Package ‘zonohedra’

February 1, 2025

Version 0.4-0

Date 2025-02-01

Title Compute and Plot Zonohedra from Vector Generators

Depends R (>= 4.0.0)

Imports logger

Suggests rgl, orientlib, microbenchmark, arrangements, knitr,
rmarkdown, gifski

Description Computes a zonohedron from real vector generators. The package also computes zonogons (2D zonotopes) and zonosegs (1D zonotopes). An elementary S3 class for matroids is included, which supports matroids with rank 3, 2, and 1. Optimization methods are taken from Heckbert (1985) <https://www.cs.cmu.edu/~ph/zono.ps.gz>.

License GPL (>= 2)

LazyLoad yes

LazyData yes

NeedsCompilation yes

Biarch no

Author Glenn Davis [aut, cre]

Maintainer Glenn Davis <gdavis@gluonics.com>

Repository CRAN

VignetteBuilder knitr

BuildVignettes yes

ByteCompile no

Date/Publication 2025-02-01 00:40:02 UTC

Contents

`zonohedra-package` *zonhedra package*

Description

This package deals with *zonohedra*, which are zonotopes of dimension 3. It also handles *zonogons* (2D zonotopes) and *zonosegs* (1D zonotopes).

The term *zonoseg* ("zonotope" + "segment") is my own personal term; I could not find an alternative term. It is a linear image of the unit cube $[0, 1]^n$ in the real numbers, and a compact segment of reals.

S3 classes

`Z`	`class(Z)`
zonohedron	`"zonohedron" "zonotope" "list"`
zonogon	`"zonogon" "zonotope" "list"`
zonoseg	`"zonoseg" "zonotope" "list"`

For example, the `section()` returns very diffferent things for a zonohedron and a zonogon, and so `section.zonohedron()` and `section.zonogon()` are coded and documented separately. A section for a zonoseg does not make sense, so `section.zonoseg()` is undefined.

Terminology

For a convex polytope, a *supporting hyperplane* is a hyperplane that intersect the polytope's boundary but *not* its interior.
A zonotope is a convex polytope. A zonohedron has supporting planes, and a zonogon has supporting lines.

In the package **zonohedra**, a *zonotope* mean a zonotope of dimension 3, 2, or 1.

A *face* of a zonotope is the intersection of the boundary of the zonotope with some supporting hyperplane. A *d-face* is a face of dimension *d*. So a *0-face* is a *vertex*, and a *1-face* is an *edge*.

A *facet* of a zonotope is a face whose dimension is 1 less than the dimension of the zonotope. A facet is a maximal proper face.

A zonohedron has 0-faces (vertices), 1-faces (edges), and 2-faces (facets).

A zonogon has 0-faces (vertices) and 1-faces (edges). Since the dimension of an edge is 1 less than the dimension of the zonogon, an edge of a zonogon is also a facet of a zonogon.

`boundarypgramdata` *compute data about specific parallelograms in the boundary of a zonohedron*

Description

The boundary of a zonohedron is the union of parallelograms, where some of them may be facets, and some may be tiles in the standard tiling of more complex facets. The edges of each parallelogram are given by a pair of distinct generators. If a zonohedron has n of these generators, then there are $n(n-1)/2$ such pairs. However, if the two generators are multiples of each other, or 0, the parallelogram is degenerate and does not count. For each pair of generators, there are 2 parallelograms which are antipodal to each other. The total number of parallelograms is $n(n-1)$. This function computes data about one parallelogram from this antipodal pair.

Usage

```
boundarypgramdata( x, gndpair, cube=FALSE )
```

Arguments

`x`	a **zonohedron** object as returned by the constructor `zonohedron()`
`gndpair`	an Mx2 integer matrix. Each row of `gndpair` must contain a pair of points in the ground set of the matroid of the zonohedron `x`. If the 1st point is less that the 2nd point, then the standard parallelogram is used. If the 2nd point is less than the 1st point, then the antipodal parallelogram is used. If one of the points is not in the ground set of the matroid, it is a silent error and all the returned data is set to `NA`. If the two vector generators are multiples of each other, or 0, then the parallelogram is degenerate and all the returned data is set to `NA`. `gndpair` can also be a numeric vector that can be converted to such a matrix, by row.
`cube`	if `TRUE`, then a point of the cube that maps to the center of the given parallelogram is returned, see **Value**.

Value

`boundarypgramdata()` returns a `data.frame` with M rows and these columns:

`gndpair`	the given `gndpair`
`hyperplaneidx`	the index of the hyperplane in the simplified matroid of `x` that contains `gndpair`
`center`	the center of the standard or antipodal parallelogram. The centers of the standard and antipodal parallelograms add to white.
`transitions`	the number of transitions in `pcube` - a point in the n-cube that maps to `center`, where `n` is the number of generators of `x`. This is a positive even integer.

And if `cube` is `TRUE`, then this column is added:

`pcube`	a point in the n-cube that maps to `center`. The sum of the spectra of the standard and the antipodal spectra is identically 1.

If a row of `gndpair` has an invalid pair, the other columns are filled with `NA`s.

In case of global error, the function returns `NULL`.

See Also

`zonohedron()`

Examples

```
zono =  zonohedron( colorimetry.genlist[[2]] )
boundarypgramdata( zono, c(570,608,  608,570, 400,450,  650,700,  650,720, 700,720,  650,900) )
##   gndpair.1 gndpair.2 hyperplaneidx     center.x     center.y     center.z transitions
## 1       570       608         49284  34.01432310  23.49690880   0.03214207           8
## 2       608       570         49284  72.85114639  83.36000830 106.86010920           8
## 3       400       450         12831   9.89612333   0.57529647  49.17990701           2
## 4       650       700             1   4.58023729   1.69316773   0.00000000           2
## 5       650       720             1   4.70484309   1.73816516   0.00000000           2
## 6       700       720            NA           NA           NA           NA          NA
## 7       650       900            NA           NA           NA           NA          NA

# In rows 1 and 2, the ground pairs are swapped, so the hyperlane index remains the same
# but the parallelograms are antipodal; the sum of their centers is the white point.
# Row 3 is a parallelogram facet, which is the usual situation.
# In rows 4 and 5, since generators for ground points 700 and 720 are multiples,
# the hyperplane index is the same. Both parallelograms are in a tiling of a non-trivial facet.
# In row 6, since the generators are multiples, the parallelogram is degenerate.
# In row 7, the point 900 is not in the ground set, so the parallelogram is undefined.
```

`genlist` *zonohedra generators useful for testing and plotting*

Description

`classics.genlist`	13 classic zonohedra generators
`colorimetry.genlist`	4 sets of Color Matching Functions (each set is a 3xN matrix)

Format

Each is an S3 class **genlist** object organized as a list of 3xN matrices (N varies). The list must have names, preferably short names or abbreviations. Each matrix can have optional attributes `"shortname"` and `"fullname"` which are useful when printing with `print.genlist()`.

Note

Making these S3 class **genlist** makes it possible to easily print a short summary using `print.genlist()`.

For colorimetry.genlist[[2]] a few remarks are in order. These generators come from the xyz CIE color matching functions of 1931, from 360 to 830 nm with 1 nm step. From 699 to 830 nm, the angles between the generators only differ by a few microradians, and it apparent that the designers tapered all 3 functions identically in that nm range. For an illustration of this in the chromaticity domain, see Burns, Figure 10. When the zonohedron is constructed from these 132 generators, with the default options, all these generators a 'collapsed' into a single one. In the original matroid these 132 *points* form a *multiple group*, and in the simplified matroid they are collapsed to a single *point*, labeled with 699.

Source

David Eppstein. **Zonohedra and Zonotopes**.
https://www.ics.uci.edu/~eppstein/junkyard/ukraine/ukraine.html

Colour & Vision Research Laboratory. University College London. http://www.cvrl.org

References

ASTM E 308 - 01. Standard Practice for Computing the Colors of Objects by Using the CIE System. Table 1

Scott A Burns. **The location of optimal object colors with more than two transitions**. Color Research & Application. Vol. 46. No. 6. pp 1180-1193. 2021.

Günther Wyszecki and W.S. Stiles. **Color Science : Concepts and Methods, Quantitative Data and Formulae**. Second Edition. Wiley-Interscience. 1982. Table I(3.3.1). pp. 723-735.

See Also

print.genlist()

Examples

```
# get the names of 3 sets of color matching functions
names(colorimetry.genlist)
# [1] "xyz1931.5nm" "xyz1931.1nm" "lms2000.1nm"


# print zonohedra metrics associated with 3 sets of color matching functions
colorimetry.genlist
#                      fullname generators vertices  edges facets     area    volume pointed
# xyz1931.5nm  xyz at 5nm step           81    5100  10146   5048  1582.722   4070.345   TRUE
# xyz1931.1nm  xyz at 1nm step          471   112910 225720 112812 39586.707 509434.149   TRUE
# lms2000.1nm  lms at 1nm step          441   146642 292860 146220 22736.652 181369.085   TRUE
# ciexyzjv.5nm xyz at 5nm step (1978)      90    8012  16020   8010  1553.535   3951.899   TRUE
```

`getmetrics.zonohedron` *Get Important Metrics about a Zonohedron, and Print and Summarize Them*

Description

Get some important zonohedron metrics; for most some computation is needed.

The `print()` function prints nicely formatted facts about a zonohedron, including its matroid.

The `summary()` function prints a single-line summary, formatted as a row in a data frame.

Usage

```
## S3 method for class 'zonohedron'
getmetrics( x )

## S3 method for class 'zonohedron'
print( x, trans2=FALSE, matroid=TRUE, ... )

## S3 method for class 'zonohedron'
summary( object, ... )
```

Arguments

`x`	a `zonohedron` object as returned by the constructor `zonohedron()`
`trans2`	if `TRUE` then print extra metrics on the 2-transition surface associated with `x`
`matroid`	if `TRUE` then print extra information about the matroid associated with `x`
`object`	a `zonohedron` object as returned by the constructor `zonohedron()`
`...`	for `print()` further arguments are ignored; for `summary()` the further arguments can be *more* zonohedron objects, which are summarized by adding more rows to the same data frame; see **Examples**.

Value

`getmetrics.zonohedron()` returns a list with these items:

`vertices`	the number of vertices
`edges`	the number of edges
`facets`	the number of facets (2D faces); all of them are zonogons
`area`	the sum of the areas of all the facets
`volume`	as a polytope

All of these are always positive.

`print.zonohedron()` returns `TRUE` or `FALSE`.

`summary.zonohedron()` returns a data frame, see **Examples**.

See Also

`genlist`, `zonohedron()`,

Examples

```
zono = zonohedron( classics.genlist[['BD']] )
zono
# zonohedron:
# fullname:                          Bilinski dodecahedron
# generators (original):             4
# generators with multiples:         0
# generators (simplified):           4
# number of facets:                  12  [6 antipodal facet-pairs]
# facets that contain 0:             4    { 1 3 4 6 }
# number of edges:                   24
# center:                            0.809017 2.118034 1.309017
# pointed:                           TRUE
# salient:                           TRUE
# area:                              38.83282
# volume:                            16.94427
#
# matroid:
# ground set:           4 points   {1 2 3 4}
# hyperplanes:          6     {1 2}  {1 3}  {1 4}  {2 3}  {2 4}  {3 4}
# rank:                 3
# loops:                0   {}
# multiple groups:      0   {}
# uniform:              TRUE
# paving:               TRUE
# simple:               TRUE
# This matroid is constructed from a 3x4 real matrix.
#            1        2        3         4
# [1,] 1.000000 1.618034 0.000000 -1.000000
# [2,] 1.618034 0.000000 1.000000  1.618034
# [3,] 0.000000 1.000000 1.618034  0.000000


summary( zono )
#                fullname generators vertices edges facets     area   volume
# 1 Bilinski dodecahedron          4       14    24     12 38.83282 16.94427

zono4 = zonohedron( classics.genlist[['RI']] )
zono7 = zonohedron( classics.genlist[['TO']] )
summary( zono, zono4, zono7 )
#                fullname generators vertices edges facets     area   volume
# 1 Bilinski dodecahedron          4       14    24     12 38.83282 16.94427
# 2   rhombic icosahedron          5       22    40     20 64.72136 42.36068
# 3  truncated octahedron          6       24    36     14 53.56922 32.00000
```

---

grpDuplicated *Grouping by duplicated elements*

Description

`grpDuplicated()` is a generic function that takes an indexed set of "elements", and outputs an integer vector with the same length. The "elements" can be components of a vector, or the row vectors or column vectors of a matrix. In the output vector, a component is 0 if and only if the corresponding element is unique. When the element is unique, it forms a *singleton group*. Output components have equal positive integer values if and only if the corresponding elements are identical to each other. These elements form a *non-singleton group*, and the positive integer is called the *group number*.

