sample.int(10, 3))
is_coord2d(p)
```

---

is_coord3d *Test whether an object has a Coord3D class*

Description

`is_coord3d()` tests whether an object has a "Coord3D" class

Usage

```
is_coord3d(x)
```

Arguments

x	An object

Value

A logical value

Examples

```
p <- as_coord3d(x = sample.int(10, 3),
                y = sample.int(10, 3),
                z = sample.int(10, 3))
is_coord3d(p)
```

---

is_line2d — *Test whether an object has a Line2D class*

Description

`is_line2d()` tests whether an object has a "Line2D" class

Usage

```
is_line2d(x)
```

Arguments

x	An object

Value

A logical value

Examples

```
l <- as_line2d(a = 1, b = 2, c = 3)
is_line2d(l)
```

---

is_plane3d — *Test whether an object has a Plane3D class*

Description

`is_plane3d()` tests whether an object has a "Plane3D" class

Usage

```
is_plane3d(x)
```

Arguments

x	An object

Value

A logical value

Examples

```
p <- as_plane3d(a = 1, b = 2, c = 3, 4)
is_plane3d(p)
```

---

is_point1d — *Test whether an object has a Point1D class*

Description

`is_point1d()` tests whether an object has a "Point1D" class

Usage

```
is_point1d(x)
```

Arguments

`x`	An object

Value

A logical value

Examples

```
p <- as_point1d(a = 1, b = 5)
is_point1d(p)
```

---

is_transform1d — *Test if 1D affine transformation matrix*

Description

`is_transform1d()` tests if object is a `transform1d()` affine transformation matrix

Usage

```
is_transform1d(x)
```

Arguments

`x`	An object

Value

A logical value

Examples

```
m <- transform1d(diag(2L))
is_transform1d(m)
is_transform1d(diag(2L))
```

---

is_transform2d   *Test if 2D affine transformation matrix*

Description

`is_transform2d()` tests if object is a `transform2d()` affine transformation matrix

Usage

```
is_transform2d(x)
```

Arguments

`x`	An object

Value

A logical value

Examples

```
m <- transform2d(diag(3L))
is_transform2d(m)
is_transform2d(diag(3L))
```

---

is_transform3d   *Test if 3D affine transformation matrix*

Description

`is_transform3d()` tests if object is a `transform3d()` affine transformation matrix

Usage

```
is_transform3d(x)
```

Arguments

x An object

Value

A logical value

Examples

```
m <- transform3d(diag(4L))
is_transform3d(m)
is_transform3d(diag(4L))
```

`Line2D` *2D lines R6 Class*

Description

`Line2D` is an `R6::R6Class()` object representing two-dimensional lines.

Public fields

a Numeric vector that parameterizes the line via the equation `a * x + b * y + c = 0`.

b Numeric vector that parameterizes the line via the equation `a * x + b * y + c = 0`.

c Numeric vector that parameterizes the line via the equation `a * x + b * y + c = 0`.

Methods

Public methods:

- `Line2D$new()`
- `Line2D$print()`
- `Line2D$clone()`

Method `new()`:

Usage:

```
Line2D$new(a, b, c)
```

Arguments:

a Numeric vector that parameterizes the line via the equation `a * x + b * y + c = 0`.

b Numeric vector that parameterizes the line via the equation `a * x + b * y + c = 0`.

c Numeric vector that parameterizes the line via the equation `a * x + b * y + c = 0`.

Method `print()`:

Usage:

```
Line2D$print(n = NULL, ...)
```

Arguments:

`n` Number of lines to print. If `NULL` print all of them.

`...` Passed to `format.default()`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Line2D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
p1 <- as_coord2d(x = 5, y = 10)
p2 <- as_coord2d(x = 7, y = 12)
theta <- degrees(45)
as_line2d(theta, p1)
as_line2d(p1, p2)
```

`normal2d` *2D normal vectors*

Description

`normal2d()` is an S3 generic that computes a 2D normal vector.

Usage

```
normal2d(x, ...)

## S3 method for class 'Coord2D'
normal2d(x, ..., normalize = TRUE)

## S3 method for class 'Line2D'
normal2d(x, ..., normalize = TRUE)
```

Arguments

`x`	Object to compute a 2D normal vector for such as a Line2D object.
`...`	Passed to or from other methods.
`normalize`	If `TRUE` coerce to a normalize vector

Value

A Coord2D (normal) vector

Examples

```
p <- as_coord2d(x = 2, y = 3)
normal2d(p)
normal2d(p, normalize = FALSE)
```

---

normal3d *3D normal vectors*

Description

`normal3d()` is an S3 generic that computes a 3D normal vector.

