Package 'levelSets'

June 22, 2026

Version 0.8.2

Date 2026-06-15

Title Ray-Based Mapping and Visualization of Level Sets (Excursion Sets)

Author Richard Raubertas [aut, cre]

Maintainer Richard Raubertas <rrprf@emvt.net>

License GPL (>= 3)

Depends R (>= 3.5.0)

Imports graphics, stats, tools, utils, proxy, withr

Suggests ggplot2, randomForest, palmerpenguins, knitr, rmarkdown

Description An (upper) level set of a function is the set of inputs for which the function value is at or above a specified threshold. (Also called an excursion set). Applications of level sets include confidence or credible regions for parameters of statistical models, where the function is the likelihood or posterior density; regions where classification rules assign high probability to a given class; and scientific or engineering models where one is interested in input regions for which model output is above a threshold. This package maps out the boundary of a level set by finding its intersections with collections of 1-dimensional rays, generalizing a proposal by Kim and Lindsay (Statistica Sinica 21:923-948, 2011). Tools are provided to generate rays, find intersections, and visualize results. The package makes few assumptions about the studied function: it may be discontinuous, it may have a complicated feasible region, and the target level set may be non-convex or have multiple, disconnected parts. Vignettes describe package usage and show examples with two to five input space dimensions.

LazyData yes

VignetteBuilder knitr, rmarkdown

NeedsCompilation no

Repository CRAN

Date/Publication 2026-06-22 14:30:02 UTC

Contents

`absTol` *Absolute (Difference Scale) Tolerance Values*

Description

Calculate absolute (difference scale) tolerance values to associate with a set of numeric values. The idea is that differences from `x` of that magnitude or less should be treated as effectively 0.

Usage

```
absTol(x, tol)
```

Arguments

`x`	Numeric vector.
`tol`	Vector with two non-negative scalars. The first value will be multiplied with `abs(x)`, the second will be treated as a constant; the larger of the two results is the calculated tolerance.

Value

Numeric vector like `x`, containing the associated tolerance values.

Note

If `x` contains infinite values and `tol[1]` is not 0, the result will be set to the maximum finite value, `.Machine$double.xmax`.

See Also

inpTol, respTol extract tol values from fnSpec and related objects. lsetPkgOpt describes how tolerances are used in this package. base::all.equal.numeric uses a different, non-continuous function of x to decide equality.

Examples

```
# The absolute tolerance is the larger of the minimum absolute tolerance
# and the relative tolerance:
x <- c(10000, 1000, 1, 0.001, 0.00001, 0)
absTol(x, tol=c(1e-3, 1e-4))

# When 'tol[1]' is 0, only absolute tolerance, 'tol[2]', is used,
# regardless of 'x' values:
absTol(c(-Inf, -10, -1, 0, 1, 10, Inf, NA), tol=c(0, 1e-6))

# Comparison to 'base::all.equal'.
x0 <- 1e-2
x <- x0 + 1e-3*c(-1, 0, 1)
atol <- absTol(x, tol=c(x0, 1e-8))  # ~1e-4
(abs(x - x0) <= atol)  # FALSE, TRUE, FALSE

# 'base::all.equal' uses a discontinuous criterion that makes it less
# sensitive to differences on the down side than the up side, when 'x'
# is close in magnitude to 'tolerance'.
all.equal(x0-1e-3, x0, tolerance=x0)  # TRUE
all.equal(x0, x0, tolerance=x0)  # TRUE
all.equal(x0+1e-3, x0, tolerance=x0)  # "Mean relative difference: 0.09090909"
# Effectively  TRUE, TRUE, FALSE
```

---

axisRays *Rays Based on the Coordinate Axes of Input Space*

Description

Create rays based on the unit vectors for the coordinate axes, and sums and differences of those vectors.

Usage

```
axisRays(spec=NULL, degree=1, scale=NULL, origin=NULL, rect=NULL,
    slice=NULL, named=(degree <= 3), warnDegenerate=TRUE)
```

Arguments

spec	An fnObj or fnSpec object with specifications for the input space. If omitted it will be taken from rect or scale, one of which must be specified in that case.

degree	Non-negative integer less than or equal to the dimension of the input space (or of slice if specified). Rays vectors having up to degree non-zero coordinates will be generated. degree=0 will return an empty inpRays object.
scale	Optional inpScale object. If provided, the unit vector directions will be rescaled and/or rotated using it.
origin	Vector or 1-row matrix with the coordinates of a single point in input space. All rays start from this point (i.e., it is position 0 for each ray). origin must belong to both rect and slice if they are specified. The default is rectRefPt(rect) if rect is specified, and otherwise the point with all coordinates equal to 0..
rect	Optional inpRect object, an axis-aligned hyper-rectangle in input space. Rays will be scaled so that position 1.0 on each ray is its intersection with the boundary of rect.
slice	Optional vector or 1-row matrix specifying one slice of input space. (See ?sliceMat for information about slices.) All the generated rays will lie in the slice. If specified it must intersect rect, and origin must lie in the slice.
named	Logical scalar. If TRUE the rays will be named; see DETAILS.
warnDegenerate	Integer or logical scalar. When rect is specified and origin is on its boundary, it is possible for certain ray directions to lead to degenerate rays (i.e., positions 0 and 1 are the same point). Degenerate rays are not allowed. warnDegenerate controls whether they are silently dropped (0 or FALSE), dropped with a warning (1 or TRUE), or treated as an error (2).

Details

Let d denote the number of input space dimensions. If slice is specified, rays are initially generated in the lower d_s-dimensional hyperplane defined by slice, where d_s is the number of NA coordinates in slice. At the end the rays are brought back to the original d-dimensional input space by including the d - d_s fixed coordinates from slice. (If slice is not specified, set d_s = d in what follows.)

Rays are generated by the following steps.

Axis ray vectors of degree 1 are the unit vectors along the d_s coordinate axes. Specifying degree=2 adds unit vectors pointing along both diagonals for each pair of coordinate axes; degree=3 also adds unit vectors pointing along the four 3-way diagonals for each set of three coordinate axes, and so on. The total number of vectors depends on d_s and degree: sum((2^(1:degree)) * choose(d_s, 1:degree)) / 2. It increases rapidly with degree.

If scale is specified, these unit vectors are then scaled and/or rotated to reflect the relative scales and correlations among the input space dimensions: They are multiplied by the inverse of matrix W from scale. (This "unstandardizes" generic unit vectors to the natural scale of the input space.)

Vectors are then translated by adding origin to their coordinates.

Finally, if rect is specified, vector lengths are adjusted so that position 1 for each ray is its intersection with the boundary of rect. Note that when origin itself is on the boundary of rect, vectors pointing away from rect will lead to degenerate rays (positions 0 and 1 being the same point). In that case a vector pointing in the opposite direction will be tried. If that also leads to a degenerate ray, the vector will be dropped.

The created rays are unique with respect to a sign change in their directions; if ray `r` is included then the ray pointing in the opposite direction is not.

If `named` is TRUE, rays will be named based on the names of the input space dimensions in `spec`. However if `scale` includes correlations between dimensions, the degree 1 rays are rotated relative to the coordinate axes, so they will be renamed as `rot1`, `rot2`, etc. Names for higher degree directions will be based on these.

Value

An `inpRays` object. The origin will be the point specified by `origin`. The position 1.0 point for each ray will by default be distance 1 from the origin, but distances may differ if `scale` or `rect` is specified. Ray vectors of degree 1 will always be orthogonal to each other.

See Also

`inpRays`, `randomRays`, `inpScale`

Examples

```
oldopt <- lsetPkgOpt(reset=TRUE)  # default options

spec <- fnSpec(inpnames=c("X1", "X2"))
layout(matrix(1:4, nrow=2, byrow=TRUE))

rays <- axisRays(spec, degree=2)
plot(rays, eqAxes=TRUE, main="Standard axis rays of degrees 1 & 2")
abline(h=0, v=0, lty=3)

# Effect of 'scale' (rescaling only).
scl <- inpScale(c(1, sqrt(5)), spec=spec)
  # (Rays will be stretched in the X2 direction)
rays <- axisRays(spec, degree=2, scale=scl)
plot(rays, eqAxes=TRUE, main="Scaled axis rays")
abline(h=0, v=0, lty=3)

# Effect of 'scale' (rescaling and rotation).
scl <- inpScale(rbind(c(1, 2), c(2, 5)), spec=spec)
  # (Rays will be rotated counterclockwise and stretched in the X2 direction.)
rays <- axisRays(spec, degree=2, scale=scl)
plot(rays, eqAxes=TRUE, main="Scaled & rotated axis rays")
abline(h=0, v=0, lty=3)

# 'scale' is applied before ray vectors are extended to the boundary of
# 'rect'.
rect <- inpRect(rbind(c(-1, 0), c(1, 2)), refpt=c(0.5, 0.5), spec=spec)
rays <- axisRays(spec, degree=2, scale=scl, rect=rect)
  # Default ray origin is taken as reference point of 'rect'.
plot(rays, eqAxes=TRUE, main="Scaled & rotated axis rays, with 'rect'")
abline(h=0, v=0, lty=3)

layout(matrix(1))
```

```
#----- 3D example to illustrate 'slice'ing.
spec <- fnSpec(inpnames=c("X1", "X2", "X3"))
slice <- c(NA, NA, -2)
rays <- axisRays(spec, degree=2, slice=slice, origin=c(0, 0, -2))  # 4 rays
print(rays, vecForm=FALSE)  # all rays lie in the slice with X3 = -2

# Specifying 'rect' and putting the ray origin on its boundary can lead
# to ray reversal or dropping rays:
rect <- inpRect(rbind(c(-1, 0, -2), c(1, 2, 3)),
                refpt=c(1, 0, -2),  # upper bdry of X1, lower of X2, X3
                spec=spec)
rays <- axisRays(spec, degree=2, slice=slice, rect=rect,
                 warnDegenerate=FALSE)
print(rays, vecForm=FALSE)  # 'X1', 'X1(-)X2' rays have been reversed
                            # and 'X1(+)X2' ray has been dropped
plot(rays, eqAxes=TRUE,
     main="Axis rays (degrees 1 & 2), origin on bdry of 'rect'")

lsetPkgOpt(oldopt)  # restore starting options
```

---

banana_2dEx *Negative of Rosenbrock's Banana Function in Two Dimensions*

Description

List with components that set up a level set problem for the negative of Rosenbrock's banana function. The response function has a maximum value of 0 at the input point (x1, x2) = (1, 1). For illustration the level set of interest is defined by the -50 contour, which is non-convex and highly curved. Also for illustration, the feasible region is set to be the half-plane x1 >= -2.