The number of singleton groups is equal to #(zeros), which is equal to the #(elements) - #(duplicated elements).
The number of non-singleton groups is equal to max(output vector).
The number of all groups is equal to #(zeros) + max(output vector).

Usage

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

## S3 method for class 'matrix'
grpDuplicated( x, MARGIN=1, ... )
```

Arguments

`x`	a vector or matrix of atomic mode `"numeric"`, `"integer"`, `"logical"`, `"complex"`, `"character"` or `"raw"`.
`MARGIN`	an integer scalar, the matrix margin to be held fixed, as in `apply`. `MARGIN=1` means that it looks for duplicated rows, and `MARGIN=2` means that it looks for duplicated columns. Other values are invalid.
`...`	arguments for particular methods.

Details

The implementation is based on `std::unordered_map` in C++11, which uses a hash-table.

Value

The return value is an integer vector with all elements ranging from 0 to `K`, where `K` is the number of non-singleton groups.
For vector `x` the elements are the vector components, and the output is the same length as the input.
For a matrix `x` with `MARGIN=1`, the elements are the rows of the matrix and the output has length `nrow(x)`.
For a matrix `x` with `MARGIN=2`, the elements are the columns of the matrix and the output has length `ncol(x)`.
The `'ngroups'` attribute of the returned vector is set to an integer 3-vector. The 1st component is the total number of groups, the 2nd component is the number of singleton groups, and the 3rd component is the number of non-singleton groups `K`.

Note

The templated C++ function that does the real work is taken from the package **uniqueAtomMat** by Long Qu, but the returned vector is slightly modified by Glenn Davis.

Author(s)

Long Qu and Glenn Davis

Source

https://github.com/cran/uniqueAtomMat/
The package **uniqueAtomMat** was removed from CRAN by its author Long Qu.

Examples

```
set.seed(0)

#   test a numeric vector
x = rnorm(7)
y = rnorm(5)
grpDuplicated( c(x,y,rev(x)) )
##  [1] 7 6 5 4 3 2 1 0 0 0 0 0 1 2 3 4 5 6 7
##  attr(,"ngroups")
##  [1] 12  5  7

# test a numeric matrix, both rows and columns
A = matrix( rnorm(3*7), 3, 7 )
B = matrix( rnorm(3*5), 3, 5 )

#   the columns of cbind(A,B,A) have the duplicates one would expect
grpDuplicated( cbind(A,B,A), MARGIN=2 )
##  [1] 1 2 3 4 5 6 7 0 0 0 0 0 1 2 3 4 5 6 7
##  attr(,"ngroups")
##  [1] 12  5  7

# but the rows of cbind(A,B,A) are unique
grpDuplicated( cbind(A,B,A), MARGIN=1 )
##  [1] 0 0 0
##  attr(,"ngroups")
##  [1] 3 3 0
```

---

inside *test points for being inside a zonotope*

Description

Test points for being inside a zonotope. The boundary points are considered to be inside.

Usage

```
## S3 method for class 'zonotope'
inside( x, p )
```

Arguments

`x`	a **zonotope** object - a **zonohedron**, **zonogon**, or **zonoseg**
`p`	an NxM numeric matrix, where M is the dimension of the zonotope. The points to be tested are in the rows. `p` can also be a numeric vector that can be converted to such a matrix, by row.

Details

The given zonotope is viewed as the intersection of *slabs*; there is a slab for each hyperplane in the simplified matroid. For each slab a signed distance to boundary of the slab is computed. For points outside the slab the distance is positive, for points on the boundary, the distance is 0, and for points in the interior of the slab the distance is negative. The distance to the zonotope is computed as the maximum over all these slab distances, and the *critical hyperplane* index is recorded. A point is inside iff the zonotope distance ≤ 0.

Value

`inside.zonotope()` returns a `data.frame` with N rows and these columns:

`p`	the given point
`distance`	the signed distance from the point to the zonotope. When `distance` < 0, the point is in the interior, and `distance` is the true Euclidean distance. For boundary points, `distance` is 0. When `distance` > 0, the point is outside, and `distance` may be larger than the true Euclidean distance, so it is really a pseudo-distance.
`inside`	whether the point is inside the zonotope. `inside` = `distance<=0`.
`idxhyper`	the index of the *critical hyperplane* in the simplified matroid. This is the index of the slab where the maximum slab distance was taken. For a **zonoseg** there is only 1 hyperplane (the empty set) so this is always 1.

If the row names of `p` are unique, they are copied to the row names of the output.
In case of error, the function returns `NULL`.

See Also

`inside2trans()`

Examples

```
zono1 = zonoseg( c(1,-2,3,0,-3,-4) )

getsegment(zono1)
# [1] -9 4
```

```
p = c( 0, -3*pi, pi, 2*pi, getsegment(zono1) )

inside( zono1, p )
#           p inside   distance idxhyper
# 1  0.000000   TRUE -4.0000000        1
# 2 -9.424778  FALSE  0.4247780        1
# 3  3.141593   TRUE -0.8584073        1
# 4  6.283185  FALSE  2.2831853        1
# 5 -9.000000   TRUE  0.0000000        1
# 6  4.000000   TRUE  0.0000000        1
```

---

inside2trans *test points against a 2-transition surface*

Description

This function tests points for being inside the 2-transition surface associated with a zonohedron.

Usage

```
inside2trans( x, p )
```

Arguments

x	a **zonohedron** object
p	an Nx3 numeric matrix. The points to be tested are in the rows. p can also be a numeric vector that can be converted to such a matrix, by row.

Details

If the surface has no self-intersections, the the definition of whether a point p is "inside" is fairly straightforward: it is where the linking number of p and the surface is non-zero. In fact, if it is non-zero then it must be +1 or -1. The *linking number* is analogous the *winding number* in 2D, for more discussion see **Note**.

Unfortunately, there is currently no test for whether the surface *has* self-intersections, For a bad surface with self-intersections, the linking number might be any integer. Since there is no such test, we simply use the same non-zero linking number rule always.

The computed linkingnumber is returned so that the user can apply the non-zero rule, or the even-odd rule, as appropriate for their situation. These 2 rules are analogous to the two winding number rules used for polygons in computer graphics, see **Point in polygon**.

The case where a point is *on* the surface (i.e. the distance to the surface is 0) is problematic. The linkingnumber is then undefined, and we currently set inside to be undefined as well. Thus inside should be interpreted as *strictly inside*. However, in some situations, the user may want to consider inside to be TRUE in this problematic case. Or the user may want to consider points that are within a very small epsilon of the surface, where roundoff might have occurred, to have inside=FALSE or inside=NA. So the both the computed linkingnumber and distance are returned so the user can use them to make their own definition of what "inside" means.

Value

`inside2trans()` returns a `data.frame` with N rows and these columns:

`p`	the given point
`distance`	the distance from the point to the surface. This is the true Euclidean distance, and not a "pseudo-distance" as in the case of `inside()`. If the point is on the surface, the `distance` should be 0 up to numerical precision.
`linkingnumber`	the linking number of the point and the surface. If the point is *on* the surface (`distance==0`), the (mathematical) linking number is undefined, and the computed `linkingnumber` is `NA` (integer).
`inside`	whether the point is inside the surface; a logical. This is currently set to `linkingnumber != 0`. If the `linkingnumber` is `NA` (integer), then `inside` is `NA` (logical).
`timecalc`	the time to do the calculations, in seconds

If the row names of `p` are unique, they are copied to the row names of the output.
In case of error, the function returns `NULL`.

Note

The standard definition of the *linking number* of a point and a surface uses intersections with rays, see the vignette The 2-Transition Subcomplex and the 2-Transition Surface for the precise definition. This is fine in theory, but in practice does not handle well the case when the ray intersects the boundary of a parallelogram. So this function uses an integral formula for the degree of a *linking map* that reduces to summing the signed area of a lot of spherical triangles, see **Spivak** p. 75 and **Guillemin and Pollack** p. 188.

References

Guillemin, Victor and Alan Pollack. *Differential Topology*. Prentice-Hall. 1974.

Point in polygon — Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/w/index.php?title=Point_in_polygon&oldid=1139808558`. 2023.

Spivak, Michael. *A Comprehensive Introduction to Differential Geometry*. Volume 1. 3rd edition. Publish or Perish. 1999.

See Also

`inside()`

invertboundary — *invert points on the boundary of a zonohedron*

Description

A *zonohedron* Z is the image of a linear map $[0,1]^n \to Z \subset \boldsymbol{R}^3$, from the n-cube to 3D space. For a point on the boundary of the zonohedron, this function computes a point in the unit cube that maps to it. All coordinates of the point in the cube are 0 or 1, except for two of them. The point is not necessarily unique.

Usage

```
## S3 method for class 'zonohedron'
invertboundary( x, point, tol=5.e-14 )
```

Arguments

`x`	a **zonohedron** object as returned by the constructor `zonohedron()`
`point`	Mx3 matrix with points on the boundary of `x` in the rows. Such a matrix typically is returned by `raytrace.zonohedron()` or `section.zonohedron()`. `point` can also be a numeric vector that can be converted to such a matrix, by row.
`tol`	points that are not within `tol` of the boundary are skipped, see **Details**

Details

Given the boundary point, the function determines the facet that contains it. The `pcube` coordinates of the *base vertex* of this facet are all 0 or 1, and fairly easy to determine. If the facet is a parallelogram, the other two coordinates are fairly easy to determine too. If the facet is a zonogon with `K` generators, with `K>2`, then the unknown `K` coordinates are calculated using `invert.zonogon()`. Because of floating point behaviour, coordinates can be slightly negative or slightly more than 1. After the calculation, they are clamped to [0,1].

Value

`invertboundary.zonohedron()` returns a `data.frame` with M rows and these columns:

`point`	the given boundary point
`distance`	signed distance to the boundary of `x`; for successful inversion its absolute value is $\le$ `tol`
`facetidx`	index of the facet pair that contains the point
`sign`	sign of the facet pair; either +1 or -1
`pcube`	a point in the unit n-cube that maps to the given boundary point; all coordinates of `pcube` are 0 or 1, except for 2 of them.
`transitions`	the number of transitions in `pcube` - a non-negative even integer

If a point `point` cannot be inverted, e.g. because `distance` is too large, the other columns are all `NA`.

If the row names of `point` are unique, they are copied to the row names of the output. The column names of `pcube` are copied from the ground set of the associated matroid.

In case of global error, the function returns `NULL`.

See Also

`zonohedron()`, `section.zonohedron()`, `raytrace.zonohedron()`, `invert.zonogon()`

`lintransform` *linear transformations of zonotopes, and vector matroids*

Description

These functions perform straightforward linear transformations on the generators of a zonotope, and the column vectors of a vector matroid

Usage

```
## S3 method for class 'zonohedron'
lintransform( x, W )
## S3 method for class 'zonogon'
lintransform( x, W )
## S3 method for class 'matroid'
lintransform( x, W )
```

Arguments

`x`	`x` can be a vector matroid object, as returned from the constructor `matroid()` that takes a matrix as input.
`W`	An invertible matrix that matches the rank of `x`. This invertibility is verified. `W` can also be a scalar; it is then replaced by that scalar multiplied by the identity matrix of the appropriate rank.

Value

If `x` is a zonohedron (or zonogon), `lintransform(x)` returns the zonohedron (or zonogon) whose generators are the generators of `x` with the matrix `W` applied on the left side. This function is optimized - it is *not* necessary to transform the generators and start all over again.

If `x` is a vector matroid, `lintransform(x)` returns the matroid whose generators are the generators of `x` with the matrix `W` applied on the left side. If `x` is a matroid, but *not* a vector matroid, it returns the original matroid and prints a warning message.

In case of error, e.g. invalid `W`, the function prints an error message and returns `NULL`.

References

Matroid - Wikipedia. https://en.wikipedia.org/w/index.php?title=Matroid&oldid=1086234057

See Also

`rank()`, `matroid()`

`matroid` *matroid construction*

Description

Construct a matroid from a matrix, or from explicit list of hyperplanes

Usage