Usage

```
normal3d(x, ...)

## S3 method for class 'Coord3D'
normal3d(x, cross, ..., normalize = TRUE)

## S3 method for class 'character'
normal3d(x, ..., normalize = TRUE)

## S3 method for class 'Plane3D'
normal3d(x, ..., normalize = TRUE)
```

Arguments

`x`	Object to compute a 3D normal vector for such as a Plane3D object
`...`	Passed to other methods such as `as_coord3d()`.
`cross`	A Coord3D vector. We'll compute the normal of `x` and `cross` by taking their cross product.
`normalize`	If `TRUE` normalize to a unit vector

Value

A Coord3D (normal) vector

Examples

```
normal3d("xy-plane")
normal3d(as_coord3d(2, 0, 0), cross = as_coord3d(0, 2, 0))
```

`Plane3D` *3D planes R6 Class*

Description

`Plane3D` is an `R6::R6Class()` object representing three-dimensional planes.

Public fields

a Numeric vector that parameterizes the plane via the equation `a * x + b * y + c * z + d = 0`.

b Numeric vector that parameterizes the plane via the equation `a * x + b * y + c * z + d = 0`.

c Numeric vector that parameterizes the plane via the equation `a * x + b * y + c * z + d = 0`.

d Numeric vector that parameterizes the plane via the equation `a * x + b * y + c * z + d = 0`.

Methods

Public methods:

- `Plane3D$new()`
- `Plane3D$print()`
- `Plane3D$clone()`

Method `new()`**:**

Usage:

```
Plane3D$new(a, b, c, d)
```

Arguments:

a Numeric vector that parameterizes the plane via the equation `a * x + b * y + c * z + d = 0`.

b Numeric vector that parameterizes the plane via the equation `a * x + b * y + c * z + d = 0`.

c Numeric vector that parameterizes the plane via the equation `a * x + b * y + c * z + d = 0`.

d Numeric vector that parameterizes the plane via the equation `a * x + b * y + c * z + d = 0`.

Method `print()`**:**

Usage:

```
Plane3D$print(n = NULL, ...)
```

Arguments:

n Number of lines to print. If `NULL` print all of them.

... Passed to `format.default()`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Plane3D$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Point1D *1D points R6 Class*

Description

Point1D is an R6::R6Class() object representing one-dimensional points.

Public fields

a Numeric vector that parameterizes the point via the equation a * x + b = 0.

b Numeric vector that parameterizes the point via the equation a * x + b = 0.

Methods

Public methods:

- Point1D$new()
- Point1D$print()
- Point1D$clone()

Method new()**:**

Usage:

```
Point1D$new(a, b)
```

Arguments:

a Numeric vector that parameterizes the line via the equation a * x + b = 0.

b Numeric vector that parameterizes the line via the equation a * x + b = 0.

Method print()**:**

Usage:

```
Point1D$print(n = NULL, ...)
```

Arguments:

n Number of lines to print. If NULL print all of them.

... Passed to format.default().

Method clone()**:** The objects of this class are cloneable with this method.

Usage:

```
Point1D$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
p1 <- as_point1d(a = 1, b = 5)
```

rotate3d_to_AA *Convert from 3D rotation matrix to axis-angle representation.*

Description

`rotate3d_to_AA()` converts from (post-multiplied) rotation matrix to an axis-angle representation of 3D rotations.

Usage

```
rotate3d_to_AA(
  mat = diag(4),
  unit = getOption("affiner_angular_unit", "degrees")
)
```

Arguments

`mat`	3D rotation matrix (post-multiplied). If you have a pre-multiplied rotation matrix simply transpose it with `t()` to get a post-multiplied rotation matrix.
`unit`	A string of the desired angular unit. Supports the following strings (note we ignore any punctuation and space characters as well as any trailing s's e.g. "half turns" will be treated as equivalent to "halfturn"): • "deg" or "degree" • "half-revolution", "half-turn", or "pi-radian" • "gon", "grad", "grade", or "gradian" • "rad" or "radian" • "rev", "revolution", "tr", or "turn"

See Also

https://en.wikipedia.org/wiki/Axis-angle_representation for more details about the Axis-angle representation of 3D rotations. `rotate3d()` can be used to convert from an axis-angle representation to a rotation matrix.