Usage

```
banana_2dEx
```

Format

The list has components:

`fobj` `fnObj` object with the response function and its gradient, input space specifications, and feasible region function.

`thresh` Value of the response function that defines the level set of interest (-50).

`rect0` `inpRect` object with an initial hyper-rectangle in input space, to guide the search for the level set boundary.

Source

The code that creates the above list is included in the package, in the location returned by `system.file("data_create/banan`

bdryFromRays *Intersections of Rays with the Boundary of a Level Set*

Description

Identify the boundary of a level set by finding points where a set of rays intersect it. Use these to identify segments along each ray that appear to lie entirely in the level set (i.e., *chords* of the level set).

Usage

```
bdryFromRays(x, lsetthresh=NULL, rays, control=list())
```

Arguments

x	fnObj object containing definitions of the response function and its input space.
lsetthresh	Response value that defines the level set of interest. If rays is a respProfiles object, the default lsetthresh will be taken from it, but if lsetthresh is specified, it overrides any existing threshold value in rays.
rays	An inpRays object defining a set of rays in input space. It may also be a respProfiles object, which will be used to initialize the search for boundary intersections along the rays that were profiled. In that case, the profiles must have been created with the same fnObj object as x.
control	Optional list with parameters to control the search process. See ?srchControl for the available parameters and their defaults. (The only parameters relevant to this function are initPosns, initPosns2, and bothDirections.) Any value specified here overrides the default from srchControl().

Details

The level set of interest is defined as the set of input points at which the response function has values greater than or equal to lsetthresh. Along a given ray a boundary of the level set is either (a) a point where the response function transitions from greater than or equal to lsetthresh to less than lsetthresh, or vice versa; or (b) a point where the ray crosses the boundary of the feasible region of the response function, with a response value still at or above lsetthresh. Both the level set and the feasible region are assumed to be closed (although possibly unbounded) sets, so boundary points are always feasible and have response values greater than or equal to lsetthresh.

Points along rays are specified by their *position*. For a given ray, position along it is measured in units of the length of the line segment from the ray origin (defining position 0) to the ray's second defining point, given by rayPosn1(rays) (position=1). See inpRays.

For a given ray the response function is evaluated at positions specified in control$initPosns. If any of those positions belongs to the level set (and assuming the level set has non-negligible volume in the input space), we know there is a segment of the ray, containing that position, that lies in the level set. This function searches for the limits of that segment in both directions. If none of the initial positions belongs to the level set, or if a segment appears to extend beyond the range of initPosns, positions in control$initPosns2 are added. If the segment still appears to extend

further, the segment is marked as *censored* in that direction. Otherwise the segment endpoint in that direction is a boundary point of the level set.

Thus searching for level set boundary points using rays naturally leads to obtaining line segments that belong to the level set, and that is the form in which this function returns the result. Each segment is defined by its two endpoints. An endpoint is either an actual boundary point of the level set, or a censored boundary point if the level set appears to extend beyond it in the direction of the segment. Censoring status for a pair of endpoints is reported in the `lsetcens1` and `lsetcens2` columns of the data frame returned by `segInfo()`.

A given ray may have 0, 1, or >1 level set segments. The ability to find segments, and to distinguish multiple short segments from a single long one, depends on the range and spacing of the starting positions in `initPosns` and `initPosns2`. In particular, if responses at a consecutive pair of `initPosns` are on the same side of `lsetthresh`, it is assumed that all of the points between them also lie on that side. This may miss boundary crossings if the response profile along the ray is not unimodal and the peaks or troughs are smaller than the spacing between test positions. Adding additional values to `initPosns` may allow multiple crossings to be found.

Although not required, it is highly recommended that the ray origin be a point in the level set. This will improve the chance that useful level set segments and boundary points will be found.

If `control$bothDirections` is TRUE and the rays have an associated hyper-rectangle (`rayRect(rays)`), it is recommended that the ray origin not be on the rectangle's boundary. Otherwise reversed rays are likely to be degenerate and dropped, so the search for level set boundary intersections in the reverse direction will be ineffective. Also in this case, note that when a ray is reversed, position 1 on the reversed ray is its intersection with the boundary of the rectangle. That position may not have the same Euclidean distance from the ray origin as the original ray (e.g. if the ray origin is not at the center of the rectangle).

Information about level set segments can be extracted from the return value with `segInfo()`. Information about individual segment endpoints can be extracted with `respInfo()` and `ptCoord()`.

The `lsetSegs` class has `summary`, `plot` and `pairs` methods to display the results. See `?lsetSegs` for more information.

Value

An object with S3 class `lsetSegs`. It contains pairs of points where (a) each point in a pair lies along one of the rays in `rays` (possibly with negative position if `bothDirections` is TRUE); (b) each point in a pair belongs to the level set; and (c) all the points on the line segment connecting them also appear to belong to the level set. The goal is that the segment endpoints should lie on the *boundary* of the level set. However this may not be possible if the boundary in the direction of the line containing the segment lies outside the search region determined by `control` components `initPosns`, `initPosns2`, and `bothDirections`.

See Also

`lsetSegs`, `respProfiles`, `fnObj`, `inpRays`, `segInfo`, `respInfo`, `srchControl`. Functions `axisRays` and `randomRays` generate useful sets of rays to explore the boundary of the level set. `lsetSegsCheck` checks that points within claimed level set segments actually do belong to the level set. `bdrySearch` extends this function by adaptively selecting multiple ray origins to better explore the full boundary of a level set. `slicedBdryFromRays` extends this function by using search rays restricted to one or more slices of the input space.

Examples

```
oldopt <- lsetPkgOpt(reset=TRUE)

#----- 2D multimodal example (level set has two separate elliptical parts).
Ex <- dbl_ellipse_2dEx  # see '?dbl_ellipse_2dEx'
fobj <- Ex$fobj
thresh <- Ex$thresh

# Specify the search region and generate some rays scaled to it.
rect1 <- inpRect(rbind(c(-5, -10), c(15, 12)), spec=fobj)
set.seed(1)
rays1 <- randomRays(50, origin=c(2, 1), rect=rect1)

segs1 <- bdryFromRays(fobj, rays=rays1, lsetthresh=thresh)  # 'lsetSegs' object
summary(segs1)
plot(segs1, rect=rect1, main="Segments found along 50 random rays")

# Some segments cross the gap between level set parts.  Check for and fix
# invalid segments.
chk <- lsetSegsCheck(segs1, fnobj=fobj, posns=(1:9)/10)
table(chk$validIn)  # 6 invalid segments
table(chk$validOut)  # all fixed, so ...
segs1 <- lsetSegs(chk)
summary(segs1)
plot(segs1, rect=rect1,
     main="Segments along 50 random rays, invalid segs fixed")

# Generate a second set of rays with an origin in the 2nd ellipse, and
# find segments along them.  Make use of what we've learned from 'segs1':
# - Scale the rays to a rectangle that bounds all the level set
#   boundary points found so far.
rect2 <- segBoundingRect(segs1, expand=1.1)
rays2 <- randomRays(50, origin=c(5, 10), rect=rect2)

# - Since 'segs1' showed that the level set has multiple parts, it is a
#   good idea to set control parameter 'initPosns' to a finer set of ray
#   positions:  the spacing between positions should be smaller than the
#   distance between separate level set parts.
segs2 <- bdryFromRays(fobj, rays=rays2, lsetthresh=thresh,
                      control=list(initPosns=(0:10)/10))
chk <- lsetSegsCheck(segs2, fnobj=fobj, posns=(1:9)/10)
table(chk$validIn)  # all valid
summary(segs2)
plot(segs2, rect=rect2, main="Segments found along 50 random rays, origin 2")

# Segments can be combined into a single 'lsetSegs' object.
segs12 <- c(segs1, segs2)
plot(segs12, rect=rect2,
     main="Segments found along 100 random rays, 2 origins")

lsetPkgOpt(oldopt)
```

bdrySearch *Ray-Based Exploration of the Boundary of a Level Set*

Description

Identify the boundary of a level set by finding pairs of boundary points, such that the line segment connecting each pair appears to lie entirely in the level set. An adaptive, ray-based search is carried out. At each step a new ray origin is chosen, rays from that origin are generated, and their boundary intersection pairs are found. The choice of origins attempts to explore as much of the boundary as possible. Results are accumulated over all steps into an `lsetSegs` object, which is returned along with the sequence of ray origins that were used.

Usage

```
bdrySearch(x, lsetthresh, rect, scale=NULL, initOrigin=NULL,
    currentBdry=NULL, control=list())
```

Arguments

`x`	`fnObj` object containing definitions of the response function and its input space.
`lsetthresh`	Response function value that defines the level set of interest.
`rect`	`inpRect` object specifying the hyper-rectangle in input space where the search for boundary points is focused. Search rays extending from each origin will be defined such that position 1 along a ray is its intersection with the boundary of `rect`.
`scale`	Optional `inpScale` object specifying scaling to be applied to input space dimensions when calculating distances and generating rays. May also be a vector or a square matrix from which an `inpScale` object will be created; see `?inpScale` for details. The default is to use the lengths of the sides of `rect` as scaling factors for the input space dimensions.
`initOrigin`	Optional vector or 1-row matrix specifying the initial ray origin to use in the search. It must belong to `rect`. Default value: If `currentBdry` is specified and is not empty, the inital origin is derived from it according to `control$originCrit` (see DETAILS). Otherwise the reference point of `rect` is used.
`currentBdry`	Optional object containing existing information about the level set boundary: either an `lsetSegs` object (e.g. from a call to `bdryFromRays()`) or the result of a previous call to `bdrySearch()`, `slicedBdrySearch()` or `slicedBdryFromRays()`.
`control`	Optional list with parameters to control the search process. See `?srchControl` for the available parameters and their defaults. Any value specified here overrides the default from `srchControl()`.

Details

See `?bdryFromRays` for details about the search for level set boundary intersections along individual rays.

For each new search origin, rays are generated based on `control$rayType` and the associated parameters in `control`, and modified according to `scale`. The search for level set boundary intersections along each ray is in both directions from position 0, with `control$initPosns` and `control$initPosns2` applied separately for each half-line. When the ray origin is in the level set, typically this results in one or more segments along each line, whose endpoints are boundary points of the level set.