```
## S3 method for class 'matrix'
matroid( x, e0=0, e1=1.e-6, e2=1.e-10, ground=NULL, ... )

## S3 method for class 'list'
matroid( x, ground=NULL, ... )
```

Arguments

`x`	`x` can be a numeric matrix with 3, 2, or 1 rows whose columns determine the matroid. The matrix must be either square or "wide", i.e. more columns than rows. The matrix must be *full-rank*, i.e. the rank must be equal to the number of rows, which is then the rank of the constructed matroid. Such a matroid is often called a *column matroid* or *vector matroid*. `x` can also be a list of vectors of positive integers, which are thought of as sets, and are the hyperplanes of the matroid. The hyperplanes are checked that they satisfy the matroid hyperplane axioms. The rank of the constructed matroid is determined automatically, and must be 3, 2, or 1.
`ground`	The *ground set* of the matroid - a vector of positive integers in strictly increasing order. When `x` is a matrix, `length(ground)` must be equal to `ncol(x)`. The point `ground[i]` corresponds to the *i'th* column of `x`. If `ground` is `NULL`, the column names of `x` are converted to such a vector if possible. If this is not possible, `ground` is set to `1:ncol(x)`. When `x` is a list, every set in the list must be a subset of `ground`. If `ground` is `NULL`, it is set to the union of all the sets in `x`. For technical reasons, when the rank is 1, `ground` is required and cannot be `NULL`, see **Details**.

`e0`	threshold, in the L^∞ norm, for a column vector of `x` to be considered 0, and thus that the corresponding point in the matroid is a loop. Since the default is `e0=0`, by default a column vector must be exactly 0 to become a loop.
`e1`	threshold, in a pseudo-angular sense, for column vectors to be multiples of each other, and thus members of a group of multiple (aka parallel) points in the matroid. This tolerance is only used when the rank is 2 or 3.
`e2`	threshold, in a pseudo-angular sense, for the planes spanned by pairs of column vectors to be considered coincident, and thus the columns to be in the same hyperplane of the matroid. This tolerance is used when the rank is 3.
`...`	not used

Details

It was mentioned above that the tolerances `e1` and `e2` are *pseudo-angular*. Specifically, vectors are normalized to the L^2 unit sphere and the distance between them is computed in the L^∞ norm.

Matroids are well-known to have many cryptomorphic definitions, e.g. independent sets, bases, circuits, rank function, closure operator, flats, and hyperplanes. See **Matroid - Wikipedia**. In this package, matroids can only be constructed from hyperplanes, but there are functions `rank()` and `is_independent()` that can be used *after* construction.

Checking that the hyperplanes satisfy the matroid hyperplane axioms is made easier by the fact that all simple matroids of rank 3 or less are *paving matroids*, see **Paving Matroid - Wikipedia**

Rank 1 matroids are extremely simple - the loops form the single hyperplane (possibly empty), and the non-loops form a multiple group. If `ground=NULL` the non-loops are unknown, so this is why `ground` is required when the rank is 1.

Value

`matroid()` returns an object with S3 class **'matroid'**.
In case of error, e.g. invalid `x` or computed hyperplanes, the function prints an error message and returns `NULL`.

Note

The *ground set* of positive integers should not be too sparse; otherwise performance may suffer.

When `x` is a matrix with 3 rows, it may happen that the computed hyperplanes do not satisfy the axioms for a matroid. In that case, the user will be prompted to try reducing tolerance `e2`. Getting the expected hyperplanes may require some *a priori* knowledge of the expected hyperplanes. For best results, the matrix should be given with maximum precision.

References

Matroid - Wikipedia.
`https://en.wikipedia.org/w/index.php?title=Matroid&oldid=1086234057`

Paving Matroid - Wikipedia.
`https://en.wikipedia.org/w/index.php?title=Paving_matroid&oldid=1021966244`

See Also

`rank()`, `simplify()`, `getsimplified()`

`matroid-getters` *matroid get functions*

Description

get some important members of a matrix

Usage

```
## S3 method for class 'matroid'
getground( x )

## S3 method for class 'matroid'
gethyperplane( x )

## S3 method for class 'matroid'
getmatrix( x )

## S3 method for class 'matroid'
getloop( x )

## S3 method for class 'matroid'
getmultiple( x )
```

Arguments

`x`	a matroid object, as returned from the constructor `matroid()`

Value

`getground()` returns an vector of positive integers in strictly increasing order = the ground set of the matroid `x`.

`gethyperplane()` returns a list of vectors of positive integers = the hyperplanes of the matroid.
If `x` is the simplification of an "original matroid", the `"lmdata"` attribute of the returned list is set to the *loop* and *multiple group* data of the "original hyperplanes". These hyperplanes can be recovered using `unsimplify()`.
If `x` was constructed from a matrix, these hyperplanes are sorted in decreasing order by length. The non-trivial hyperplanes come first, followed by the trivial hyperplanes. A hyperplane is *trivial* iff it is independent in the matroid. For a matroid of rank 3, a hyperplane is trivial iff it has 2 points.

`getmatrix()` returns the matrix passed to the `matroid.matrix()` constructor, or `NULL` if the list constructor was used. The column names are labeled with the ground set.

getloop() returns an integer vector with the loops of x. If x is simple, it is the empty vector.

getmultiple() returns a list of integer vectors - the multiple groups of x. If x is simple, it is the empty list.

References

Matroid - Wikipedia. https://en.wikipedia.org/w/index.php?title=Matroid&oldid=1086234057

See Also

rank(), simplify(), unsimplify(), getsimplified(),

Examples

```
# construct a classic matroid with 7 points, but assign an unusual ground set
mat = matroid( classics.genlist[['TRD']], ground=11:17 )

getmatrix( mat )

##       11 12 13 14        15       16       17
##  [1,]  1  1  1  1 1.732051 0.000000 0.000000
##  [2,]  1  1 -1 -1 0.000000 1.732051 0.000000
##  [3,]  1 -1  1 -1 0.000000 0.000000 1.732051
```

---

matroid-props *matroid properties*

Description

get some important boolean properties of a matrix, see **Matroid - Wikipedia** for the definitions.

Usage

```
## S3 method for class 'matroid'
is_simple( x )

## S3 method for class 'matroid'
is_uniform( x )

## S3 method for class 'matroid'
is_paving( x )
```

Arguments

x	a matroid object, as returned from the constructor matroid()

Value

`is_simple()` returns a logical. A matroid is *simple* iff it has no loops and no multiple groups.

`is_uniform()` returns a logical. A matroid is *uniform* iff all the hyperplanes have the same size, which is the rank-1.

`is_paving()` returns a logical. For the definition of *paving* see **Paving Matroid - Wikipedia**. This property is important because the hyperplane axioms are fairly easy to check.

References

Matroid - Wikipedia. https://en.wikipedia.org/w/index.php?title=Matroid&oldid=1086234057

Paving Matroid - Wikipedia. https://en.wikipedia.org/w/index.php?title=Paving_matroid&oldid=1021966244

See Also

`matroid()`

`minkowskisum` *Minkowski sum of Two zonotopes*

Description

A zonotope can be viewed as a Minkowski sum of line segments, with one endpoint at 0. Therefore, the Minkowski sum of two zonotopes (in the same dimension) is also a zonotope.

Usage

```
## S3 method for class 'zonotope'
minkowskisum( zono1, zono2, e0=0, e1=1.e-6, e2=1.e-10, ground=NULL, ... )

## S3 method for class 'zonotope'
zono1 %+% zono2
```

Arguments

`zono1`	a zonotope object - a **zonohedron**, a **zonogon**, or a **zonoseg**
`zono2`	a zonotope object with the same dimension as `zono1`
`e0`	see `zonohedron()`
`e1`	see `zonohedron()`
`e2`	see `zonohedron()`
`ground`	the ground set of the returned zonotope. If `ground` is `NULL`, it is set to the ground set of `zono1` followed by the ground set of `zono2` translated sufficiently to not intersect that of `zono1`.
`...`	not used

Details

After verifying that `zono1` and `zono2` are the same dimension, it takes the 2 matrices, `cbind`s them, and passes the new matrix to the appropriate constructor, along with the other arguments. There are no special optimizations.

Value

`minkowskisum()` returns a zonotope of the same dimension as `zono1` and `zono2`.
`%+%` is a more convenient binary operator that calls `minkowskisum()`, but without the flexibility of the extra arguments.
In case of error, the function returns `NULL`.

References

Zonohedron - Wikipedia.
`https://en.wikipedia.org/wiki/Zonohedron`.

See Also

`zonohedron()`, `zonogon()`, `zonoseg()`

`plot2trans` *plot the 2-transition surface associated with a zonohedron*

Description

The 2-transition surface has the topology of a sphere and is contained in the zonohedron. All the facets are parallelograms. The surface is centrally symmetric, with the same center as the zonohedron. The surface may have self-intersections.

Usage

```
plot2trans(  x, type='ef', ecol='black', econc=FALSE,
                fcol='yellow', falpha=0.5, level=NULL,
                normals=FALSE, both=TRUE, bgcol="gray40", add=FALSE, ... )
```

Arguments

`x`	a **zonohedron** object as returned by the constructor `zonohedron()`
`type`	a character string with what parts to draw. If `type` contains `'e'`, then draw the edges. If `type` contains `'f'`, then draw filled facets. If `type` contains `'p'`, then draw points at the centers of the facets.
`ecol`	The color to use when drawing the edges.
`econc`	If `TRUE` then draw the concave edges in red, and with extra thickness
`fcol`	The color to use when drawing the facets.
`falpha`	The opacity to use when drawing the facets.

`level`	An integer vector which is a subvector of `0:(M-2)`, where `M` is the number of simplified generators. Only the facets and edges at the specified levels are drawn. When `level=NULL` then *all* facets and edges are drawn. This argument does not affect the drawing of points.
`normals`	If `TRUE` then draw the unit facet normals.
`both`	if `FALSE` then draw only one half of the centrally symmetric surface. Otherwise draw both halves (the default).
`bgcol`	the background color
`add`	If `TRUE` then add to the current 3D plot. If there is no current 3D plot, it is an error.
`...`	not used

Details

Facets and regular edges are drawn with `rgl::quads3d()`. Concave edges are drawn with `rgl::segments3d()`. Points are drawn with `rgl::points3d()`.

Value

The function returns `TRUE`; or `FALSE` in case of error.

Note

The package **rgl** is required for 3D plots. A large black point is drawn at 0, a large white point at the "white point", and a 50% gray point at the center. A line from the black point to the white point is also drawn.

See Also

`zonohedron()`, `plothighertrans()`, `plot.zonohedron()`

`plothighertrans` *plot abundant and deficient parallelograms*

Description

The 2-transition surface associated with a zonohedron is a topological sphere and is contained in the zonohedron. The surface is centrally symmetric, with the same center as the zonohedron. The surface may have self-intersections. For this function, the surface is required to be strictly starshaped at the center. For the definition of *strictly starshaped* see the vignette The 2-Transition Subcomplex and the 2-Transition Surface.

The 2-transition surface is a union of parallelograms. Each parallelogram has a unit normal that defines a linear functional.
If a 2-transition parallelogram is in the *interior* of the zonohedron then the functional is not maximized on the parallelogram, and there is a corresponding similar parallelogram on the boundary of the zonohedron where the functional *is* maximized. The first parallelogram (in the surface)

is called *deficient* because the functional is not maximized, and the second parallelogram (in the boundary) is called *abundant* because the number of corresponding transitions across this parallelogram is more than 2.
If the 2-transition parallelogram is on the boundary, then it is called *coincident*. The coincident parallelograms are ignored and not drawn in this function.

Because of this 1-1 correspondence between deficient parallelograms (in the 2-transition surface) and the abundant parallelograms (in the boundary of the zonohedron), the area of these two surfaces are the same.

Usage