Examples

```
# axis-angle representation of 90 degree rotation about the x-axis
rotate3d_to_AA(rotate3d("x-axis", 90, unit = "degrees"))

# find Axis-Angle representation of first rotating about x-axis 180 degrees
# and then rotating about z-axis 45 degrees
R <- rotate3d("x-axis", 180, unit = "degrees") %*%
       rotate3d("z-axis", 45, unit = "degrees")
AA <- rotate3d_to_AA(R)

# Can use `rotate3d()` to convert back to rotation matrix representation
all.equal(R, do.call(rotate3d, AA))
```

`transform1d` *1D affine transformation matrices*

Description

`transform1d()`, `reflect1d()`, `scale2d()`, and `translate1d()` create 1D affine transformation matrix objects.

Usage

```
transform1d(mat = diag(2L))

project1d(point = as_point1d("origin"), ...)

reflect1d(point = as_point1d("origin"), ...)

scale1d(x_scale = 1)

translate1d(x = as_coord1d(0), ...)
```

Arguments

`mat`	A 2x2 matrix representing a post-multiplied affine transformation matrix. The last **column** must be equal to `c(0, 1)`. If the last **row** is `c(0, 1)` you may need to transpose it to convert it from a pre-multiplied affine transformation matrix to a post-multiplied one. If a 1x1 matrix we'll quietly add a final column/row equal to `c(0, 1)`.
`point`	A Point1D object of length one representing the point you with to reflect across or project to or an object coercible to one by `as_point1d(point, ...)` such as "origin".
`...`	Passed to `as_coord1d()`.
`x_scale`	Scaling factor to apply to x coordinates
`x`	A Coord1D object of length one or an object coercible to one by `as_coord1d(x, ...)`.

Details

`transform1d()` User supplied (post-multiplied) affine transformation matrix.

`reflect1d()` Reflections across a point.

`scale1d()` Scale the x-coordinates by multiplicative scale factors.

`translate1d()` Translate the coordinates by a Coord1D class object parameter.

`transform1d()` 1D affine transformation matrix objects are meant to be post-multiplied and therefore should **not** be multiplied in reverse order. Note the Coord1D class object methods auto-pre-multiply affine transformations when "method chaining" so pre-multiplying affine transformation matrices to do a single cumulative transformation instead of a method chain of multiple transformations will not improve performance as much as as it does in other R packages.

To convert a pre-multiplied 1D affine transformation matrix to a post-multiplied one simply compute its transpose using `t()`. To get an inverse transformation matrix from an existing transformation matrix that does the opposite transformations simply compute its inverse using `solve()`.

Value

A 2x2 post-multiplied affine transformation matrix with classes "transform1d" and "at_matrix"

Examples

```
p <- as_coord1d(x = sample(1:10, 3))

# {affiner} affine transformation matrices are post-multiplied
# and therefore should **not** go in reverse order
mat <- transform1d(diag(2)) %*%
         scale1d(2) %*%
         translate1d(x = -1)
p1 <- p$
  clone()$
  transform(mat)

# The equivalent result appyling affine transformations via method chaining
p2 <- p$
  clone()$
  transform(diag(2))$
  scale(2)$
  translate(x = -1)

all.equal(p1, p2)
```

`transform2d` *2D affine transformation matrices*

Description

`transform2d()`, `project2d()`, `reflect2d()`, `rotate2d()`, `scale2d()`, `shear2d()`, and `translate2d()` create 2D affine transformation matrix objects.

Usage

```
transform2d(mat = diag(3L))

permute2d(permutation = c("xy", "yx"))

project2d(line = as_line2d("x-axis"), ..., scale = 0)

reflect2d(line = as_line2d("x-axis"), ...)
```

```
rotate2d(theta = angle(0), ...)

scale2d(x_scale = 1, y_scale = x_scale)

shear2d(xy_shear = 0, yx_shear = 0)

translate2d(x = as_coord2d(0, 0), ...)
```

Arguments

mat	A 3x3 matrix representing a post-multiplied affine transformation matrix. The last **column** must be equal to `c(0, 0, 1)`. If the last **row** is `c(0, 0, 1)` you may need to transpose it to convert it from a pre-multiplied affine transformation matrix to a post-multiplied one. If a 2x2 matrix (such as a 2x2 post-multiplied 2D rotation matrix) we'll quietly add a final column/row equal to `c(0, 0, 1)`.
permutation	Either "xy" (no permutation) or "yx" (permute x and y axes)
line	A Line2D object of length one representing the line you with to reflect across or project to or an object coercible to one by `as_line2d(line, ...)` such as "x-axis" or "y-axis".
...	Passed to `as_angle()` or `as_coord2d()`.
scale	Oblique projection scale factor. A degenerate `0` value indicates an orthogonal projection.
theta	An `angle()` object of length one or an object coercible to one by `as_angle(theta, ...)`.
x_scale	Scaling factor to apply to x coordinates
y_scale	Scaling factor to apply to y coordinates
xy_shear	Horizontal shear factor: `x = x + xy_shear * y`
yx_shear	Vertical shear factor: `y = yx_shear * x + y`
x	A Coord2D object of length one or an object coercible to one by `as_coord2d(x, ...)`.