The adaptive selection of new origins relies on success at finding level set segments in previous steps (or in `currentBdry`): new candidate origins are generated from the accumulated set of level set line segments using `control$originBias`. From among the candidates one is selected according to the method `control$originCrit` (described below). The search stops when `control$maxSteps` is reached, or earlier if the set of candidate origins is empty (e.g., if the search is not finding level set boundary segments).

The `maxproj` criterion focuses the search for the level set boundary in a particular direction in input space: the candidate origin that has maximum position when projected onto the `control$srchDirection` vector is selected. (Position is equal to the inner product of the candidate vector with the search direction vector, divided by the square of the length of the latter.) Note that the projection does not use the `scale` argument to transform input space coordinates.

The other two criteria attempt to encourage wide exploration of the level set. The `maximindist` criterion selects the candidate origin that maximizes the minimum distance to any existing origin. Distance is computed as Euclidean distance after transforming point coordinates according to `scale`.

The `minimaxdist` criterion selects the candidate origin that minimizes the maximum distance from any current boundary point to the nearest (existing + new) origin. Distance is computed as Euclidean distance after transforming point coordinates according to `scale`.

If requested by `control$updateRect1` and `control$updateScale`, at each step the search region `rect` and input scaling `scale` will be updated. The updates are based on previously found level set segments. The updates will be done only if at least `d` segments are available, to avoid basing `rect` or `scale` on skimpy information about the level set.

Value

List with components:

`segs`	`lsetSegs` object containing pairs of points in the level set, such that all points on the line segment connecting a pair also appear to belong to the level set. The goal is that the segment endpoints should lie on the *boundary* of the level set, although this may not be possible if the boundary in the direction of the line segment lies outside the search region determined by `rect`, `control$initPosns`, and `control$initPosns2`.
`srchOrigins`	Matrix containing the coordinates of any search origins in `currentBdry`, followed by any new search origins generated during the function call. All new origins will belong to `rect`.
`rect`	`inpRect` object, the search region used for the last step of the search. Same as the argument `rect` if `control$updateRect1` and `control$updateRect2` are both 0.
`scale`	`inpScale` object, the scaling of input space dimensions used for the last step of the search. Same as the argument `scale` if `control$updateScale` is 0.

If argument currentBdry was provided, the return value will combine the information in it with the new level set segments and origins.

See Also

bdryFromRays, lsetSegs, srchControl. lsetSegsCheck checks the assumption that points along each level set segment actually do belong to the level set. slicedBdrySearch extends this function to do a separate boundary search for each of a set of slices of the input space.

Examples

```
oldopt <- lsetPkgOpt(reset=TRUE)

#----- 2D multimodal example (level set has two separate elliptical parts),
Ex <- dbl_ellipse_X1bnd_2dEx  # see '?dbl_ellipse_2dEx'
fobj <- Ex$fobj
thresh <- Ex$thresh
rect1 <- Ex$rect0

desc <- "Two-ellipse level set, with feasible region X1 \u2265 0"

# Boundary search with defaults:
srch1 <- bdrySearch(fobj, lsetthresh=-40, rect=rect1)
segs1 <- srch1$segs
srch1$srchOrigins   # d^2 = 4 origins used, 2 rays per origin
summary(segs1)  # 10 segments on 8 lines
plot(segs1, rect=srch1$rect, main=desc, sub=paste0("\nbdrySearch(): ",
     "defaults"))

# Clear evidence that the level set has disconnected parts.  Need to
# increase the density of initial positions along each ray to have a
# good chance of detecting that for individual rays.
srch1a <- bdrySearch(fobj, lsetthresh=-40, rect=rect1,
                     control=list(initPosns=(0:4)/4))
segs1a <- srch1a$segs
summary(segs1a)  # 12 segments on 8 lines
plot(segs1a, rect=srch1a$rect, main=desc, sub=paste0("\nbdrySearch(): ",
     "initPosns=(0:4)/4"))

# Use the 'control' argument to increase the number of search steps and rays.
srch2 <- bdrySearch(fobj, lsetthresh=-40, rect=rect1,
                    control=list(initPosns=(0:4)/4,
                                 maxSteps=8,  # increases number of origins
                                 axisDegree=2))  # increases rays per origin
segs2 <- srch2$segs
srch2$srchOrigins   # 8 origins used, 4 rays per origin
summary(segs2)  # 46 segments on 32 lines
plot(segs2, rect=srch2$rect, main=desc, sub=paste0("\nbdrySearch(): ",
     "initPosns=(0:4)/4, axisDegree=2, maxSteps=8"))
  # Clear picture of boundaries of both parts of the level set.

# Check segment validity.
```

```
chk <- lsetSegsCheck(segs2, fnobj=fobj, posns=(1:4)/5)
table(chk$validIn)  # All appear valid

lsetPkgOpt(oldopt)  # restore previous settings
```

---

boundingRect *Hyper-Rectangle Bounding a Set of Points in Input Space*

Description

Create an axis-aligned hyper-rectangle just large enough to contain a specified set of points in input space. Optionally expand or contract the hyper-rectangle from that size.

Usage

```
boundingRect(..., refpt=NULL, spec=NULL, expand=1.0, warnEmpty=TRUE)
```

Arguments

`...`	One or more vectors, matrices, or `inpRect`, `inpRays` or `lsetSegs` objects, containing coordinates of points in input space. Infinite coordinates are not allowed, and for each dimension there must be at least one point with a non-NA coordinate value.
`spec`	`fnObj` or `fnSpec` object with specifications for the input space. Optional if `...` contains an object with that information.
`refpt`	A vector or one-row matrix containing the coordinates of the point to use as the reference point for the new hyper-rectangle. The new hyper-rectangle will always include it. This is also the point relative to which `expand` will be applied. The default is the centroid of the points in `...`.
`expand`	Scalar expansion factor for the hyper-rectangle, relative to what is required to just contain all the points in `...` and `refpt`.
`warnEmpty`	Logical scalar. If TRUE, a warning will be raised if `...` does not contain any points.

Details

All the objects in `...` are combined into a single matrix of point coordinates. For `inpRect` objects, the lower and upper corner points are used. For `inpRays` objects the position 0 and position 1 points are used. For `lsetSegs` objects, the coordinates of the segment endpoints are used.

Each point in the combined matrix is adjusted so that its distance from `refpt` is multiplied by factor `expand`. The smallest axis-aligned hyper-rectangle containing all of the adjusted points and `refpt` is calculated.

`inpRect` objects are required to have positive volume, so any sides of the hyper-rectangle that have near-zero length will be increased by an amount based on `inpTol(spec)`.

Value

An `inpRect` object containing the calculated hyper-rectangle, or NULL if `...` does not contain any points.

See Also

`inpRect`. `segBoundingRect` is a similar function specifically for bounding level set segments. It handles expansion of the bounding rectangle differently.

Examples

```
oldopt <- lsetPkgOpt(reset=TRUE)

fspec <- fnSpec(inpnames=c("X1", "X2"))
set.seed(1)
pts <- matrix(runif(10), ncol=2)  # 5 points in unit square
rect <- boundingRect(pts, refpt=c(0, 0), spec=fspec)
plot(rect, main="Bounding rectangle for 5 points and a reference point")
points(pts)  # rectangle just includes 'refpt' and 'pts'

# Expand the rectangle away from 'refpt':
rect <- boundingRect(pts, refpt=c(0, 0), spec=fspec, expand=1.2)
plot(rect, main="Bounding rectangle expanded away from 'refpt'")
points(pts)  # rectangle includes 'refpt' and a margin around 'pts'

# Rectangle is guaranteed to have non-zero volume, even when points lie in
# a lower dimensional space:
rect <- boundingRect(pts[1, ], refpt=pts[1, ], spec=fspec)
rectSize(rect)  # sides are small but not 0

lsetPkgOpt(oldopt)
```

---

circuitFailure_3dEx *Example of a 3D Confidence Region as a Level Set*

Description

List with components that illustrate the use of the `levelSets` package to identify and visualize a confidence region for the parameters of a statistical model.

Usage

```
circuitFailure_3dEx
```

Format

The list has components:

`fobj` `fnObj` object with the response function, input space specifications, and feasible region function.

`thresh` Log-likelihood value that defines the 95% asymptotic confidence region.

`theta_mle` Three-element vector containing the maximum likelihood estimates of the parameters.

`theta_vcov` 3-by-3 matrix containing the approximate variance-covariance matrix for 'theta_mle'.

`bsrch` List returned by `bdrySearch()` with results from a search for level set boundary points and segments.

`slsrch` List returned by `slicedBdrySearch()`, with results from a level set search within six slices of input space, each defined by a specified value of parameter 'scale'.

The `theta_mle` and `theta_vcov` components were obtained with `stats::optim`.

Details

The data are failure times of integrated circuits under accelerated testing. 4156 circuits were tested for 1370 hours at elevated temperature and humidity. The goal was to estimate the proportion of circuits with a manufacturing defect that leads to relatively early failure. 28 of the 4156 circuits failed, at various times, during the test; the failure time of the remaining circuits was censored at 1370 hours.

The data and proposed statistical model are given in section 18.1 of Meeker, Hahn, and Escobar (2017). The model assumes that a fraction 'dfrac' of circuits are defective, with failure times having a Weibull distribution with parameters 'shape' and 'scale'. The remaining fraction '1 - dfrac' are assumed to have essentially infinite failure times. Meeker, Hahn, and Escobar refer to this as the limited failure population (LFP) model. It is also a version of a "cure model", where '1 - dfrac' is the proportion of patients who are cured and thus never succumb to the disease being studied.

Here we examine the 3-dimensional likelihood ratio-based 95% confidence region for the full set of model parameters `(dfrac, shape, scale)`. Therefore the response function is the log-likelihood, and the confidence region is the level set consisting of parameter vectors whose log-likelihood is no more than `qchisq(0.95, df=3) / 2 = 3.907` below its maximum.

Source

The code that creates the above list is included in the package, in the location returned by `system.file("data_create/circ`

References

Meeker, William Q., Hahn, Gerald J., Escobar, Luis A. *Statistical Intervals: A Guide for Practitioners and Researchers, 2nd ed*. John Wiley & Sons, 2017.

dbl_ellipse_2dEx *Example of a Multi-Part Level Set in a 2D Input Space*

Description

Lists with components that illustrate the use of the `levelSets` package to identify and visualize a level set of a function. The input space has two dimensions, the response function has two peaks, and the level set consists of two separate elliptical regions around those peaks. The only difference between `dbl_ellipse_2dEx` and `dbl_ellipse_X1bnd_2dEx` is that the latter adds an arbitrary constraint to illustrate the specification of a feasible region for the response function.