```
plothighertrans( x, abalpha=1, defcol='green', defalpha=0, ecol=NA,
                  connections=FALSE, bgcol="gray40", both=TRUE, ...  )
```

Arguments

`x`	a **zonohedron** object as returned by the constructor `zonohedron()`
`abalpha`	The opacity to use when drawing the abundant parallelograms. If `abalpha=0` then they are not drawn.
`defcol`	The color to use when drawing the deficient parallelograms
`defalpha`	The opacity to use when drawing the deficient parallelograms. If `defalpha=0` (the default), then they are not drawn.
`ecol`	The color to use when drawing the edges. If `ecol=NA` (the default), then they are not drawn.
`connections`	If `TRUE` then draw segments between centers of the deficient parallelograms in the 2-transition surface, and centers of the the corresponding abundant parallelograms in the zonohedron boundary
`bgcol`	the background color
`both`	if `FALSE` then draw only one half of the centrally symmetric boundary. Otherwise draw both halves. This affects edges and parallelograms.
`...`	not used

Details

Connections are drawn with `rgl::segments3d()` and `rgl::points3d()` . parallelograms and edges are drawn with `rgl::quads3d()`. Both parallelograms and edges are drawn unlit (`lit=FALSE`). The parallelograms are colored by the number of transitions using the color codes in **Burns**, up to 10 transitions.

A large black point is drawn at 0, a large white point at the "white point", and a 50% gray point at the center. A line from the black point to the white point is also drawn.

Value

The function returns `TRUE` when successful; or `FALSE` in case of error.

WARNING

This function currently only works when the 2-transition surface is starshaped at the center. This excludes many of the classic zonohedra.

Note

The package **rgl** is required for 3D plots.

References

Scott A Burns. **The location of optimal object colors with more than two transitions**. Color Research & Application. Vol. 46. No. 6. pp 1180-1193. 2021.

See Also

`zonohedron()`, `plot.zonohedron()`, `plot2trans()`

---

plotpolygon — *plot the* generator polygon *associated with a pointed zonohedron*

Description

A zonohedron is *pointed* iff there is a vector **n** so the inner product of all the zonohedron generators with **n** is positive. In other terminology, it is *pointed* iff there is an open halfspace that contains all the generators.
When **n** exists, a neighborhood of 0 can be cut by a plane orthogonal to **n** and the intersection is a polygon. Since **n** is not unique, the polygon is only unique up to a 2D projective transformation.

Usage

```
plotpolygon( x, normal=NULL, points=TRUE, labels=TRUE )
```

Arguments

`x`	a **zonohedron** object as returned by the constructor `zonohedron()`. It must be pointed.
`normal`	the vector **n** to use - a non-zero numeric vector of length 3. If it is given, the validity is checked and if invalid it is an error. If it is `NULL`, a few canonical normals are first tested for validity. If they are invalid, then a valid one is computed.
`points`	If `TRUE` then draw the vertices of the polygon
`labels`	If `TRUE` then draw labels, taken from the ground set of the simplified matroid, near the vertices

Details

The selected normal vector **n** is added to the title of the plot.

Value

The function returns `TRUE`; or `FALSE` in case of error.

See Also

`zonohedron()`, `plot.zonohedron()`

`print` *Print Basic Facts about a Matroid*

Description

The function prints a nicely formatted summary of a matroid, including the ground set, the rank, loops, multiple groups, and some boolean properties. It prints the number of hyperplanes, broken down by their size. If it is a vector matroid, and its matrix is not too large, it prints that matrix. If the matroid is not simple, it also prints the simplified matroid.

Usage

```
## S3 method for class 'matroid'
print( x, ... )
```

Arguments

`x`	a `matroid` object as returned by the constructor `matroid()`
`...`	further arguments ignored, but required by the generic `print()`

Value

The function returns `TRUE` or `FALSE`.

See Also

`matroid()`

`print.genlist` *Print Basic Metrics for Each Zonohedron Generated by the Matrices in a* ***genlist*** *object*

Description

An S3 class **genlist** object is organized as a named list of 3xN matrices, when N varies. The `print()` method constructs a zonohedron object from each matrix and then prints some basic metrics about each zonohedron, as a data frame. If the matrix has the `"fullname"` attribute, it is added as a column. The names of the list are copied to the rownames of the data frame.

Usage

```
## S3 method for class 'genlist'
print( x, full=TRUE, ... )
```

Arguments

`x`	a `genlist` object
`full`	if `TRUE`, include `area` and `volume` columns
`...`	not used

Details

`print.genlist()` uses `summary.zonohedron()`.

Value

The function returns `TRUE` or `FALSE`.

See Also

`genlist`, `summary.zonohedron()`

Examples

```
# print zonohedra metrics associated with 3 sets of color matching functions
colorimetry.genlist
#                    fullname generators vertices  edges facets      area     volume
# xyz1931.5nm xyz at 5nm step         81     5100  10146   5048  1582.722   4070.345
# xyz1931.1nm xyz at 1nm step        471   112910 225720 112812 39586.707 509434.149
# lms2000.1nm lms at 1nm step        441   146642 292860 146220 22736.652 181369.085


names(classics.genlist)
#  [1] "C"   "RD"   "BD"   "RI"   "RHD" "RT"   "TO"   "TRD" "TC"   "RE"   "RH"   "TI"   "TSR"

print( classics.genlist, full=FALSE )
```

```
#                               fullname generators vertices edges facets
# C                                 cube          3        8    12      6
# RD                 rhombic dodecahedron          4       14    24     12
# BD                Bilinski dodecahedron          4       14    24     12
# RI                  rhombic icosahedron          5       22    40     20
# RHD         rhombo-hexagonal dodecahedron          5       18    28     12
# RT               rhombic triacontahedron          6       32    60     30
# TO                  truncated octahedron          6       24    36     14
# TRD        truncated rhombic dodecahedron          7       32    48     18
# TC               truncated cuboctahedron          9       48    72     26
# RE              rhombic enneacontahedron         10       92   180     90
# RH            rhombic hectotriadiohedron         12      134   264    132
# TI           truncated icosidodecahedron         15      120   180     62
# TSR truncated small rhombicosidodecahedron         21      240   360    122
```

`rank` *Rank and Independence*

Description

calculate the rank of any subset of a matroid, or determine whether any subset is independent

Usage

```
rank( x, subs )

is_independent( x, subs )
```

Arguments

`x`	a matroid object, as returned from the constructor `matroid()`
`subs`	a list of integer vectors, representing subsets of the ground set of `x`. `subs` can also be an integer vector, which is put into a list of length 1.

Value

`rank(x,subs)` returns an integer vector with the same length as the list `subs`. The i'th value is the rank of the i'th set in `subs`. If a set is not a subset of the ground set of `x`, the value is `NA`, and a warning message is printed.

`is_independent(x,subs)` returns a logical vector with the same length as the list `subs`. The i'th value is the independence of i'th set in `x`. It is equal to `TRUE` iff the rank of the subset is equal to the cardinality of the subset.

For both functions the names are copied from input to output.

References

Matroid - Wikipedia. https://en.wikipedia.org/w/index.php?title=Matroid&oldid=1086234057

See Also

matroid()

Examples

```
# make a matroid with rank 3
mat = matroid( classics.genlist[['RT']] )


# the ground set itself should have rank 3
rank( mat, getground(mat) )
##  [1] 3


# single points should have rank 1  (there are no loops)
rank( mat, as.list(getground(mat)) )
##  [1] 1 1 1 1 1 1

# all hyperplanes should have rank 2
rank( mat, gethyperplane(mat) )
##  [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2


# a point not in the ground set should have rank NA
# and the emtpy set should have rank 0
rank( mat, list(100L,integer(0)) )
##  1 of 1 subsets are not a subset of ground.
##  [1] NA 0
```

`raytrace2trans` *compute the intersection of a ray and the 2-transition surface associated with a zonohedron*

Description

The *open ray* with basepoint $\boldsymbol{b}$ and non-zero direction $\boldsymbol{d}$ is the set of the form $\boldsymbol{b} + t\boldsymbol{d}$ where $t > 0$.

This function computes the intersection of an open ray and the 2-transition surface associated with a zonohedron. The linking number of the surface and $\boldsymbol{b}$ must be ± 1. This is verified at the beginning, and if not true, then it is an error. The linking number condition implies that an intersection exists for every ray based at $\boldsymbol{b}$. Note also that the condition implies that $\boldsymbol{b}$ is not on the surface. For discussion of uniqueness, see **Details**. For the definition of *linking number* see The 2-Transition Subcomplex and the 2-Transition Surface.

The 2-transition surface is a union of parallelograms. The surface is symmetric about the center of the zonhedron, so each parallelogram has an antipodal parallelogram. Each parallelogram is specified by an ordered pair of distinct generators from the *simplified* matroid associated with the zonohedron. Thus, if there are N generators, there are $N(N-1)$ parallelograms. Swapping the generators of a parallelogram changes it to the antipodal parallelogram.

The 2-transition surface has two *poles* - the point 0 and the sum of all the generators. It is OK for the ray to pass through one of these poles.

Usage

```
raytrace2trans( x, base, direction, invert=FALSE, plot=FALSE, tol=1.e-12, ... )
```

Arguments

`x`	a **zonohedron** object as returned by the constructor `zonohedron()`
`base`	a numeric 3-vector - the basepoint of all the rays. The surface must be strictly starshaped at `base`, and this is verified.
`direction`	a numeric Mx3 matrix with M non-zero directions in the rows. The basepoint and these directions define M rays. `direction` can also be a numeric vector that can be converted to such a matrix, by row.
`invert`	if `TRUE`, then compute a point in the unit cube that maps to the point on the 2-transition surface associated with `x`, and add it as a column in the returned `data.frame`
`plot`	if `TRUE`, the computed rays, up to the boundary, are *added* to an existing plot of the zonohedron `x`, see `plot.zonohedron()`. The segments are drawn in the color red. If there is no open 3D plot, a warning is issued.
`tol`	the tolerance for being strictly starshaped, and for intersection with a *pole*.
`...`	not used

Details

The function is designed for the situation when the intersection of the ray and the surface exists and is unique. This is guaranteed for all ray directions $\boldsymbol{d}$ when the surface is strictly starshaped at $\boldsymbol{b}$. This condition is checked at the beginning of the function, and if false then a warning is issued that the intersection point may not be unique. For the definition of *strictly starshaped* see The 2-Transition Subcomplex and the 2-Transition Surface.

For finding a parallelogram of intersection, a brute-force search is used; all parallelograms are searched until the first one that intersects the ray is found. To speed things up, the 3D problem is reduced to 2D, and the search is programmed in plain C.

If `plot` is `TRUE`, the rays are drawn in red using `rgl::segments3d()` and `rgl::points3d()`.

Value

`raytrace2trans()` returns a `data.frame` with M rows and these columns:

`base`	the given basepoint - this is the same in every row
`direction`	the given direction
`gndpair`	the 2 generators of the parallelogram that the ray intersects, taken from the ground set of the simplified matroid. If the ray passes through a pole, both of these are `NA`.

`alpha`	the 2 coordinates of the intersection point within the parallelogram
`tmax`	ray parameter of the intersection with the parallelogram, always positive
`point`	the point on the surface; the intersection of the ray and the parallelogram
`iters`	the number of parallelograms searched, until the desired one was found. If the ray intersects a pole, this is 0.
`timetrace`	the computation time for the given ray, in seconds. This does not include the initial preprocessing time.

And if `invert` is `TRUE`, then this column is added:

`pcube`	a point in the unit cube that maps to `point`. This point in the cube always has 2 transitions.

If `base` and `direction` in a row cannot be processed, the rest of the row is `NA`.

If the row names of `direction` are unique, they are copied to the row names of the output.

In case of error, the function returns `NULL`.

Note

The package **rgl** is required for 3D plotting.

See Also

`zonohedron()`, `plot.zonohedron()`, `section2trans()`, `raytrace.zonohedron()`

`section2trans` *compute the intersection of a plane and the 2-transition surface associated with a zonohedron*

Description

In general, the 2-transition surface may be highly non-convex, possibly with self-intersections. The intersection of a plane and the 2-transition surface is a union of polygons, possibly with self-intersections and intersecting each other. This function computes one of those polygons. If there are other polygons, it issues a warning and does not try to compute them.

Usage