Details

`transform2d()` User supplied (post-multiplied) affine transformation matrix.

`project2d()` Oblique vector projections onto a line parameterized by an oblique projection scale factor. A (degenerate) scale factor of zero results in an orthogonal projection.

`reflect2d()` Reflections across a line. To "flip" across both the x-axis and the y-axis use `scale2d(-1)`.

`rotate2d()` Rotations around the origin parameterized by an `angle()`.

`scale2d()` Scale the x-coordinates and/or the y-coordinates by multiplicative scale factors.

`shear2d()` Shear the x-coordinates and/or the y-coordinates using shear factors.

`translate2d()` Translate the coordinates by a Coord2D class object parameter.

`transform2d()` 2D affine transformation matrix objects are meant to be post-multiplied and therefore should **not** be multiplied in reverse order. Note the Coord2D class object methods auto-pre-multiply affine transformations when "method chaining" so pre-multiplying affine transformation matrices to do a single cumulative transformation instead of a method chain of multiple transformations will not improve performance as much as as it does in other R packages.

To convert a pre-multiplied 2D affine transformation matrix to a post-multiplied one simply compute its transpose using `t()`. To get an inverse transformation matrix from an existing transformation matrix that does the opposite transformations simply compute its inverse using `solve()`.

Value

A 3x3 post-multiplied affine transformation matrix with classes "transform2d" and "at_matrix"

Examples

```
p <- as_coord2d(x = sample(1:10, 3), y = sample(1:10, 3))

# {affiner} affine transformation matrices are post-multiplied
# and therefore should **not** go in reverse order
mat <- transform2d(diag(3)) %*%
         reflect2d(as_coord2d(-1, 1)) %*%
         rotate2d(90, "degrees") %*%
         scale2d(1, 2) %*%
         shear2d(0.5, 0.5) %*%
         translate2d(x = -1, y = -1)
p1 <- p$
  clone()$
  transform(mat)

# The equivalent result appyling affine transformations via method chaining
p2 <- p$
  clone()$
  transform(diag(3L))$
  reflect(as_coord2d(-1, 1))$
  rotate(90, "degrees")$
  scale(1, 2)$
  shear(0.5, 0.5)$
  translate(x = -1, y = -1)

all.equal(p1, p2)
```

---

transform3d *3D affine transformation matrices*

Description

`transform3d()`, `project3d()`, `reflect3d()`, `rotate3d()`, `scale3d()`, `shear3d()`, and `translate3d()` create 3D affine transformation matrix objects.

Usage

```
transform3d(mat = diag(4L))

permute3d(permutation = c("xyz", "xzy", "yxz", "yzx", "zyx", "zxy"))

project3d(
  plane = as_plane3d("xy-plane"),
  ...,
  scale = 0,
  alpha = angle(45, "degrees")
)

reflect3d(plane = as_plane3d("xy-plane"), ...)

rotate3d(axis = as_coord3d("z-axis"), theta = angle(0), ...)

scale3d(x_scale = 1, y_scale = x_scale, z_scale = x_scale)

shear3d(
  xy_shear = 0,
  xz_shear = 0,
  yx_shear = 0,
  yz_shear = 0,
  zx_shear = 0,
  zy_shear = 0
)

translate3d(x = as_coord3d(0, 0, 0), ...)
```

Arguments

`mat`	A 4x4 matrix representing a post-multiplied affine transformation matrix. The last **column** must be equal to `c(0, 0, 0, 1)`. If the last **row** is `c(0, 0, 0, 1)` you may need to transpose it to convert it from a pre-multiplied affine transformation matrix to a post-multiplied one. If a 3x3 matrix (such as a 3x3 post-multiplied 3D rotation matrix) we'll quietly add a final column/row equal to `c(0, 0, 0, 1)`.
`permutation`	Either "xyz" (no permutation), "xzy" (permute y and z axes), "yxz" (permute x and y axes), "yzx" (x becomes z, y becomes x, z becomes y), "zxy" (x becomes y, y becomes z, z becomes x), "zyx" (permute x and z axes)
`plane`	A Plane3D object of length one representing the plane you wish to reflect across or project to or an object coercible to one using `as_plane3d(plane, ...)` such as "xy-plane", "xz-plane", or "yz-plane".
`...`	Passed to `as_angle()` or `as_coord3d()`.
`scale`	Oblique projection foreshortening scale factor. A (degenerate) `0` value indicates an orthographic projection. A value of `0.5` is used by a "cabinet projection" while a value of `1.0` is used by a "cavalier projection".