Usage

```
dbl_ellipse_2dEx
```

Format

Each list (`dbl_ellipse_2dEx` and `dbl_ellipse_X1bnd_2dEx`) has components:

`fobj` `fnObj` object with the response function, input space specifications, and feasible region function.

`thresh` Value of the response function that defines the level set of interest (-40).

`origin0` 1-row matrix with the coordinates of a point in input space to use as a ray origin.

`rect0` `inpRect` object with a hyper-rectangle in input space, to use as an initial search region for the level set boundary.

`rays0` `inpRays` object containing 50 random rays, with origin and position 1 rectangle given by `origin0` and `rect0` respectively.

`segs`, `segs_checked` `lsetSegs` objects with level set segments for `fobj`. `segs` was obtained by applying `bdryFromRays()` to `rays0`. Some of the segments are invalid, and `segs_checked` contains the segments after being checked and fixed by `lsetSegsCheck()`.

Source

The code that creates the above lists is included in the package, in the location returned by `system.file("data_create/dbl_`

feasBnds *Box Constraints on a Feasible Region*

Description

Return any box constraints on point coordinates in the feasible region of a response function.

Usage

```
feasBnds(x)
```

Arguments

x	An fnSpec object, or object containing one. (That includes fnObj, inpRays, inpRect, inpScale, respProfiles and lsetSegs objects.)

Details

These represent only the box constraints on the feasible region that were specified by the feasbnds argument to fnObj. The actual feasible region may be more complicated: the box constraints are combined with any other feasible region restrictions specified by argument feasfn when an fnObj object is created.

Value

Two-row matrix with d columns, where d is the number of dimensions in the input space. Each column contains the lower and upper bounds for coordinates for the corresponding dimension, as specified by x. If no box constraint was specified for a given dimension, the default is -Inf for lower bound and Inf for upper bound.

See Also

fnSpec, fnObj

Examples

```
# See examples in '?fnSpec'
```

---

fnObj *Create an Object that Represents a Response Function*

Description

Create an object containing a response function and related information, including about its input space, feasible region, and gradient availability.

Usage

```
fnObj(x, respfn, feasfn=NULL, hasgrad=FALSE,
    derivmethod={ if (hasgrad) "gradient" else "none" }, warnNAresp=2,
    warnInfresp=1, ...)
```

Arguments

x	A character vector of unique names for the dimensions of the input space. They must be valid names for R objects (see base::make.names). Or an existing fnSpec object with dimension names and other information about the input space and response function.

respfn — Function that takes a matrix of input point coordinates as its first argument (one input point per row), and returns a vector with response function values at those points.

feasfn — Optional function that takes a matrix of input point coordinates as its first argument, and returns a logical vector indicating whether each point is in the feasible region of respfn.

hasgrad — Logical scalar. TRUE means that respfn has the ability to also calculate the gradient of the response function at input points (i.e., the vector of derivatives of response with respect to each input dimension). In that case, respfn must accept an argument inclGrad, a logical scalar that controls whether gradients are calculated on a particular call. When gradients are calculated, respfn should return them as the gradient attribute of the vector of response values. The attribute should be a matrix with each row containing the gradient vector at the corresponding input point. The default for hasgrad is FALSE.

derivmethod — Character string indicating whether and how to calculate the derivative of the response function with respect to position along a line or ray. One of gradient, finite difference, or none (to skip the calculation). The default is gradient if hasgrad is TRUE, and none if not.

warnNAresp, warnInfresp — Scalars specifying how response function values of NA or +/-Inf, at points called feasible by feasfn, should be handled. Valid values are 0 (or FALSE), 1 (or TRUE), and 2. See DETAILS.

... — Additional arguments for specific methods (see ?fnObj.character), plus any arguments to be included in each call to respfn and feasfn. Must not include inclGrad.

Details

The purpose of fnObj objects is to wrap user-supplied response and feasibility functions, providing a consistent interface for other package functions. This includes consistent handling of NA's and infeasible input points, and a consistent structure for return values.

A simple way to evaluate the response function at a set of points x, given an fnObj object for the function, is respInfo(x, fnobj). This will return a data frame that includes response value and other information (including feasibility) for each point.

Feasible region of input space. The feasible region can be specified explicitly and/or implicitly. Explicit specification is through the feasfn argument (and/or feasbnds for the .character method). Implicit specification is through respfn: it can simply return NA for any point it considers infeasible. (If the latter approach is used, warnNAresp should be set to 0 or FALSE to turn off warnings/errors due to NA response values.) A point will be considered feasible only if it passes both implicit and any specified explicit checks.

If provided, a feasfn function should be robust: it should accept any set of numeric point coordinates without generating errors, and return only TRUE or FALSE for each point. (Any NA values returned by feasfn will be treated as FALSE.) In addition the algorithms in this package assume that the feasible region is a *closed* (although possibly unbounded) set. Thus weak inequalities ("<=", ">=") rather than strong inequalities ("<", ">") should be used when defining the feasible region in a feasfn function.

respfn will never be called with input points that explicit feasibility checks call infeasible. Instead the response value at those points (and the gradient vector if applicable) is set to NA. Conversely, any point for which respfn returns NA will implicitly be considered infeasible. Depending on the value of warnNAresp this will be handled silently (0 or FALSE), with a warning (1 or TRUE), or as an error (2). The default is 2, treating unexpected NA response values as errors.

Argument warnInfresp offers the same options to report when +/-Inf values are returned by respfn. Points with infinite response are not considered infeasible. The default for warnInfresp is 1.

Gradients and derivatives. Coding and use of response function gradients is optional. This package does not assume response functions are differentiable. The only use of derivatives is to detect possibly missed level set boundary crossings along rays. (For example, if the derivative with respect to position along a ray is negative at two consecutive crossings, that suggests there is another crossing between them with a positive derivative.) If a level set is known to be a single convex region of input space this check is superfluous, and derivmethod can be set to none. Similarly if the response function is highly discontinuous or piecewise-constant, derivatives won't be useful and derivmethod should be none.

See ?fnObj.character and ?lsetPkgOpt for a description of tolerance specifications for coordinates of points in input space, and for response function values.

Accessor and update functions. Accessor functions can be used to extract individual specifications from an fnObj object: inpDim(), inpNames(), feasBnds(), inpTol(), respTol(), hasGrad(). fnSpec() will extract an fnSpec object that includes the first five of those.

There is an update method for fnObj objects that can be used to update some of its specifications. See ?update.fnObj for details.

Value

An object with S3 class fnObj.

See Also

fnSpec; respInfo to evaluate the response function in an fnObj object at a set of points. lsetPkgOpt. lsetPkgOpt("warn.showBadRespPts") controls whether and how many point coordinates are printed as part of warnings or errors due to NA or infinite response function values.

Examples

```
oldopt <- lsetPkgOpt(reset=TRUE)  # default options

# Function of 3 input variables ("X1", "X2", "X3"), with gradient available,
# and an extra argument 'powr'.
f <- function(x, inclGrad=FALSE, powr) {
  y <- x[, "X1"]^powr[1] + x[, "X2"]^powr[2] + x[, "X3"]^powr[3]
  if (inclGrad) {
    grad <- array(NA_real_, dim=dim(x), dimnames=dimnames(x))
    for (j in 1:3) {
      grad[, j] <- { if (powr[j] == 0)  0
                     else  powr[j] * (x[, j]^(powr[j] - 1)) }
    }
    attr(y, "gradient") <- grad
```

```
  }
  y
}
# Associated feasibility function, always passed the same extra argument(s).
feasf <- function(x, powr) {
  feas <- rep(TRUE, nrow(x))
  # Want to avoid applying non-integer powers to negative numbers.
  fracpow <- (powr != round(powr))
  for (j in which(fracpow)) {
    feas <- feas & (x[, j] >= 0)  # NOTE: '>=', not '>'. Feasible set
                                  # should be _closed_.
  }
  feas
}

fobj <- fnObj(c("X1", "X2", "X3"), respfn=f, feasfn=feasf, hasgrad=TRUE,
              powr=c(-1, 0, 2.5))

# Evaluate the response function at a set of points.
pts <- rbind(c(3, -2, 0), c(2, 0, -1), c(1, 6, 1), c(-1, 1, 1))
colnames(pts) <- inpNames(fobj)
resp1 <- respInfo(pts, fnobj=fobj)  # data frame
resp1  # response is NA for infeasible points.  No level set threshold has
       # been set, so 'lset' is NA for feasible points.

# Define a threshold for a level set of the response, and ask for gradients.
resp2 <- respInfo(pts, fnobj=fobj, lsetthresh=1.2, inclGrad=TRUE)
resp2
attr(resp2, "gradient")

# Box constraints on the feasible region can be specified directly in
# 'feasbnds', without the need for a separate feasibility function.
fobj <- fnObj(c("X1", "X2", "X3"), respfn=f, hasgrad=TRUE,
              feasbnds=rbind(rep(0, 3), rep(NA, 3)), powr=c(-1, 0, 2.5))
respInfo(pts, fnobj=fobj)

# Accessor functions:
inpDim(fobj)  # 3
inpNames(fobj)
inpTol(fobj)  # 2 x 3 matrix
respTol(fobj)  # 2-element vector
feasBnds(fobj)  # 2 x 3 matrix
hasGrad(fobj)
fnSpec(fobj)

lsetPkgOpt(oldopt)  # restore starting options
```

`fnObj.character` *Create an Object that Represents a Response Function*

Description

Create an object containing a response function and related information, including about its input space, feasible region, and gradient availability.