```
section2trans( x, normal, beta, invert=FALSE, plot=FALSE, tol=1.e-12, ... )
```

Arguments

`x`	a **zonohedron** object as returned by the constructor `zonohedron()`
`normal`	a non-zero numeric 3-vector - the normal of all the planes
`beta`	a numeric M-vector of plane constants. The equation of the k'th plane k is: `<x,normal>` = `beta[k]`.
`invert`	if `TRUE`, then compute a point in the unit cube that maps to the point on the 2-transition surface, and add it as a column in the returned `data.frame`
`plot`	if `TRUE`, the polygons formed by the the intersection of the planes and the 2-transition surface. *added* to an existing 3D plot of the zonohedron `x`, see `plot.zonohedron()`. The polygons are drawn in red.
`tol`	a small positive number, used as the tolerance for the plane intersecting the interior of each parallelogram, see **Details**.
`...`	not used

Details

The function is designed for the situation when the intersection of a plane and the surface is a single polygon.

Given a plane, the function finds all the parallelograms of the surface whose interiors intersect the plane. Each intersection is a line segment. For each parallelogram it associates one of the endpoints of the segment. The parallelograms are put in polygon order by picking an arbitrary one as the starting point, and then "marching" from one to the next using the canonical parallelogram adjacency relation. After returning to the starting point, if there are other parallelograms remaining, it means that there are other polygons in the section and a warning is issued.

Value

`section2trans()` returns a list of length M (=`length(beta)`), and the i'th item in the list is a data frame with these columns:

`point`	a Px3 matrix with the P points of the i'th polygon in the rows. If the plane does not intersect the 2-transition surface, then P=0 and the matrix has 0 rows. The row names of `point` are the indexes of the facets that contain the vertices of the polygon; see **Details**.
`gndpair`	the 2 indexes from the ground set that generates the parallelogram containing `point`. See **Details** for a description of the "marching parallelogram" procedure.

And if `invert` is `TRUE`, then this column is added:

`pcube`	a point in the unit cube that maps to `point`. This point in the cube always has 2 transitions.

The names of the returned list are readable strings that contain `normal` and `beta[i]`.

In case of error, the function returns `NULL`.

Note

The package **rgl** is required for 3D plotting.

See Also

zonohedron(), plot.zonohedron(), section.zonohedron(), raytrace2trans()

---

simplify *simplify and unsimplify*

Description

A *simple matroid* has no loops and no multiple groups. Simplification is the process of removing all loops, and every point except one from each multiple group. The result is a simple matroid.
The functions below simplify a matroid, or an explicit list of hyperplanes.
The hyperplanes can be *unsimplified* if the original loops and multiple groups are known.

Usage

```
## S3 method for class 'matroid'
getsimplified( x, ... )

## S3 method for class 'list'
simplify( x, ground=NULL, ... )

## S3 method for class 'list'
unsimplify( x, loop=NULL, multiple=NULL, ground=NULL, ... )
```

Arguments

`x`	`x` can be a matroid object, as returned from the constructor `matroid()` `x` can also be a list of vectors of positive integers, which are thought of as sets. All must be subsets of `ground`. They do not have to satisfy the matroid hyperplane axioms. For a definition of *loop* and *multiple group* in this case, see **Details**.
`ground`	The *ground set* of the sets in `x` (both `simplify()` and `unsimplify()`). It must be a vector of positive integers in strictly increasing order (not verified). If `ground` is `NULL`, it is set to the union of the sets in `x`.
`loop`	a vector of positive integers, the loops, to add to the list `x`; `loop` must be disjoint from `ground` (verified). If `loop` is `NULL`, the function looks for `loop` in the attribute data `attr(x,'lmdata')`. If there is no such attribute, `loop` is set to the empty set.
`multiple`	a list of vectors of positive integers, the multiple groups, to add to the list `x`; these groups must be pairwise disjoint and disjoint from `loop` (not verified). Each group must intersect the ground set in exactly one point (verified). If `multiple` is `NULL`, the function looks for `multiple` in the attribute data `attr(x,'lmdata')`. If there is no such attribute, `multiple` is set to the empty list.
`...`	not used

Details

First consider the case when x is a list of vectors of positive integers. Each vector represents a subset of the ground set. They are not required to satisfy the hyperplane axioms, but by abuse of language we will call them hyperplanes in this paragraph. A *loop* is a point (an integer in the ground set) that is in every hyperplane. Imagine now that all loops have been removed. Say that two points p and q are *multiples* iff for every hyperplane H, $p \in H$ iff $q \in H$. This is an equivalence relation, and the *multiple groups* are the equivalence classes with more than one point. For computation it is convenient to think of a boolean *incidence matrix*. There is a column for each point in the ground set, and a row for each hyperplane. An entry is `TRUE` iff the point is in the hyperplane. A *loop* is then a column of all `TRUE`s. A *multiple group* is a maximal set of duplicate columns. This is basically how `simplify()` is implemented, except with optimizations that avoid computing the very large incidence matrix.

Now consider the case when x is a matroid object. When x was constructed, the simplification of x was computed (with help from the *previous* `simplify()`) and stored as a member of x (unless x was already simple). So in this case `getsimplified(x)` does not do any real work and only takes microseconds.

These functions are accelerated with C/C++.

Value

If x is a matroid, `getsimplified(x)` returns x when x is simple, and a member of x when x is not simple. It does not do any real work.

If x is a list, `simplify(x)` returns a list of the same length, but with all loops removed, and every point except one from each multiple group removed. The integer that remains is the smallest one in the group. The order of the sets is preserved. It also sets the `'lmdata'` attribute of the returned list to a list of 2 objects - the loop and multiple group data found in x.

If x is a list, `unsimplify(x)` returns a list of the same length, but with the loops and multiples added back. The order of the sets is preserved.

In case of error, e.g. invalid x etc., the function prints an error message and returns `NULL`.

References

Matroid - Wikipedia. https://en.wikipedia.org/w/index.php?title=Matroid&oldid=1086234057

See Also

`rank()`, `matroid()`

Examples

```
# an example using simplify.list() and unsimplify.list()
# get the matrix for CIE XYZ at 5 nm step size
mat3x81 = colorimetry.genlist[[1]]
```

```
# create the matroid
mat5 = matroid( mat3x81 )

#  test for simplicity
is_simple(mat5)
##  [1] FALSE

# get the list of hyperplanes, and simplify
hyper = gethyperplane( mat5 )
hypersimple = simplify( hyper )

# print the loop and multiple data found
attr(hypersimple,'lmdata')

# unsimplify and compare to the originals
# the list attr(hypersimple,'lmdata') is 'secretly' used in unsimplify()
identical( unsimplify(hypersimple), hyper )
##  [1] TRUE
```

`spherize` *spherize a zonotope*

Description

The input is a zonotope with a best-fit ellipsoid (or ellipse for a **zonogon**) with axes that may have very different lengths. The function computes a *spherizing matrix* `W`, and then transforms the zonotope so its boundary is close to a sphere.

Usage

```
## S3 method for class 'zonotope'
spherize( x, method='ZCA', ... )
```

Arguments

`x`	a zonotope object - a **zonohedron**, a **zonogon**, or a **zonoseg**.
`method`	for computing the matrix `W`, either `'ZCA'` or `'PCA-COR'`. Matching is partial and case-insensitive.
`...`	not used

Details

The 2 methods are taken from **Kessy, et. al.**.

Value

After computing the matrix `W`, the function return `lintransform(x,W)`. The `"sphering"` attribute is set to `W`.
If `x` is a 1D **zonoseg**, sphering is not really possible, so the function prints a warning message and returns `x`. In case of error, the function returns `NULL`.

References

Agnan Kessy, Alex Lewin, Korbinian Strimmer. **Optimal whitening and decorrelation**. `https://arxiv.org/abs/1512.00809` v4 2016.

See Also

`lintransform()`

`support` *the support function for a zonotope*

Description

Compute the classical support function for a zonotope. It also computes a point on the boundary where the linear functional is maximized, and the dimension of the face where the supporting hyperplane intersects the zonotope.

Usage

```
## S3 method for class 'zonotope'
support( x, direction, tol=5.e-15 )
```

Arguments

`x`	a zonotope object - a **zonohedron**, a **zonogon**, or a **zonoseg**
`direction`	an NxM matrix with N directions in the rows. If `x` is a **zonohedron**, M must be 3. If `x` is a **zonogon**, M must be 2. If `x` is a **zonoseg**, M must be 1. `direction` can also be a vector that can be converted to such a matrix, by row. The direction is normal to the supporting hyperplane
`tol`	the tolerance for determining whether the supporting hyperplane intersects a face with positive dimension. This does not affect the value of the support function. For a **zonoseg**, `tol` is ignored.

Value

The function returns a `data.frame` with N rows and these columns:

`direction`	the given direction
`value`	the value of the support function of `x`, in the given direction
`argmax`	a point on the boundary of `x` where the functional max is taken. This point is the center of the face where the supporting hyperplane intersects the zonotope.
`dimension`	of the face where the supporting hyperplane intersects the zonotope. 0 means a vertex, 1 means an edge, and 2 means a 2-face.

If `direction` is 0, the other columns are `NA`. If the rownames of `direction` are unique, they are copied to the row names of the output.
In case of error, the function returns `NULL`.

References

Wikipedia - Support function
https://en.wikipedia.org/wiki/Support_function

See Also

`zonoseg()`

`symmetrize` *symmetrize a zonotope*

Description

The input is a zonotope whose matroid is simple. The function adds new generators that creates a new zonotope that is a translate of the original, and has center of symmetry at 0.

Usage

```
## S3 method for class 'zonotope'
symmetrize( x, e0=0, e1=1.e-6, e2=1.e-10, ... )
```

Arguments

`x`	a zonotope object - a **zonohedron**, a **zonogon**, or a **zonoseg**. The matroid of this zonotope must be simple.
`e0`	see `zonohedron()`
`e1`	see `zonohedron()`
`e2`	see `zonohedron()`
`...`	not used

Details

Each generator g (a column of the matrix) is replace by 2 generators: `g/2` and `-g/2`. The new set of generators correponds to a *star* at 0, from Sec 2-8 of **Coxeter**.
The new ground points are obtained by translating the original ground points by the their maximum.

Value

The function returns a zonotope that is a translate of the original, and has center of symmetry at 0. In case of error, the function returns `NULL`.

References

Coxeter, H.S.M. **Regular Polytopes**. Dover Publications. 1973.

See Also

`zonohedron()`, `zonogon()`, `zonoseg()`

`transitionsdf` *summarize the number of transitions and associated data, over all parallelograms in the boundary of a zonohedron*

Description

The 2-transition surface is a union of parallelograms. For this function, the surface is required to be strictly starshaped at the center. For the definition of *strictly starshaped* see The 2-Transition Subcomplex and the 2-Transition Surface.

Each parallelogram has a unit normal that defines a linear functional.
If the 2-transition parallelogram is in the *interior* of the zonohedron then the functional is not maximized on the parallelogram, and there is a corresponding similar parallelogram on the boundary of the zonohedron where the functional *is* maximized. The first parallelogram (in the surface) is called *deficient* because the functional is not maximized, and the second parallelogram (in the boundary) is called *abundant* because the number of corresponding transitions across this parallelogram is more than 2. The difference between the functional values is called the *deficit*.
If the 2-transition parallelogram is on the boundary, then it is called *coincident*. It is also called *non-deficient* and the deficit is 0.

Usage

```
transitionsdf( x, trans2=TRUE )
```

Arguments

`x`	a **zonohedron** object as returned by the constructor `zonohedron()`. The 2-transition surface must be strictly starshaped.
`trans2`	if `TRUE`, then include metrics on the non-deficient (coincident) parallelograms, with 2 transitions. This is always the first row of the returned data frame. if `FALSE`, then data on the non-deficient parallelograms is not included, and the returned data frame only has data on the deficient parallelograms, with more than 2 transitions.

Value

`transitionsdf()` returns a `data.frame` with a row for each number of transitions found, plus a final row with totals on appropriate columns. The columns are:

`transitions`	the number of transitions, a positive even integer, in increasing order.
`parallelograms`	the number of parallelograms with the given number of transitions
`area`	the min and max of the area of the parallelograms with the given number of transitions
`area.sum`	the total area of the parallelograms with the given number of transitions

`deficit`	the min and max of the deficit of the parallelograms with the given number of transitions. When there are 2 transitions the deficit should be exactly 0, but is usually slightly non-0 due to truncation. When there are more than 2 transitions the deficit is positive.
`example`	the 2 generators (from the ground set of the simplified matroid) of the parallelogram with the maximum area

In case of error, the function returns `NULL`.

Note

Because of the 1-1 correspondence between similar parallelograms, the surface areas of the 2-transition surface and the boundary of the zonohedron are equal.

See Also

`zonohedron()`, `plot.zonohedron()`

`zonogon` *zonogon construction*

Description

Construct a zonogon from a numeric matrix with 2 rows.

Usage