alpha	Oblique projection angle (the angle the third axis is projected going off at). An angle() object or one coercible to one with as_angle(alpha, ...). Popular angles are 45 degrees, 60 degrees, and arctangent(2) degrees.
axis	A Coord3D class object or one that can coerced to one by as_coord3d(axis, ...). The axis represents the axis to be rotated around.
theta	An angle() object of length one or an object coercible to one by as_angle(theta, ...).
x_scale	Scaling factor to apply to x coordinates
y_scale	Scaling factor to apply to y coordinates
z_scale	Scaling factor to apply to z coordinates
xy_shear	Shear factor: x = x + xy_shear * y + xz_shear * z
xz_shear	Shear factor: x = x + xy_shear * y + xz_shear * z
yx_shear	Shear factor: y = yx_shear * x + y + yz_shear * z
yz_shear	Shear factor: y = yx_shear * x + y + yz_shear * z
zx_shear	Shear factor: z = zx_shear * x + zy_shear * y + z
zy_shear	Shear factor: z = zx_shear * x + zy_shear * y + z
x	A Coord3D object of length one or an object coercible to one by as_coord3d(x, ...).

Details

transform3d() User supplied (post-multiplied) affine transformation matrix.

scale3d() Scale the x-coordinates and/or the y-coordinates and/or the z-coordinates by multiplicative scale factors.

shear3d() Shear the x-coordinates and/or the y-coordinates and/or the z-coordinates using shear factors.

translate3d() Translate the coordinates by a Coord3D class object parameter.

transform3d() 3D affine transformation matrix objects are meant to be post-multiplied and therefore should **not** be multiplied in reverse order. Note the Coord3D class object methods auto-pre-multiply affine transformations when "method chaining" so pre-multiplying affine transformation matrices to do a single cumulative transformation instead of a method chain of multiple transformations will not improve performance as much as as it does in other R packages.

To convert a pre-multiplied 3D affine transformation matrix to a post-multiplied one simply compute its transpose using t(). To get an inverse transformation matrix from an existing transformation matrix that does the opposite transformations simply compute its inverse using solve().

Value

A 4x4 post-multiplied affine transformation matrix with classes "transform3d" and "at_matrix"

Examples

```
p <- as_coord3d(x = sample(1:10, 3), y = sample(1:10, 3), z = sample(1:10, 3))

# {affiner} affine transformation matrices are post-multiplied
# and therefore should **not** go in reverse order
mat <- transform3d(diag(4L)) %*%
         rotate3d("z-axis", degrees(90)) %*%
         scale3d(1, 2, 1) %*%
         translate3d(x = -1, y = -1, z = -1)
p1 <- p$
  clone()$
  transform(mat)

# The equivalent result appyling affine transformations via method chaining
p2 <- p$
  clone()$
  transform(diag(4L))$
  rotate("z-axis", degrees(90))$
  scale(1, 2, 1)$
  translate(x = -1, y = -1, z = -1)

all.equal(p1, p2)
```

---

trigonometric-functions

Angle vector aware trigonometric functions

Description

`sine()`, `cosine()`, `tangent()`, `secant()`, `cosecant()`, and `cotangent()` are `angle()` aware trigonometric functions that allow for a user chosen angular unit.

Usage

```
sine(x, unit = getOption("affiner_angular_unit", "degrees"))

cosine(x, unit = getOption("affiner_angular_unit", "degrees"))

tangent(x, unit = getOption("affiner_angular_unit", "degrees"))

secant(x, unit = getOption("affiner_angular_unit", "degrees"))

cosecant(x, unit = getOption("affiner_angular_unit", "degrees"))

cotangent(x, unit = getOption("affiner_angular_unit", "degrees"))
```

Arguments

`x`	An angle vector or an object to convert to it (such as a numeric vector)
`unit`	A string of the desired angular unit. Supports the following strings (note we ignore any punctuation and space characters as well as any trailing `s`'s e.g. "half turns" will be treated as equivalent to "halfturn"): • "deg" or "degree" • "half-revolution", "half-turn", or "pi-radian" • "gon", "grad", "grade", or "gradian" • "rad" or "radian" • "rev", "revolution", "tr", or "turn"

Value

A numeric vector

Examples

```
sine(pi, "radians")
cosine(180, "degrees")
tangent(0.5, "turns")

a <- angle(0.5, "turns")
secant(a)
cosecant(a)
cotangent(a)
```

Index