Usage

```
## S3 method for class 'character'
fnObj(x, respfn, feasfn=NULL, hasgrad=FALSE,
    derivmethod={ if (hasgrad) "gradient" else "none" }, warnNAresp=2,
    warnInfresp=1, feasbnds=matrix(NA_real_, nrow=2, ncol=length(x)),
    inptol=lsetPkgOpt("inptol"), resptol=lsetPkgOpt("resptol"), ...)
```

Arguments

`x`	Character vector of unique names for the dimensions of the input space. They must be valid names for R objects (see `base::make.names`). `inpNames()` will extract these names from an `fnObj` object.
`respfn`, `feasfn`, `hasgrad`, `derivmethod`, `warnNAresp`, `warnInfresp`	See the documentation for the generic.
`feasbnds`	Optional matrix or list specifying lower and upper bounds (box constraints) on the feasible region of the response function. Bounds are applied strictly, without any numerical tolerance from `inptol`. A matrix must have two rows and `d` columns, where `d` is the number of dimensions in the input space. The first row contains lower bounds and the second upper bounds. Alternatively, a list may be specified with named components for one or more of the input dimensions. Each component should be a two-element vector with lower and upper limits for the corresponding input. NA values are treated as `-Inf` for lower bounds and `Inf` for upper bounds. Defaults are `-Inf` for all lower bounds, `+Inf` for all upper bounds. Any bounds on coordinates specified in `feasbnds` are always merged with the contraints specified by the `feasfn` function, if present.
`inptol`	Two-row matrix with `d` columns. The first row contains relative tolerances for coordinate values for each dimension and the second row contains minimum absolute tolerances. Relative tolerances must be no greater than 0.001 and minimum absolute tolerances must be strictly positive. `inptol` may also be specified as a two-element vector, interpreted as a single column, to be repeated `d` times. A one-element vector is interpreted as a relative tolerance to be used for all dimensions, with the minimum absolute tolerance set to `sqrt(.Machine$double.eps)`. The default is `lsetPkgOpt("inptol")`.
`resptol`	Two-element vector with the relative and minimum absolute tolerances to use for response function values. Valid values are the same as for `inptol`. If only a single value is specified, it will be treated as the relative tolerance. The default is `lsetPkgOpt("resptol")`.
`...`	Additional arguments to be included in each call to `respfn` and `feasfn`. Must not include `inclGrad`.

Details

This method for the generic fnObj function constructs an fnObj object from scratch, starting with a vector of names for the input space dimensions. See the documentation of the generic for the purpose and details of fnObj objects.

Value

An object with S3 class fnObj.

See Also

fnSpec; lsetPkgOpt for more information about the tolerances specified by inptol and resptol.

Examples

```
# See the examples in '?fnObj'
```

---

fnObj.fnSpec *Create an Object that Represents a Response Function*

Description

Create an object containing a response function and related information, including about its input space, feasible region, and gradient availability.

Usage

```
## S3 method for class 'fnSpec'
fnObj(x, respfn, feasfn=NULL, hasgrad=FALSE,
    derivmethod={ if (hasgrad) "gradient" else "none" }, warnNAresp=2,
    warnInfresp=1, ...)
```

Arguments

x	fnSpec object containing basic information about the input space and tolerances.
respfn, feasfn, hasgrad, derivmethod, warnNAresp, warnInfresp	See the documentation for the generic.
...	Additional arguments to be included in each call to respfn and feasfn. Must not include inclGrad.

Details

This method for the generic fnObj function constructs an fnObj object based on the specifications in an fnSpec object. See the documentation of the generic for the purpose and details of fnObj objects.

Value

An object with S3 class fnObj.

See Also

fnSpec

---

fnSpec *Input Space and Related Specifications*

Description

Create an fnSpec object or extract one from another object. fnSpec objects contain a subset of the information in an fnObj object. For package users their main value is as a way to specify enough information to create inpRect and inpRays objects that are not tied to a specific response function.

Usage

```
fnSpec(x=NULL, inpnames, feasbnds=NULL, inptol=NULL, resptol=NULL)
```

Arguments

x	Optional object from which an fnSpec object will be extracted and returned. If specified, other arguments will be used to update the returned fnSpec. See DETAILS.
inpnames, feasbnds, inptol, resptol	See ?fnObj.character.

Details

An fnSpec object describes selected properties of a response function: the number and names of its input variables, limits on the ranges of those variables, and numerical tolerances associated with point coordinates and response values. It does not contain code to actually calculate response values at given input points; for that see fnObj.

If the first argument x is already an fnSpec object or contains one, the fnSpec object will be returned, updated with any of the other arguments that are not NULL. However the number and names of the input space dimensions (argument inpnames) cannot be updated.

As for fnObj objects, accessor functions can be used to extract individual specifications from an fnSpec object: inpDim(), inpNames(), feasBnds(), inpTol(), and respTol(). There is also a print method.

Value

An object with S3 class fnSpec.

See Also

fnObj

Examples

```
fspec <- fnSpec(inpnames=c("X1", "X2", "X3"))
# 'fspec' has default tolerances and other specifications.
inpTol(fspec)  # 2 x 3 matrix
respTol(fspec)  # 2-element vector
feasBnds(fspec)  # 2 x 3 matrix
# Other accessor functions:
inpDim(fspec)  # 3
inpNames(fspec)

# Create rays or hyper-rectangles independently of any specific response
# function in an 'fnObj' object.
rays <- axisRays(fspec)  # 3 rays, one along each coordinate axis
rect <- inpRect(rbind(c(0, 0, 0), c(1, 2, 3)), spec=fspec)

# Updating:
fspec2 <- fnSpec(fspec, feasbnds=list(X2=c(0, 10), X1=c(NA, 5), X3=c(NA, 6)),
                 resptol=c(1e-4, 2e-6))
feasBnds(fspec2)
respTol(fspec2)

# Extract an 'fnSpec' from another object.
identical(fnSpec(rays), fspec)
identical(fnSpec(rect), fspec)
```

---

hasGrad *Can a Response Function Calculate Gradient Vectors?*

Description

Inquire whether the response function in an fnObj object is able to calculate gradient vectors as well.

Usage

```
hasGrad(x)
```

Arguments

x	An fnObj object.

Value

Logical scalar (may be NA).

See Also

fnSpec, fnObj

Examples

```
# See examples in '?fnObj'
```

---

inpDim *Number of Dimensions for the Input Space*

Description

Return the number of dimensions for the input space associated with an object.

Usage

```
inpDim(x)
```

Arguments

x	An fnSpec object, or object containing one. (That includes fnObj, inpRays, inpRect, inpScale, respProfiles and lsetSegs objects.)

Value

Scalar, the number of dimensions of the input space associated with x.

See Also

fnObj, inpNames, fnSpec

Examples

```
# See examples in '?fnSpec'
```

inpLines *Lines in Input Space*

Description

Create an inpLines object, or extract one from another object. inpLines objects represent sets of 1-dimensional lines in input space. Each line is defined by a pair of distinct points, the first representing position 0 on the line, and the second position 1.0.

Usage

```
inpLines(x, ...)
```

Arguments

x	Either a matrix of point coordinates to be used to create a set of lines (see inpLines.default for details); or a respProfiles or lsetSegs object from which the associated set of lines or rays will be extracted.
...	Additional arguments for specific methods.

Details

This is an S3 generic function. There is a default method (to create a new set of lines), and methods to extract an existing set of lines from another object. (Package users should call the generic.)

inpLines class objects are represented internally as pairs of distinct points in input space, which can be extracted as matrices by function ptCoord. Lines can be given names via the names and names<- generic functions.

The class also has methods for length (the number of lines), [(selecting lines), and c (to combine sets of lines). Positions along a line (see inpRays) can be converted to point coordinates using function posnToCoord().

inpRays objects are a special case of inpLines, where all lines have the same position 0 point.

Value

An object with S3 class inpLines (and possibly also inpRays, if x is a respProfiles or lsetSegs object).

See Also

inpRays, inpLines.default, ptCoord, posnToCoord

Examples

```
fspec <- fnSpec(inpnames=c("X1", "X2", "X3"))
pts <- rbind(c(0, 1, 3), c(2, 2, 4), c(5, -1, 2), c(1, 1, 1))
# The default method of defining positions 0 and 1 for each line is
# to use consecutive pairs of rows of 'pts':
ilines <- inpLines(pts, spec=fspec)
ilines
length(ilines)  # 2, the number of lines

# Alternative way of specifying positions 1 and 0 is by two matrices.  The
# first has position 1 points, the second position 0 points:
ilines2 <- inpLines(pts[c(2, 4), ], x0=pts[c(1, 3), ], spec=fspec)
ilines2

# Extract coordinates of the defining points (positions 0 and 1) for
# each line.
ptCoord(ilines, which="paired")  # 4 rows:  first two for line 1, second two for line 2

# Set and retrieve names for lines.
names(ilines) <- c("foo", "bar")
names(ilines)

# Combine multiple 'inpLines' objects.
ilines3 <- inpLines(rbind(c(1, 2, 3), c(2, 3, 4)), spec=fspec)  # 1 line
ilines13 <- c(ilines, ilines3)
length(ilines13)  # 3 lines

# Select a subset of lines.
length(ilines13[2:3])  # 2
length(ilines13[0])  # 0 (still an 'inpLines' object, but empty)
ilines13[0]

# Rays are a special case of lines (they share the same position 0 point).
rays <- axisRays(fspec)  # 3 rays, one along each coordinate axis
inherits(rays, "inpLines")  # TRUE
```

---

inpLines.default *Lines in Input Space*

Description

Create an `inpLines` object from a matrix of point coordinates.

Usage

```
## Default S3 method:
inpLines(x, x0=NULL, pairidxs=NULL, spec, warnDegenerate=TRUE, ...)
```

Arguments

x	Matrix containing coordinates of points in input space. NA coordinates are not allowed.
x0	Optional matrix containing coordinates of points in input space. NA coordinates are not allowed. If specified it must have the same number of rows as x. Each row of x0 defines position 0 for a line and the corresponding row of x defines position 1.
pairidxs	Alternative to argument x0. Two-row matrix. Each column contains the indices of rows in x defining positions 0 and 1 for one line. If pairidxs has column names, they will be used to name the lines. If both x0 and pairidxs are NULL, the default for pairidxs is to use consecutive pairs of points in x to define lines.
spec	fnObj or fnSpec object with specifications for the input space.
warnDegenerate	Integer or logical scalar. If 1 or TRUE (the default), a warning will be raised if any lines are dropped because their position 0 and 1 points are equal, to within a numerical tolerance. If 0 or FALSE, degenerate lines are dropped silently. If 2, degenerate lines will be treated as an error.
...	Ignored, with a warning. (Only present for compatibility with the generic.)

Value

An object with S3 class inpLines. See the documentation for the generic function for details.

Examples

```
# See the examples in '?inpLines'
```

---

inpNames *Names of the Dimensions of the Input Space*

Description

Return the names of the input space dimensions associated with an object.

Usage

```
inpNames(x)
```

Arguments

x	An fnSpec object, or object containing one. (That includes fnObj, inpRays, inpRect, inpScale, respProfiles and lsetSegs objects.)

Value

Character vector. Dimension names are unique and are valid R object names.

See Also

fnObj, inpDim, fnSpec

Examples

```
# See examples in '?fnSpec'
```

---

inpRays *Rays in Input Space*

Description

Create an inpRays object, or extract one from another object. inpRays objects represent sets of 1-dimensional rays in input space, with all the rays sharing the same origin.