```
zonogon( mat, e0=0, e1=1.e-6, ground=NULL )

polarzonogon( n, m=n, ground=NULL )
```

Arguments

`mat`	a numeric 2xM matrix, where $2 \leq$ M. The matrix must have rank 2 (verified). The M columns are the generators of the zonogon.
`e0`	threshold for a column of `mat` to be considered 0, in the L^∞ norm. Since the default is `e0=0`, by default a column must be exactly 0 to be considered 0.
`e1`	threshold, in a pseudo-angular sense, for non-zero column vectors to be multiples of each other, and thus members of a group of multiple (aka parallel) points in the associated matroid. It OK for a column to be a negative multiple of another.
`ground`	The *ground set* of the associated matroid of rank 2 - an integer vector in strictly increasing order, or `NULL`. When `ground` is `NULL`, it is set to `1:ncol(mat)`. If `ground` is not `NULL`, `length(ground)` must be equal to `ncol(mat)`. The point `ground[i]` corresponds to the *i'th* column of `mat`.

n	an integer ≥ 3. The generators are computed as n equally spaced points on the unit circle, starting at (1,0).
m	an integer with $2 \leq$ m $\leq$ n. When m < n, only the first m points are used as generators of the zonogon.

Details

polarzonogon() is useful for testing. The term *polar zonogon* is my own, and based on the *polar zonohedron* in *Chilton & Coxeter*. It it loads the matrix mat and passes it to zonogon(). When m=n the zonogon is a regular 2n-gon. When m<n the zonogon is a has 2m vertices, but is not necessarily regular. The generators correspond to the n'th-roots of unity.

Value

zonogon() and polarzonogon() return a list with S3 class 'zonogon'. In case of error, e.g. invalid mat, the functions print an error message and returns NULL.

Note

The *ground set* of positive integers should not be too sparse; otherwise performance may suffer.

References

B. L. Chilton and H. S. M. Coxeter. **Polar Zonohedra**. The American Mathematical Monthly. Vol 70. No. 9. pp. 946-951. 1963.

See Also

zonohedron(), zonoseg(),

---

zonogon-getmetrics *get important metrics about a zonogon, and print basic facts about a zonogon.*

Description

Get some important zonogon metrics; some computation is used. Also print the data, and more.

Usage

```
## S3 method for class 'zonogon'
getmetrics( x )

## S3 method for class 'zonogon'
print( x, ... )
```

Arguments

`x`	a **zonogon** object
`...`	not used

Details

`print.zonogon()` prints some basic information about the zonogon, and the associated matroid.

Value

`getmetrics()` returns a list with these items:

`vertices`	the number of vertices
`perimeter`	the sum of the lengths of the all edges
`area`	as a polygon

All of these are always positive.

`print.zonogon()` returns `TRUE` or `FALSE`.

See Also

`zonogon()`, `getmetrics.zonohedron()`

`zonogon-invert` *invert points in a zonogon*

Description

A *zonogon* Z is the image of a linear map $[0,1]^n \to Z \subset \boldsymbol{R}^2$, from the n-cube to the plane. For a point in a zonogon, this function finds a point in the unit cube that maps to it. There are infinitely many such points in general (unless $n = 2$), but this function picks a specific point using the *standard tiling*, see **Details**.

Usage

```
## S3 method for class 'zonogon'
invert( x, z, tol=0, plot=FALSE, ... )
```

Arguments

`x`	a **zonogon** object as returned by the constructor `zonogon()`
`z`	a numeric Mx2 matrix, with M points in the rows. `z` can also be a numeric vector that can be converted to such a matrix, by row.
`tol`	points that are within `tol` of the boundary are still processed, see **Details**
`plot`	if `TRUE` then the points in `z` are *added* to an existing plot of the zonogon `x`, using a red X symbol, see `plot.zonogon()`. If there is no plot open, a warning is issued.
`...`	not used

Details

The given points are first tested for being inside the zonogon, using inside() and the given tol. If any are outside, a warning is issued. When the corresponding point pcube is computed, it is clamped to the unit cube, so the inversion error may be as large as tol.

Inversion is not unique in general. For this function, the *standard tiling* of the zonogon by parallelograms is computed; it is an example of a *zonotopal tiling*. It is a *regular zonotopal tiling* because it arises from the projection of a zonohedron onto the plane, see *Ziegler*. The function plot.zonogon() has an option to plot this tiling. Given the point z, the function determines a parallelogram that contains the point. The pcube coordinates of the *base* of this parallelogram are all 0 or 1, and the coordinates of z *within* the parallelogram are in [0,1]. Thus, all coordinates of pcube are 0 or 1, except possibly for 2 of them.

Value

invert.zonogon() returns a data.frame with M rows and these columns:

z	the given point
pcube	a point in the unit cube that maps to z. Every pcube has all coordinates 0 or 1, except possibly for the 2 given by hyper, see **Details**.
hyper	the 2 indexes of the generators of the parallelogram that contains z, in the simplified matroid. These 2 coordinates in pcube are not 0 or 1 in general.
hyperidx	the index of the parallelogram that contains z

If a point z cannot be inverted, the other columns are all NA, and a warning message is printed.

If the row names of z are unique, they are copied to the row names of the output. The column names of pcube are copied from the ground set of the associated matroid.

In case of error, the function returns NULL.

References

Ziegler, G.M. **Lectures on Polytopes**. Graduate Texts in Mathematics. Springer New York. 2007.

See Also

zonogon(), inside(), plot.zonogon()

Examples

```
#   make a zonogon with 5 generators
pz20 = polarzonogon( 20, 5 )

#   make 7 random points in the zonogon
set.seed(0)
pcube = matrix( runif(5*7), 5, 7 )
z = t( getmatrix(pz20) %*% pcube )

#  invert these 7 points back to the cube
```

```
invert( pz20, z )
#         z.1       z.2    pcube.1    pcube.2    pcube.3    pcube.4    pcube.5
# 1 2.0676319 1.6279807 0.00000000 0.70030526 1.00000000 1.00000000 0.01553241
# 2 2.4031738 1.9658035 0.00000000 0.96572153 1.00000000 1.00000000 0.28450140
# 3 0.9230336 1.0885446 0.00000000 0.00000000 0.39548689 1.00000000 0.04948838
# 4 2.5242122 1.7395069 0.16540765 1.00000000 1.00000000 1.00000000 0.03542132
# 5 2.2598725 1.0601592 0.38111324 1.00000000 1.00000000 0.20192029 0.00000000
# 6 1.1387813 1.2636700 0.00000000 0.00000000 0.65250505 1.00000000 0.07478012
# 7 1.6315341 1.0777737 0.00000000 0.64210923 1.00000000 0.36039509 0.00000000

#   hyper.1 hyper.2 hyperidx
# 1       2       5        7
# 2       2       5        7
# 3       3       5        9
# 4       1       5        4
# 5       1       4        3
# 6       3       5        9
# 7       2       4        6
```

zonogon-plot *plot a zonogon*

Description

Plot a **zonogon** object, with many options.

Usage

```
## S3 method for class 'zonogon'
plot( x, orientation=TRUE, normals=FALSE, elabels=FALSE,
                              tiling=FALSE,  tlabels=FALSE,
                              trans2=FALSE, trans2type='both', ... )
```

Arguments

`x`	a **zonogon** object as returned by the constructor `zonogon()`
`orientation`	if `TRUE` then draw the edges with orientation arrows. Otherwise just draw unoriented line segments.
`normals`	if `TRUE` then draw an outward-pointing unit normal on each edge
`elabels`	if `TRUE` then label each edge with its generator
`tiling`	if `TRUE` then draw the standard tiling of the zonogon by parallelograms
`tlabels`	if `TRUE` then label each parallelogram in the tiling with its generators. If `tiling` is `FALSE` then this is ignored.
`trans2`	if `TRUE` then draw the image of the 2-transition subcomplex of the unit cube $[0,1]^n$, in the color blue. `trans2` can also be an integer 2-vector defining a range of levels of the subcomplex, where the *level* of a vertex of the n-cube is the number of 1s. Both integers should be between 0 and n.

`trans2type`	which part of the 2-transition subcomplex to draw. It can be `'BP'` for *bandpass* (aka Type 1), `'BS'` for *bandstop* (aka Type 2), or `'both'` for both.
`...`	not used

Details

A white dot is plotted at the center of the zonogon. A suitable is title is added above the plot. If the zonogon was returned from `spherize.zonotope()` the string `"[spherized]"` is added to the title.

Value

The function returns `TRUE`; or `FALSE` in case of error.

See Also

`zonogon()`, `spherize.zonotope()`

`zonogon-raytrace` *compute the intersection of a ray, based in the interior of a zonogon, and the boundary of that zonogon*

Description

The *open ray* with basepoint $\boldsymbol{b}$ and non-zero direction $\boldsymbol{d}$ is the set of the form $\boldsymbol{b} + t\boldsymbol{d}$ where $t > 0$.

This function computes the intersection of an open ray and the boundary of a zonogon Z. The basepoint is normally required to be in the interior of Z, but an exception is made if the basepoint is 0, and on the boundary of Z, and the direction points into the interior of Z. In these two cases the intersection of the open ray and the boundary of Z is unique. In the second case, the basepoint is also allowed to be the sum of all the generators - the so-called *white point* of Z.

Usage

```
## S3 method for class 'zonogon'
raytrace( x, base, direction, plot=FALSE, ... )
```

Arguments

`x`	a **zonogon** object as returned by the constructor `zonogon()`
`base`	a numeric 2-vector - the basepoint of all the rays. `base` must either be in the interior of `x`, or 0 or the *white point* and on the boundary of `x`.
`direction`	a numeric Mx2 matrix with M non-zero directions in the rows. The basepoint and these directions define M rays. `direction` can also be a numeric vector that can be converted to such a matrix, by row.
`plot`	if `TRUE`, the computed rays, up to the boundary, are *added* to an existing plot of the zonogon `x`, see `plot.zonogon()`. The segments are drawn in the color red. If there is no open plot, a warning is issued.
`...`	not used

Value

`raytrace.zonogon()` returns a `data.frame` with M rows and these columns:

`base`	the given basepoint - this is the same in every row
`direction`	the given direction
`facetidx`	the index of the facet (an edge) where ray exits the zonogon
`sign`	of the facet, either +1 or -1
`tmax`	ray parameter of the intersection with the exit facet, always positive
`point`	the point on the boundary; the intersection of the ray and the facet
`timetrace`	the computation time, in seconds

If `base` and `direction` in a row cannot be processed, the rest of the row is `NA`.

If the row names of `direction` are unique, they are copied to the row names of the output.

In case of error, the function returns `NULL`.

See Also

`zonogon()`, `plot.zonogon()`, `section.zonohedron()`

Examples

```
#   make a zonogon with 5 generators
pz20 = polarzonogon( 20, 5 )

#   make 4 random directions
set.seed(0)
dir = matrix(rnorm(4*2),4,2)

#  use basepoint in the interior of the zonogon
raytrace( pz20, c(0.5,0.5), dir )
#  base.1 base.2 direction.1 direction.2 facetidx sign     tmax boundary.1 boundary.2   timetrace
# 1   0.5    0.5   1.2629543   0.4146414        4   -1 2.0503073 3.08944438 1.35014236 7.680000e-05
# 2   0.5    0.5  -0.3262334  -1.5399500        1   -1 0.3246859 0.39407664 0.00000000 4.649995e-05
# 3   0.5    0.5   1.3297993  -0.9285670        2   -1 0.4868719 1.14744192 0.04790678 4.310103e-05
# 4   0.5    0.5   1.2724293  -0.2947204        2   -1 0.9354693 1.69031851 0.22429808 4.149997e-05


#  use basepoint at 0 - on the boundary of the zonogon
raytrace( pz20, c(0,0), dir )
#  base.1 base.2 direction.1 direction.2 facetidx sign    tmax boundary.1 boundary.2 timetrace
# 1     0     0   1.2629543   0.4146414       4   -1 2.192481  2.7690037  0.9090936 0.0001216
# 2      0      0  -0.3262334  -1.5399500       NA   NA       NA         NA         NA        NA
# 3      0      0   1.3297993  -0.9285670       NA   NA       NA         NA         NA        NA
# 4      0      0   1.2724293  -0.2947204       NA   NA       NA         NA         NA        NA
```

`zonogon-section` *compute the intersection of a line and the boundary of a zonogon*

Description

Generically, a line intersects the boundary of a zonogon in 2 points. Computing those 2 points is the chief goal of this function.
For a supporting line, the intersection is a face of the zonogon, but in this function only one point of intersection is computed and returned.

Usage

```
## S3 method for class 'zonogon'
section( x, normal, beta, tol=1.e-10, plot=FALSE, ... )
```

Arguments

`x`	a **zonogon** object as returned by the constructor `zonogon()`
`normal`	a non-zero numeric 2-vector - the normal of all the lines
`beta`	a numeric M-vector of line-constants. The equation of the k'th line k is: `<x,normal> = beta[k]`.
`tol`	a small positive number, used as the tolerance for the line being considered a supporting line
`plot`	if `TRUE`, the line segments formed by the the intersection of the lines and the zonogon are *added* to an existing plot of the zonogon `x`, see `plot.zonogon()`. The segments are drawn in dashed linestyle and the color red. `boundary1` and `boundary2` are plotted as points. If there is no open plot, a warning is issued.
`...`	not used

Value

`section.zonogon()` returns a `data.frame` with M rows and these columns:

`normal`	the given normal vector - this is the same in every row
`beta`	the given line constant
`boundary1`	the 1st intersection point - a 2-vector
`boundary2`	the 2nd intersection point - a 2-vector

Regarding orientation, if `normal` is considered "north" then `boundary1` is on the "west" and `boundary2` is on the "east".

If a line is a supporting line of the zonogon, then `boundary1` is some point in the boundary face (vertex or edge), and `boundary2` is `NA`. If a line does not intersect the zonogon, both `boundary1` and `boundary2` are `NA`.

If the names of `beta` are unique, they are copied to the row names of the output.

In case of error, the function returns `NULL`.

See Also

zonogon(), plot.zonogon(), section.zonohedron()

Examples

```
#   make a zonogon with 5 generators
pz20 = polarzonogon( 20, 5 )

section( pz20, normal=c(1,1), beta=-1:5 )
#   normal.1 normal.2 beta   boundary1.1   boundary1.2 boundary2.1 boundary2.2
# 1        1        1   -1            NA            NA          NA          NA
# 2        1        1    0 -2.220446e-16  0.000000e+00          NA          NA
# 3        1        1    1  2.452373e-01  7.547627e-01   1.0000000   0.0000000
# 4        1        1    2  6.203838e-01  1.379616e+00   1.7547627   0.2452373
# 5        1        1    3  1.095537e+00  1.904463e+00   2.3796162   0.6203838
# 6        1        1    4  1.674729e+00  2.325271e+00   2.9044629   1.0955371
# 7        1        1    5  2.420068e+00  2.579932e+00   3.3252706   1.6747294
```

---

zonohedron *zonohedron construction*

Description

Construct a zonohedron from a numeric matrix with 3 rows. Also construct some special zonohedra useful for testing.

Usage

```
zonohedron( mat, e0=0, e1=1.e-6, e2=1.e-10, ground=NULL )

polarzonohedron( n, m=n, height=pi, ground=NULL )

regularprism( n, m=n, axis=c(0,0,1), ground=NULL )
```

Arguments

mat	a numeric 3xM matrix, where $3 \leq M$. The matrix must have rank 3 (verified). The M columns are the generators of the zonohedron.
e0	threshold for a column of mat to be considered 0, in the L^∞ norm. Since the default is e0=0, by default a column must be exactly 0 to be considered 0.
e1	threshold, in a pseudo-angular sense, for non-zero column vectors to be multiples of each other, and thus members of a group of multiple (aka parallel) points in the associated matroid. It OK for a column to be a negative multiple of another.
e2	threshold, in a pseudo-angular sense, for the planes spanned by pairs of column vectors to be considered coincident, and thus the columns to be in the same hyperplane of the associated matroid.

`ground`	The *ground set* of the associated matroid of rank 3 - an integer vector in strictly increasing order, or `NULL`. When `ground` is `NULL`, it is set to `1:ncol(mat)`. If `ground` is not `NULL`, `length(ground)` must be equal to `ncol(mat)`. The point `ground[i]` corresponds to the *i'th* column of `mat`.
`n`	an integer ≥ 3. The generators are computed as `n` equally spaced points on a circle. See **Details** for more on this circle.
`m`	an integer with $2 \leq$ `m` $\leq$ `n`. When `m` < `n`, only the first `m` points are used as generators of the zonohedron.
`height`	the z value at the apex of the zonohedron, which is the sum of all the generators. The z value of all the generators is set to make this happen. When `height=pi`, as $n \to \infty$ the zonohedron converges to the interior of the surface of revolution of the curve $x = sin(z)$ for $z \in [0, \pi]$, see **Chilton & Coxeter**.
`axis`	the axis of the regular prism. It must be a 3-vector with z value non-zero.

Details

In `zonohedron()`, the contruction of the *zones* (or *belts*) is optimized by following the procedure in *Heckbert*. The key step is sorting face normals that all lie on a great circle of the unit sphere.

For `polarzonohedron()` the circle is centered at (0,0,`height/n`) and parallel to the xy-plane. The radius is `height/n`.

For `regularprism()` the circle is the unit circle in the xy-plane. The 3-vector `axis` is added as column `m+1` of the matrix. The returned zonohedron is the Minkowski sum of a zonogon and the line segment defined by `axis`. If `m` < `n`, the zonogon may not be regular.

Both of these functions are useful for testing. They load the matrix `mat` and pass it to `zonohedron()`.

Value

`zonohedron()` and `polarzonohedron()` return a list with S3 class `'zonohedron'`. In case of error, e.g. invalid `mat`, the functions print an error message and returns `NULL`.

Note

The *ground set* of positive integers should not be too sparse; otherwise performance may suffer.

References

B. L. Chilton and H. S. M. Coxeter. **Polar Zonohedra**. The American Mathematical Monthly. Vol 70. No. 9. pp. 946-951. 1963.

Paul Heckbert. **An Efficient Algorithm for Generating Zonohedra**. 3-D Technical Memo 11. 24 February 1985. Computer Graphics Lab. New York Institute of Technology

See Also

`zonohedron()`, `zonoseg()`,

`zonohedron-plot` *plot a zonohedron*

Description

Plot a **zonohedron** object in 3D, with many options.

Usage

```
## S3 method for class 'zonohedron'
plot( x, type='e', pcol=NULL, ecol=NULL, ewd=3, etcol=NA,
        fcol=NULL, falpha=1, normals=FALSE, bgcol="gray40", both=TRUE, ... )
```

Arguments

`x`	a **zonohedron** object as returned by the constructor `zonohedron()`
`type`	a string of letter with what parts to draw. If `type` contains an `'p'`, then draw a point at the center of each facet. If `type` contains an `'e'`, then draw the edges. If `type` contains an `'f'`, then draw filled facets.
`pcol`	The color to use when drawing points. It can be a vector of 2 colors, and then when `both` is `TRUE`, the first color is used for one half, and the second color is used for the antipodal half. When `pcol` is `NULL`, it is set to `c('black','red')`.
`ecol`	A vector of colors to use when drawing the edges. Let N be the number of *simplified* generators of the zonohedron. Each edge is parallel to exactly one of the generators, so this divides the edges into N *zones*, or *belts*. `ecol` can be a vector of N colors, one for each zone. If `ecol` is shorter than N, it is extended to length N using the last color. If `ecol` is longer than N, the extra colors are ignored. If `ecol` is `NULL`, it is set to `rainbow(N)`.
`ewd`	width of the edges, in pixels
`etcol`	color of the *tiling edges*, for the standard tiling of the facets by parallelograms. This only applies to facets that are *not* parallelograms. The default `etol=NA` means do not draw these edges.
`fcol`	A vector of colors to use when drawing the facets. The 1st color is used for parallelograms, the next color for hexagons, etc. For facets with more edges than colors available, the last color is used. If `fcol` is `NULL`, it is set to `c( 'blue', 'red', 'yellow', 'green', 'orange', 'purple' )`.
`falpha`	opacity of the facets
`normals`	if `TRUE` then draw an outward-pointing unit normal from each facet
`bgcol`	the background color
`both`	if `FALSE` then draw only one half of the centrally symmetric boundary. Otherwise draw both halves. This affects points, edges, and facets.
`...`	not used

Details

Points are drawn with `rgl::points3d()`. Edges are drawn with `rgl::segments3d()`. Edges of the tiles are drawn with `rgl::quads3d()`. Facets are drawn with `rgl::quads3d()`; facets with more than 4 edges are split into trapezoids. Facet normals are drawn with `rgl::arrow3d()`.

Value

The function returns `TRUE`; or `FALSE` in case of error.

Note

The package **rgl** is required for 3D plots. A large black point is drawn at 0, a 50% gray point at the center, and a large white point at the "white point" (which is 2*center).

A line from the black point to the white point is also drawn.

See Also

`zonohedron()`, `spherize.zonotope()`

`zonohedron-raytrace` *compute the intersection of a ray, based in the interior of a zonohedron, and the boundary of that zonohedron*

Description

The *open ray* with basepoint $\boldsymbol{b}$ and non-zero direction $\boldsymbol{d}$ is the set of the form $\boldsymbol{b} + t\boldsymbol{d}$ where $t > 0$.

This function computes the intersection of an open ray and the boundary of a zonohedron Z. The basepoint is normally required to be in the interior of Z, but an exception is made if the basepoint is 0, and on the boundary of Z, and the direction points into the interior of Z. In these two cases the intersection of the open ray and the boundary of Z is unique. In the second case, the basepoint is also allowed to be the sum of all the generators - the so-called *white point* of Z.

Usage

```
## S3 method for class 'zonohedron'
raytrace( x, base, direction, invert=FALSE, plot=FALSE, ... )
```

Arguments

`x`	a **zonohedron** object as returned by the constructor `zonohedron()`
`base`	a numeric 3-vector - the basepoint of all the rays. `base` must either be in the interior of `x`, or 0 or the *white point* and on the boundary of `x`.
`direction`	a numeric Mx3 matrix with M non-zero directions in the rows. The basepoint and these directions define M rays. `direction` can also be a numeric vector that can be converted to such a matrix, by row.

`invert`	if `TRUE`, then compute a point in the unit cube that maps to the point on the boundary of `x` and add it as a column in the returned `data.frame`
`plot`	if `TRUE`, the computed rays, up to the boundary, are *added* to an existing plot of the zonohedron `x`, see `plot.zonohedron()`. The segments are drawn in the color red. If there is no open plot, a warning is issued.
`...`	not used

Details

If `plot` is `TRUE`, the rays are drawn with `rgl::segments3d()`.

Value

`raytrace.zonohedron()` returns a `data.frame` with M rows and these columns:

`base`	the given basepoint - this is the same in every row
`direction`	the given direction
`facetidx`	the index of the facet (a zonogon) where ray exits the zonohedron
`sign`	of the facet, either +1 or -1
`tmax`	ray parameter of the intersection with the exit facet, always positive
`point`	the point on the boundary; the intersection of the ray and the facet
`timetrace`	the computation time, in seconds

And if `invert` is `TRUE`, then these columns are added:

`distance`	signed distance to the boundary of `x`
`pcube`	a point in the unit cube that maps to `boundary`
`transitions`	the number of transitions in `pcube` - a non-negative even integer

If `base` and `direction` in a row cannot be processed, the rest of the row is `NA`.

If the row names of `direction` are unique, they are copied to the row names of the output.

In case of error, the function returns `NULL`.

Note

The package **rgl** is required for 3D plotting.

See Also

`zonohedron()`, `plot.zonohedron()`, `section.zonohedron()`, `invertboundary()`, `raytrace.zonogon()`

Examples

```
#   make a regular prism, a regular 20-gon extruded 1 unit along z-axis
rp10 = regularprism( 10 )

#   make 7 random directions
set.seed(0)
dir = matrix(rnorm(7*3),7,3)

#  use basepoint in the interior of the zonohedron
raytrace( rp10, c(0.5,0.5,0.5), dir )
#   base.1 base.2 base.3  direction.1  direction.2  direction.3 facetidx sign     tmax  ...
# 1    0.5    0.5    0.5  1.262954285 -0.294720447 -0.299215118        1    1 1.6710386  ...
# 2    0.5    0.5    0.5 -0.326233361 -0.005767173 -0.411510833        1    1 1.2150348  ...
# 3    0.5    0.5    0.5  1.329799263  2.404653389  0.252223448        6   -1 0.8724774  ...
# 4    0.5    0.5    0.5  1.272429321  0.763593461 -0.891921127        1    1 0.5605877  ...
# 5    0.5    0.5    0.5  0.414641434 -0.799009249  0.435683299        1   -1 1.1476226  ...
# 6    0.5    0.5    0.5 -1.539950042 -1.147657009 -1.237538422        1    1 0.4040279  ...
# 7    0.5    0.5    0.5 -0.928567035 -0.289461574 -0.224267885        1    1 2.2294766  ...