Usage

```
inpRays(x, ...)
```

Arguments

x	Either a matrix of point coordinates to be used to create a set of rays (see inpRays.default for details); or a respProfiles or lsetSegs object from which the associated set of rays will be be extracted.
...	Additional arguments for specific methods.

Details

This is an S3 generic function. There is a default method (to create a new set of rays), and methods to extract or update an existing set of rays. (Package users should call the generic however.)

A ray in input space is defined by two points: an origin and a second, distinct point that defines the direction of the ray. All rays in an inpRays object must have the same origin. inpRays objects are thus a special case of inpLines objects, in which all lines have the same position 0 point (the origin).

The second, direction-defining point represents position 1.0 for that ray. Other positions along a ray are defined by linear interpolation or extrapolation from those two points. Function posnToCoord() converts positions along rays to the corresponding point coordinates. The rayOrigin() function extracts the origin as a 1-row matrix, rayPosn1() extracts the position 1 points as a matrix, and rayVec() extracts the ray direction vectors.

inpRays objects optionally include an inpRect object (a hyper-rectangle in input space). If present, the hyper-rectangle must contain the ray origin, and the reference point of the hyper-rectangle will be set to that origin. Position 1.0 on each ray will be set to the ray's intersection with the boundary of the hyper-rectangle. The hyper-rectangle associated with an inpRays object can be extracted with the rayRect() function (which returns NULL if the object has no associated inpRect).

In order for a ray to be well-defined, its origin and position 1 points must be distinct. Testing for degenerate rays uses a numerical tolerance; i.e., it tests for practical degeneracy, rather than requiring exactly zero length. The tolerance is specified by an `fnSpec` specification object.

When a hyper-rectangle is specified and the ray origin is on its boundary, it is possible for a ray to lie entirely outside the hyper-rectangle except for its origin. Such rays will be dropped, because once their position 1 points are required to intersect the hyper-rectangle they become degenerate.

Since `inpRays` objects are also `inpLines` objects, they inherit the latter's methods for `length` (number of rays), `[` (selecting a subset of rays), and `names` and `names<-` (extract and set ray names). There are also `plot` and `update` methods for `inpRays` objects.

Value

An object with S3 class `inpRays`, inheriting from `inpLines`.

See Also

Methods `inpRays.default`, `update.inpRays`. `inpRect`, `inpLines`, `rayOrigin`, `rayPosn1`, `rayVec`, `posnToCoord`, `rayRect`, `plot.inpRays`, `update.inpRays`. `axisRays` and `randomRays` generate common types of rays.

Examples

```
oldopt <- lsetPkgOpt(reset=TRUE)  # default options

fspec <- fnSpec(inpnames=c("X1", "X2", "X3"))

# Rays may be defined by specifying their position 1 points and origin:
pts <- rbind(c(0, 1, 3), c(2, 2, 4), c(5, -1, 2))
orig <- c(0.2, 1.1, 5.5)
rays <- inpRays(pts, origin=orig, spec=fspec)
# Or by specifying their direction vectors and origin:
vecs <- sweep(pts, 2, orig, "-")  # equivalent to 'rayVec(rays)'
rays2 <- inpRays(vecs=vecs, origin=orig, spec=fspec)
all.equal(rays, rays2)  # TRUE

# Methods and accessor functions:
length(rays)  # 3
rayOrigin(rays)  # 1 x 3 matrix
rayPosn1(rays)  # 3 row matrix
rayVec(rays)  # 3 row matrix
plot(rays, dims=c(1, 3))  # rays projected to X1, X3 plane

# Quick generation of common types of rays:
axisRays(spec=fspec, degree=2, origin=orig)
set.seed(1)
randomRays(n=5, spec=fspec, origin=orig)

#-- Ray names:
rownames(pts) <- paste0("ray", seq_len(nrow(pts)))
rays3 <- inpRays(pts, spec=fspec, origin=orig)
names(rays3)  # taken from row names of 'pts'
```

```
names(rays3) <- LETTERS[1:3]
names(rays3)  # changed

#-- Selecting subsets of rays.
rays23 <- rays[2:3]
length(rays23)  # 2

#-- Empty 'inpRays' objects are allowed:
rays0 <- inpRays(spec=fspec, origin=orig)  # no rays, but origin still defined
length(rays0)  # 0
identical(rays0, rays[0])  # TRUE

#-- Ray position 1 points can be (re)defined by each ray's intersection
# with the boundary of a hyper-rectangle:
rect1 <- inpRect(rbind(c(0,0,0), c(1,1,1)), spec=fspec)
  # (default reference pt is center: c(0.5, 0.5, 0.5))
vecs1 <- rbind(c(-1,-1,0), c(0,0,1), c(0, -1, 1))
rays1 <- inpRays(vecs=vecs1,
                 rect=rect1)  # default 'origin' and 'spec' taken from 'rect'
# Position 1 points for all rays are on the boundary of 'rect1':
print(rays1, vecForm=FALSE)

# Extract the hyper-rectangle associated with rays (which might be NULL).
rayRect(rays1)  # 'inpRect' object

# An explicit 'origin' argument takes precedence over 'rect':  the hyper-
# rectangle must include 'origin', and the copy of the rectangle stored
# within the 'inpRays' object has its reference point set to the ray origin.
rays1 <- inpRays(vecs=vecs1, origin=c(1, 1, 0.5), rect=rect1)
print(rays1, vecForm=FALSE)
all(rectRefPt(rayRect(rays1)) == rayOrigin(rays1))  # TRUE

# An existing 'inpRays' object can be updated with a new origin and/or
# hyper-rectangle (the updated rectangle must contain the updated origin).
rect2 <- inpRect(rbind(c(-1, -2, -3), c(1, 2, 3)), spec=fspec)
rays2 <- update(rays1, origin=c(0, -1, 2), rect=rect2)
rayRect(rays2)  # now set to 'rect2'
rayOrigin(rays2)  # updated

#-- Degenerate rays are not allowed, and will be dropped.
vecs <- rbind(c(-1,-1,0), c(0,0,0), c(0, -1, 1))  # 2nd vector has 0 length
rays <- inpRays(vecs=vecs, origin=orig, spec=fspec,
                warnDegenerate=FALSE)  # turns off default warning/error
length(rays)  # 2, not 3

# One can also end up with a degenerate ray if a rect is specified and
# its reference point is on the boundary.  In that case a ray's only
# intersection with the rect boundary may be the origin:
rect <- inpRect(rbind(c(0,0,0), c(1,1,1)), spec=fspec)
orig <- c(1,1,0.5)  # on the boundary of 'rect'
vecs <- rbind(c(-1,-1,0), c(0,1,1), c(0, -1, 1))
  # The second direction vector points away from the interior of 'rect',
  # relative to 'orig'.
```

```
rays <- inpRays(vecs=vecs, rect=rect, spec=fspec,
                origin=c(1,1,0.5),  # on the boundary of 'rect'
                warnDegenerate=FALSE)
    # (Second ray's only intersection with 'rect' is the origin, so dropped.)
length(rays)  # 2, not 3

lsetPkgOpt(oldopt)  # restore starting options
```

`inpRays.default` *Rays in Input Space*

Description

Create an `inpRays` object directly from point coordinates in a matrix. Can also be used to create an empty `inpRays` object.

Usage

```
## Default S3 method:
inpRays(x, vecs=NULL, origin=NULL, rect=NULL, spec=NULL,
    warnDegenerate=TRUE, ...)
```

Arguments

`x`	Matrix, each row giving the coordinates of a point in input space. Rays are defined as the lines starting at `origin` and passing through these points, so the number of rays is the number of points in `x`. Degenerate rays are not allowed: points that are numerically (nearly) equal to origin will be dropped. NA coordinates are not allowed.
`vecs`	Matrix, each row specifying a vector in input space. This is an alternative to `x` as a way of defining rays: Rays start at `origin` and extend in the directions specified by the vectors in `vecs`. (This is equivalent to specifying an `x` with its `i`-th row equal to `vecs[i, ] + origin`.) This argument is ignored if `x` is specified.
`origin`	Vector or 1-row matrix specifying a single point in the input space. All rays start from this point (i.e., it is position 0 for all rays). The default is the reference point of `rect` if that is specified, otherwise it is the point with all coordinates 0.
`rect`	Optional `inpRect` object, representing a hyper-rectangle in input space. If specified, the hyper-rectangle must include the point `origin`.
`spec`	`fnObj` or `fnSpec` object with specifications for the input space. Optional if `rect` is specified, since that contains the input space information.
`warnDegenerate`	Integer or logical scalar. If 1 or TRUE (the default), a warning will be raised if any rays are dropped because their position 1.0 point is equal to `origin`, to within a numerical tolerance. If 0 or FALSE, degenerate rays are dropped silently. If 2, degenerate rays will be treated as an error.
`...`	Ignored, with a warning. (Only present for compatibility with the generic.)

Details

If neither x nor vecs is specified, the result will be an inpRays object with no rays but with a defined origin.

origin defines position 0 along each ray, and the points specified by x (directly, or indirectly via vecs) define position 1 along each ray.

Value

An object with S3 class inpRays, inheriting from inpLines. Ray names are taken from the row names of x or vecs, if any.

Examples

```
# See the examples in '?inpRays'
```

---

inpRect *Hyper-Rectangle in Input Space*

Description

Create an inpRect object, representing a finite, axis-aligned, hyper-rectangle in the input space of a response function.

Usage

```
inpRect(x, refpt=NULL, spec)
```

Arguments

x	A two-row matrix. The first row contains the coordinate vector of the lower corner of the hyper-rectangle, and the second the coordinate vector of the upper corner.
refpt	Optional vector or one-row matrix containing the coordinates of a single point. The point must be in the hyper-rectangle (see DETAILS). It can be used to indicate a reference or special point of the rectangle. If not specified, the center of the rectangle will be used.
spec	fnObj or fnSpec object with specifications for the input space.

Details

All coordinates in x must be non-NA and finite, and the hyper-rectangle must have non-zero volume (see below).

If refpt is specified it will be checked to make sure it belongs to the hyper-rectangle. If it is slightly outside, to within tolerances specified by spec, then x will be adjusted to put refpt exactly on its boundary.