#  use basepoint 0 on the boundary of the zonohedron
#  note that only 2 directions point into the interior
raytrace( rp10, c(0,0,0), dir )
#   base.1 base.2 base.3  direction.1  direction.2  direction.3 facetidx sign     tmax  ...
# 1      0      0      0  1.262954285 -0.294720447 -0.299215118       NA   NA       NA  ...
# 2      0      0      0 -0.326233361 -0.005767173 -0.411510833       NA   NA       NA  ...
# 3      0      0      0  1.329799263  2.404653389  0.252223448        6   -1 1.128580  ...
# 4      0      0      0  1.272429321  0.763593461 -0.891921127       NA   NA       NA  ...
# 5      0      0      0  0.414641434 -0.799009249  0.435683299        1   -1 2.295245  ...
# 6      0      0      0 -1.539950042 -1.147657009 -1.237538422       NA   NA       NA  ...
# 7      0      0      0 -0.928567035 -0.289461574 -0.224267885       NA   NA       NA  ...
```

`zonohedron-section` *compute the intersection of a plane and the boundary of a zonohedron*

Description

Generically, a plane intersects the boundary of a zonohedron in a convex polygon. Computing that polygon is the chief goal of this function.
For a supporting plane, the intersection is a face of the zonohedron, but in this function only one point of intersection is computed and returned.

Usage

```
## S3 method for class 'zonohedron'
section( x, normal, beta, tol=1.e-10, plot=FALSE, ... )
```

Arguments

`x`	a **zonohedron** object as returned by the constructor `zonohedron()`
`normal`	a non-zero numeric 3-vector - the normal of all the planes
`beta`	a numeric M-vector of line-constants. The equation of the k'th plane k is: `<x,normal>` = `beta[k]`.
`tol`	a small positive number, used as the tolerance for the plane being considered a supporting plane
`plot`	if `TRUE`, the polygons formed by the the intersection of the planes and the boundary of the zonohedron are *added* to an existing 3D plot of the zonohedron `x`, see `plot.zonohedron()`. The polygons are drawn in red.
`...`	not used

Details

Given a plane, the function finds all the facets of the zonohedron that intersect the plane. For each such facet it computes a single point of intersection on the boundary of the facet. For the parallelograms, the computation is done in a C function; and for zonogon facets with 3 or more generators, the computation is done in `section.zonogon()`. Orientation is handled carefully so that no point appears twice. The facets are not processed in order around the boundary, so these points are in no particular order. They are put in polygon order by sorting them by angle around a suitable "diameter" of the zonohedron.

Value

`section.zonohedron()` returns a list of length M (=`length(beta)`), and the i'th item in the list is a data frame with these columns:

`point`	a Px3 matrix with the P points of the i'th polygon in the rows. If the plane does not intersect the zonohedron, then P=0 and the matrix has 0 rows. If the plane is a supporting plane, the polygon is degenerate and P=1 and the matrix has 1 row. The row names of `section` are the indexes of the facets that contain the vertices of the polygon; see **Details**.
`hyperidx`	index of a hyperplane that contains the given point
`sign`	The sign specifying which of the 2 facets (selected or antipodal) contains the given point. The value is +1 or -1.

The names of the list are readable strings that contain `normal` and `beta[i]`.

In case of error, the function returns `NULL`.

Note

The package **rgl** is required for 3D plotting.

See Also

`zonohedron()`, `plot.zonohedron()`, `section.zonogon()`

`zonoseg` *zonoseg construction, querying, and printing*

Description

Construct a zonoseg from a numeric matrix with one row.

A *zonoseg* ("zonotope" + "segment") is my own personal term for a 1-dimensional zonotope. I could not find an alternative term. It is a linear image of the unit cube $[0,1]^n$ in the real numbers, and a compact segment of reals. The order of the generators has no effect on the zonoseg.

The image of the *2-transition subcomplex* of $[0,1]^n$ is a compact subsegment of the zonoseg. The order of the generators affects this subsegment in a major way.

Usage

```
zonoseg( mat, e0=0, ground=NULL )

## S3 method for class 'zonoseg'
getsegment( x )

## S3 method for class 'zonoseg'
getsegment2trans( x )

## S3 method for class 'zonoseg'
print( x, ... )
```

Arguments

`mat`	a numeric matrix with 1 row whose entries determine the zonoseg. One or more entries must be non-zero. It is OK to have both positive and negative entries. `mat` can also be a numeric vector which is then converted to a matrix with 1 row.
`e0`	threshold for an entry of `mat` to be considered 0. Since the default is `e0=0`, by default an entry must be exactly 0 to become a loop in the associated matroid.
`ground`	The *ground set* of the associated matroid of rank 1 - an integer vector in strictly increasing order, or `NULL`. When `ground` is `NULL`, it is set to `1:ncol(mat)`. If `ground` is not `NULL`, `length(ground)` must be equal to `ncol(mat)`. The point `ground[i]` corresponds to the *i'th* column of `mat`.
`x`	a zonoseg object as returned by `zonoseg()`
`...`	not used

Details

A **zonoseg** object is a list with only 3 items: the associated matroid, the endpoints of the segment, and endpoints of the *2-transition subsegment*.

print.zonoseg() prints some information about the generators, and the endpoints of the segment plus the 2 vertices of the unit cube that map to these endpoints. It prints similar data for the 2-transition subsegment. Finally, it prints data on the associated matroid.

Value

zonoseg() returns a list with S3 class 'zonoseg'. In case of error, e.g. invalid mat, the function prints an error message and returns NULL.

getsegment() and getsegment2trans() return numeric 2-vectors - the min and max endpoints of the corresponding segments.

print.zonoseg() returns TRUE or FALSE.

Note

The *ground set* of positive integers should not be too sparse; otherwise performance may suffer.

References

Matroid - Wikipedia.
https://en.wikipedia.org/w/index.php?title=Matroid&oldid=1086234057

See Also

rank()

Examples

```
zono1 = zonoseg( c(1,-2,3,0,-3,-4) )
zono1

# generators:        6 -- 3 negative, 2 positive, and 1 loops.
#
# segment:                     [-9,4]
#      value pcube.1 pcube.2 pcube.3 pcube.4 pcube.5 pcube.6
# zmin    -9       0       1       0       0       1       1
# zmax     4       1       0       1       0       0       0
#
# 2-transition subsegment:   [-8,3]
#             value source.1 source.2 source.3 source.4 source.5 source.6
# tmin-2trans    -8        1        1        0        0        1        1
# tmax-2trans     3        0        0        1        1        0        0
#
# matroid:
# ground set:           6 points   {1 2 3 4 5 6}
# hyperplanes:          1     {4}
# rank:                 1
# loops:                1   {4}
# multiple groups:      1     {1 2 3 5 6}
# uniform:              FALSE
# paving:               TRUE
# simple:               FALSE
# This matroid is constructed from a 1x6 real matrix.
#      1  2 3 4  5  6
# [1,] 1 -2 3 0 -3 -4
#
# The summary of the simplified matroid is:
#     ground set:           1 points   {1}
#                 Point 1 corresponds to the multiple group {1 2 3 5 6} in the original ...
#      hyperplanes:          1     {}
#      rank:                 1
#      loops:                0   {}
#      multiple groups:      0   {}
#      uniform:              TRUE
#      paving:               TRUE
#      simple:               TRUE
#      This matroid is constructed from a 1x1 real matrix.
#           1+...+6
#      [1,]     -13


## so the 2-transition subsegment is a proper subset of the zonoseg
```

---

zonoseg-invert *invert points in a zonoseg*

Description

For points in a zonoseg, find points in the unit cube that map to those points.

Usage

```
## S3 method for class 'zonoseg'
invert( x, z, tol=0, ... )
```

Arguments

x	a zonoseg object as returned by the constructor zonoseg()
z	a numeric M-vector
tol	points that are within tol of a boundary point are taken to be that point
...	not used

Details

For a point in the interior of the zonoseg, there are infinitely many points in the cube that map to it. This function tries to find one with the fewest number of non-zero components.

Value

`invert.zonoseg()` returns a `data.frame` with M rows and these columns:

z	the given point
pcube	a point in the unit cube that maps to z. For the 2 boundary points, pcube is a vertex. If z is outside the zonoseg, pcube is all NA, and a warning message is printed

If the names of z are unique, they are copied to the row names of the output. The column names are copied from the ground set of the associated matroid.

In case of error, the function returns `NULL`.

See Also

`zonoseg()`

Examples

```
zono1 = zonoseg( c(1,-2,3,0,-3,-4) )
zono1
# generators:        6 -- 3 negative, 2 positive, and 1 loops.
#
# segment:                   [-9,4]
#      value pcube.1 pcube.2 pcube.3 pcube.4 pcube.5 pcube.6
# zmin    -9       0       1       0       0       1       1
# zmax     4       1       0       1       0       0       0
#
# 2-transition subsegment:   [-8,3]
#             value source.1 source.2 source.3 source.4 source.5 source.6
# tmin-2trans    -8        1        1        0        0        1        1
# tmax-2trans     3        0        0        1        1        0        0


z = c( 0, -3*pi, pi, 2*pi, getsegment(zono1) )

invert( zono1, z )
#           z   pcube.1   pcube.2   pcube.3   pcube.4   pcube.5   pcube.6
# 1  0.000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
# 2 -9.424778        NA        NA        NA        NA        NA        NA
# 3  3.141593 1.0000000 0.0000000 0.7138642 0.0000000 0.0000000 0.0000000
# 4  6.283185        NA        NA        NA        NA        NA        NA
# 5 -9.000000 0.0000000 1.0000000 0.0000000 0.0000000 1.0000000 1.0000000
# 6  4.000000 1.0000000 0.0000000 1.0000000 0.0000000 0.0000000 0.0000000
```

`zonotope-getters` *zonotope get functions*

Description

get some important members of a zonotope

Usage

```
## S3 method for class 'zonotope'
getmatrix( x )

## S3 method for class 'zonotope'
getmatroid( x )

## S3 method for class 'zonotope'
getcenter( x )
```

Arguments

`x`	a zonotope object - a **zonohedron**, a **zonogon**, or a **zonoseg**

Value

`getmatrix()` returns the matrix originally used to construct the zonotope `x`.

`getmatroid()` returns the matroid (possibly nonsimple) constructed from the matrix

`getcenter()` returns the center of the zonotope; which is also the center of radial symmetry. If `x` is an object-color solid, the center corresponds to the 50% graypoint. For the whitepoint multiply by 2.

See Also

`zonohedron()`, `zonogon()`, `zonoseg()`

`zonotope-props` *zonotope properties*

Description

Get some important boolean properties of a zonotope.

pointed means that 0 is a vertex of the zonotope. *salient* means that 0 is in the boundary of the zonotope. So pointed implies salient, but not the reverse.

A zonotope has an associated *convex cone* - allow the coefficients of the generators to be any non-negative numbers. For convex cones, *pointed* means that the cone is in an open linear halfspace (except for 0). And *salient* means that the cone is in a closed linear halfspace (the cone may contain a line).

In terms of generators (of both zonotopes and convex cones), *pointed* means that the generators are in an open linear halfspace (except for 0 generators). And *salient* means that the generators are in a closed linear halfspace.

Usage

```
## S3 method for class 'zonotope'
is_pointed( x )

## S3 method for class 'zonotope'
is_salient( x )
```

Arguments

`x`	a zonotope object - a **zonohedron**, a **zonogon**, or a **zonoseg**

Details

For a **zonohedron**, if 0 is in the interior of an edge or a facet, then the zonohdron is salient but not pointed.
For a **zonogon**, if 0 is in the interior of an edge, then the zonogon is salient but not pointed.
For a **zonoseg**, both *pointed* and *salient* are equivalent to 0 being a boundary point. And this is equivalent to all the non-zero generators having the same sign (all negative or all positive).

Value

`TRUE` or `FALSE`

References

Zonohedron - Wikipedia.
https://en.wikipedia.org/wiki/Zonohedron

See Also

`zonohedron()`, `zonogon()`, `zonoseg()`

Examples

```
zono1 = zonoseg( c(1,-2,3,0,-3,-4) )

is_pointed( zono1 )
# [1] FALSE

is_salient( zono1 )
# [1] FALSE
```

Index