Positive volume is defined as follows: For each dimension `j`, absolute tolerances for the lower and upper limits are calculated as `atol[, j] <- absTol(x[, j], inptol=inpTol(spec)[, j])`. The separation between `x[1, j]` and `x[2, j]` is considered non-zero if `x[2, j] - x[1, j]` is strictly greater than `1.5*(atol[2] + atol[1])`. The hyper-rectangle is considered to have positive volume only if this separation is non-zero for all dimensions.

The corners of an `inpRect` object can be extracted with function `rectCorners()`, the reference point with `rectRefPt()`, and the lengths of the sides with `rectSize()`.

There are `plot`, `print`, and `update` methods for this class.

Value

An object with S3 class `inpRect`.

See Also

`rectCorners`, `rectRefPt`, `rectSize`, `absTol`

Examples

```
oldopt <- lsetPkgOpt(reset=TRUE)  # default settings

fspec <- fnSpec(inpnames=c("X1", "X2", "X3"))
# Directly create a hyper-rectangle:
rect1 <- inpRect(rbind(c(0, -1.1, -4), c(2, -0.5, 1000)), spec=fspec)
rect1

# Accessor functions.
rectCorners(rect1)
rectRefPt(rect1)
rectSize(rect1)

# Non-default reference point:
rect2 <- inpRect(rbind(c(0, 1, 1), c(0.5, 3, 6)), refpt=c(0, 1, 4),
                 spec=fspec)
rectRefPt(rect2)
# Update reference point:
rect2b <- update(rect2, refpt=c(0.2, 3, 3))
rectRefPt(rect2b)

# The reference point must belong to the rectangle.  However if it is only
# slightly outside (within a numerical tolerance) the rectangle will
# be adjusted to accomodate it.
fspec <- fnSpec(inpnames=c("X1", "X2", "X3"),
                inptol=c(0, 1e-3))  # defines "slightly"
rect3 <- inpRect(rectCorners(rect2), spec=fspec)
update(rect3, refpt=c(0.2, 3 + 5e-4, 3))  # slightly bigger, ref pt on boundary

lsetPkgOpt(oldopt)  # restore starting settings
```

inpScale *Scaling Information for Dimensions of Input Space*

Description

Create an inpScale object, containing scaling information for the dimensions of the input space of a response function.

Usage

```
inpScale(x=rep(1, d), spec)
```

Arguments

x	Vector or matrix. If a vector, its length must equal the number of dimensions of the input space, and all values must be strictly positive and finite. A matrix must be square, symmetric, and positive-definite, with dimensions matching the input space. See DETAILS.
spec	fnObj or fnSpec object with specifications for the input space.

Details

When dimensions of the input space have different units; or point coordinates for different dimensions have different magnitudes; or the level set of interest is elongated in certain directions, it may be useful to rescale and/or weight the dimensions when calculating distances or angles in input space. inpScale objects provide a way to specify such scaling.

The simplest case is when x is a vector of scaling factors, one scale factor per input space dimension. For example if the input space has three dimensions X1, X2, and X3, and in the region of interest their typical magnitudes are 2,000, 0.003, and 5, then set x to c(2e3, 3e-3, 5). The goal is that dividing coordinates by their scaling factors should standarize them: coordinates for different dimensions should be roughly similar in magnitude, preferably with a magnitude close to 1.

A matrix x generalizes this by effectively allowing a rotation of the coordinate axes before applying scaling. Details are given below, in terms of the components of the returned object. Note that specifying a vector x is equivalent to a matrix x where the diagonal contains the _squares_ of the vector values, and off-diagonal elements are 0.

In the following, let V denote the matrix specified by x (either x itself, or diag(x^2) if x is a vector).

Weighted distance in input space. The weighted distance between points u and v in input space is defined as the square root of the quadratic form t(u - v) %*% V^(-1) %*% (u - v), where V^(-1) is the inverse of V. Defining square matrix W so that t(W) %*% W = V^(-1), the weighted distance between u and v is the ordinary unweighted Euclidean distance between vectors W %*% u and W %*% v.

The analogous statistical concept is Mahalanobis distance, where V represents a covariance matrix.

Other interpretations of inpScale *objects.* A related use for inpScale objects is to convert back and forth between point coordinates in input space, and coordinates in a more generic, standardized space where all dimensions and directions are considered equivalent. If y represents a point in input

space and z represents the same point in standardized space, the transformations are: z = W %*% y and y = W^(-1) %*% z.

For example, points that lie on the ellipsoid t(y) %*% V^(-1) %*% y = 1 in input space can be transformed to points that lie on the unit hypersphere t(z) %*% z = 1 in standardized space, and vice versa.

Yet another way to think about an inpScale object is that it defines a new coordinate system in input space, with axes given by the columns of W^(-1). For a given point y in the original coordinate system, its coordinates in the new system are z = W %*% y.

Note that coordinate vectors in this package are stored as rows of d-column matrices, rather than as individual column vectors. So the way to apply transformation matrix W to a set of points in n-by-d matrix X is tcrossprod(X, W), yielding another n-by-d matrix, and analogously for W^(-1).

Construction of transformation matrices. In this function W is constructed from the spectral decomposition of V: W = diag(1 / sqrt(lambda)) %*% t(P), where P is a matrix whose columns are the eigenvectors of V and lambda is the vector of eigenvalues of V. Thus W^(-1) = P %*% diag(sqrt(lambda)). However in the special case when V is diagonal, the columns of P (which are just permutations of c(1, 0, ..., 0)) and elements of lambda (which is just w^2) are sorted to match the dimensions of input space, rather than sorted by decreasing lambda. This preserves a correspondence between individual dimensions in input and standardized spaces, when standardization only involves scaling, not rotation.

The sign of eigenvectors is arbitrary. To increase consistency of the returned object, the sign of the i-th column of P is chosen so that its i-th element is positive, provided the latter is not negligible in magnitude (i.e., it is greater than 1/(100*sqrt(d))). Otherwise the sign is chosen so that the element with the largest magnitude is positive.

Value

An object of S3 class inpScale. This is a list with components:

V	Square, symmetric, positive definite matrix equal to argument x if that was a matrix, or to diag(x^2) if it was a vector.
w	Vector, the reciprocals of the square root of the diagonal of V.
W	Square matrix, a square root of the inverse of V: t(W) %*% W = V^(-1).
Winv	Square matrix, the inverse of W, and thus a square root of V: Winv %*% t(Winv) = V.
V_is_diag	Logical scalar, TRUE if V is diagonal.
fnSpec	fnSpec object containing specifications for the input space.

See Also

fnSpec. Examples for randomRays show effects of specifying various inpScale objects.

Examples

```
oldopt <- lsetPkgOpt(reset=TRUE)  # default settings

dat <- data.matrix(iris[, 1:4])  # 150 points in 4D space
fspec <- fnSpec(inpnames=colnames(dat))
```

```
covmat <- cov(dat)
sds <- sqrt(diag(covmat))
cormat <- cor(dat)

# Mahalanobis distances of points from 0-origin.
mdist <- apply(dat, 1, function(y) { sqrt( t(y) %*% solve(covmat) %*% y) })
# 'inpScale' objects include a matrix to transform point coordinates so that
# Euclidean distances for transformed points equal Mahalanobis distances
# for original points.
scl <- inpScale(covmat, spec=fspec)
tdat <- tcrossprod(dat, scl$W)  # Transformed points
edist <- sqrt(rowSums(tdat^2))  # Euclidean dist using transformed vectors
all.equal(mdist, edist)  # TRUE

# Coordinate system rescaling/rotation.
chk1 <- sweep(dat, 2, scl$w, "*")  # re-scale each dimension to unit SD
all.equal(cov(chk1), cormat)  # TRUE
chk2 <- tcrossprod(dat, scl$W)  # both re-scale and orthogonalize
all.equal(cov(chk2), diag(4))  # TRUE
# Reverse the transformation:
z_to_y <- scl$Winv
chk3 <- tcrossprod(chk2, z_to_y)
dimnames(chk3) <- dimnames(dat)
all.equal(chk3, dat)  # TRUE

# Default 'inpScale' contains only identity transformations ('w', a vector
# of 1's, 'V' the identity matrix):
scl <- inpScale(spec=fspec)
scl$w
scl$V
scl$V_is_diag

#----- 2D example of change of coordinate system.
fspec <- fnSpec(inpnames=c("X1", "X2"))
scl <- inpScale(matrix(c(5, 2, 2, 1), nrow=2), spec=fspec)
W <- scl$W
Winv <- scl$Winv
# Point in original coordinate system:
y <- c(3, 1)
plot(c(-1, 5), c(-1, 5), type="n", axes=FALSE, xlab="X1", ylab="X2")
axis(side=1, pos=0)
axis(side=2, pos=0)
points(x=y[1], y=y[2])
# Draw axes of new coordinate system, with tick marks ('x') at unit intervals.
ax1units <- rbind(c(1, 0), c(2, 0), c(3, 0))
ax2units <- rbind(c(0, -1), c(0, 1), c(0, 2), c(0, 3), c(0, 4))
abline(a=0, b=Winv[2, 1] / Winv[1, 1], col="orchid1")
points(ax1units %*% t(Winv), pch=4, col="orchid1")
abline(a=0, b=Winv[2, 2] / Winv[1, 2], col="orchid1")
points(ax2units %*% t(Winv), pch=4, col="orchid1")
# Same point in new coordinate system:
z <- W %*% y
cat("Point coordinates in new system= ", toString(round(z, 3)), "\n")
```

```
# Projection of the point onto the new axes (expressed in the original
# coordinates, because that's what the plot uses):
y_z1 <- Winv %*% c(z[1], 0)
y_z2 <- Winv %*% c(0, z[2])
segments(x0=y[1], y0=y[2], x1=c(y_z1[1], y_z2[1]), y=c(y_z1[2], y_z2[2]),
         lty=3, col="orchid1")

lsetPkgOpt(oldopt)  # restore starting settings
```

`inpTol` *Numerical Tolerance Specifications for the Input Space*

Description

Return tolerance specifications for each dimension of the input space associated with an object.

Usage

```
inpTol(x)
```

Arguments

`x`	An `fnSpec` object, or object containing one. (That includes `fnObj`, `inpRays`, `inpRect`, `inpScale`, `respProfiles` and `lsetSegs` objects.)

Value

Two-row matrix with `d` columns, where `d` is the number of dimensions in the input space. The first row contains a relative tolerance for each dimension and the second row contains a minimum absolute tolerance.

See Also

`lsetPkgOpt` and `absTol` describe how tolerances are specified in this package. `fnSpec`, `fnObj.character`

Examples

```
# See examples in '?fnSpec'
```

lsetPkgOpt *Extract or Set Options for the 'levelSets' Package*

Description

Extract or set user-level options for the `levelSets` package.

Usage

```
lsetPkgOpt(..., reset=FALSE)
```

Arguments

`...`	Either a single unnamed character vector, or one or more `name=value` pairs, or a single unnamed list containing `name=value` pairs.
`reset`	Logical. If TRUE, options are reset to their `factory-fresh` default values, and `...` is ignored. It has no effect if FALSE.

Details

The available options, with their `factory-fresh` defaults are:

resptol, inptol Tolerances to use when deciding whether two response values, or coordinates for two points in input space, should be treated as equal or equivalent. Tolerances are necessary to allow for representation and roundoff errors in floating point calculations. In addition, for most applications there are limitations on measurement, data, or model accuracy which are even larger than floating point errors. These mean that small numerical differences are not meaningful in practice. Therefore tolerances larger than those required just by floating point errors are usually appropriate.

Values for these options are handled like the `tol` argument to `absTol()`, so are two-element vectors in general. The first value specifies a *relative* tolerance (i.e., as a proportion of the magnitude of a number), and the second specifies a minimum absolute tolerance (constant regardless of the magnitude of a number). If only a single value is specified it is used as the relative tolerance, and the minimum absolute tolerance is set to `sqrt(.Machine$double.eps)`. Relative tolerances must be no larger than `1e-3` (implying at least three significant digits in response or point coordinate values). The minimum absolute tolerance must be strictly greater than 0.

The default for `resptol` is `c(1e-5,sqrt(.Machine$double.eps))`. For `inptol` it is `c(1e-6,sqrt(.Machine$double`

warn.showBadRespPts Non-negative integer. The maximum number of feasible points with NA or infinite response values for which coordinates will be printed, as part of warnings/errors. (See `?fnObj`.) Default is 3.

warn.invalidProfile Non-negative integer. Whether to generate warnings about response profiles along a line/ray that show an unexpected pattern of level set boundary crossings. (For example, two consecutive boundary crossings where the response trend is increasing for both or decreasing for both–such a pattern suggests there should be an intermediate crossing with the opposite trend.) Such warnings depend on derivative/ trend information, so will not be generated if `derivMethod` is `none`. A value of 0 suppresses the warning, 1 generates just the warning, and 2 also prints the problematic profile. Default is 1.

Value

If reset is TRUE, the whole list of previous (before the reset) option values is returned, invisibly.

Otherwise the return value depends on the form of .... If the first argument does not have a name and is a plain character vector then the rest of ... is ignored and a list containing the values of the options named by the vector is returned. However, if the vector contains only a single option name, the actual value of the option is returned (not wrapped in a list of length 1).

If the first argument does not have a name and is a list with named components then the rest of ... is ignored and the list components are used to set the values of the named options. A list of the previous values of those options is returned, invisibly.

If ... is a series of name=value pairs, they are treated as if they were a single list with named components.

In all cases it is an error to request or set the value of an unknown package option.

lsetPkgOpt() returns a list with current values of all options.

Examples

```
oldopt <- lsetPkgOpt(reset=TRUE)
# Default 'factory-fresh' options:
defopt <- lsetPkgOpt()
str(defopt)

# Single option requested:  returns plain vector, not list
tol1 <- lsetPkgOpt("resptol")
oldtol1 <- lsetPkgOpt(resptol=1e-4)  # returns old value as list of length 1
identical(tol1, oldtol1[[1]])

# Two or more options requested:  list is returned
opts <- lsetPkgOpt(c("warn.invalidProfile", "warn.showBadRespPts"))
oldopts <- lsetPkgOpt(warn.invalidProfile=0, warn.showBadRespPts=10)
identical(opts, oldopts)  # both lists with two components

lsetPkgOpt(reset=TRUE)  # restores defaults
identical(lsetPkgOpt(), defopt)  # TRUE

# Restore starting options.
lsetPkgOpt(oldopt)
```

---

lsetSegs *Line Segments in a Level Set of a Response Function*

Description

Create an lsetSegs object, or extract one from another object. lsetSegs objects contain 1-dimensional line segments in the input space of a response function. Each segment is defined by a pair of distinct points, the segment endpoints, that belong to the level set, with the assumption that every point on the segment connecting them also belongs to the level set.

Usage

```
lsetSegs(x, ...)
```

Arguments

x	Object containing information from which an lsetSegs object is to be created or extracted. Valid x include the return values from functions respProfiles, bdryFromRays, bdrySearch, and lsetSegsCheck.
...	Additional arguments for internal methods only.

Details

An lsetSegs object contains pairs of points that represent endpoints of line segments in input space. The restrictions on the endpoints are that (a) all endpoints belong to the level set; (b) the endpoints in a pair are distinct points; (c) it can be assumed that all the points on the line segment connecting a pair also belong to the level set; and (d) each pair of endpoints lies along a line that is explicitly represented in the object (by an inpLines or inpRays object).

Requirement (c) follows trivially from (a) if the level set is a single, convex region of input space. However if the level set may be non-convex or has disconnected parts, it is up to the function that creates the object to ensure that the assumption is reasonable.

Function inpLines() extracts the input space lines or rays from requirement (d). A given line/ray may have 0, 1, or >1 level set segments in the lsetSegs object.

Requirement (b) for distinct points means segment endpoints must be separated by at least an amount calculated from input tolerances in the response function specification. (See ?lsetPkgOpt and ?fnObj.character).

Function respInfo() applied to an lsetSegs object returns a data frame with response values and other information about the segment *endpoints*. Function segInfo() applied to an lsetSegs object returns a data frame with one row per *segment*. See their documentation for the columns included. Note that column line indexes the line (in the object returned by applying inpLines() to the lsetSegs object) along which the endpoint or segment lies.

The class has methods for length (the number of segments, which might be 0), [(selecting segments), c (to combine sets of segments), summary, plot, and pairs.

Extraction from other objects. This function will extract the lsetSegs object contained in the return value from functions bdryFromRays, bdrySearch, and lsetSegsCheck.

If x is a respProfiles object, this function will create an lsetSegs object from it, using the threshold specified when x was created (which must not be NA). Note that this function does not actually search for level set boundary points (see the functions mentioned in the previous paragraph for that). It only examines the existing discrete set of profile points in x.

A profile point is considered to belong to the level set if its lset value (from respInfo(x)) is TRUE. A level set segment consists of a run of consecutive profile points that all belong to the level set, and is represented by the first and last points in the run. Runs consisting of a single point are dropped (since segment endpoints must be distinct.) A given profile may have zero, one, or more than one level set segment.

A segment is considered uncensored at one of its endpoints if either lsetbdry or feasbdry for the point is TRUE, and censored otherwise. Therefore the censoring indicator lsetcens for segment endpoints will never be NA.

Value

An object with S3 classes `c("lsetSegs", "respProfiles")`.

See Also

`lsetSegs` objects are created by functions `bdryFromRays` and `bdrySearch` and their sliced versions, and by `lsetSegsCheck`. Useful accessor functions and methods are: `segInfo`, `respInfo`, `summary.lsetSegs`, `plot.lsetSegs`, `pairs.lsetSegs`. `lsetThresh` extracts the response value that defines the level set, and `fnSpec` extracts information about the response function and its input space.

Examples

```
# see examples in '?bdryFromRays', '?respInfo', '?segInfo'
```

---

lsetSegsCheck *Check the Validity of Level Set Line Segments*

Description

Check the validity of level set line segments in an `lsetSegs` object. A segment is valid if every point along it belongs to the level set; this function checks at a series of user-specified positions. Optionally attempt to fix invalid segments by replacing them with one or more sub-segments that are valid. Also check the consistency of multiple segments along the same line/ray.

Usage

```
lsetSegsCheck(x, fnobj, posns, fixCycles=1, resptol=respTol(x))
```

Arguments

`x`	An `lsetSegs` object.
`fnobj`	`fnObj` object containing the response function and associated information.
`posns`	Vector of values in the interval [0, 1] specifying positions along each segment in `x`, relative to its endpoints. The response function will be evaluated at each position to check whether the point belongs to the level set.
`fixCycles`	Non-negative integer. If greater than 0 an attempt will be made to fix invalid segments by finding valid subintervals. Those subintervals will themselves be checked and fixed, using up to `fixCycles - 1` additional cycles.
`resptol`	Optional vector of length 2, specifying tolerance parameters for the function response. The default is taken from `fnSpec(x)`. See `?fnObj.character`.

Details

The result may still contain invalid level set segments (those unable to be fixed after `fixCycles` attempts). They can be removed by subseting the `segs` component by the `validOut` component.

Note that the presence of invalid segments implies the level set is non-convex or has disconnected parts along the lines containing those segments.

If `x` contains any lines/rays with more than one segment, there will be a check whether the segments are overlapping or contiguous, and if so, whether they are consistent and can be merged:

Contiguous or overlapping segments on a line are consistent if all segment endpoints in the interior of the merged segment are censored. They are merged into a single segment. Uncensored interior endpoints indicate that the individual searches that produced them probably missed some intermediate boundary crossings. (For example, `initPosns` was too coarse relative to fluctuations in the response function.) Inconsistent pairs of segments are identified in component `inconOut` but are not merged.

Value

A list with components:

`validIn`	Logical vector with one element per segment in `x`. TRUE means all evaluated positions along the segment belonged to the level set (to within a numerical tolerance specified by `resptol`). FALSE means at least one evaluated position along the segment did not belong to the level set, including being an infeasible point.
`validOut`	Logical vector similar to `validIn`, but applying to the (possibly fixed or merged) segments returned in component `segs`.
`inconOut`	Two-column matrix. Each row contains the indices of a pair of segments in `segs` that appear to be inconsistent (see DETAILS).
`segs`	`lsetSegs` object containing segments after any attempts to merge them or fix invalid segments. (The presence of FALSE values in `validOut` indicates the attempts were not completely successful.)
`resp`	Matrix of response values at each position (column) on each segment (row) of `segs`.

See Also

`lsetSegs`

Examples

```
oldopt <- lsetPkgOpt(reset=TRUE)

Ex <- dbl_ellipse_X1bnd_2dEx  # see '?dbl_ellipse_2dEx'
fobj <- Ex$fobj
segs <- Ex$segs
rect0 <- Ex$rect0

plot(segs, rect=rect0, main="Original segments from 'bdryFromRays()'")
# Some segments cross the gap between parts of the level set, making them
```

```
# invalid:
chk <- lsetSegsCheck(segs, fnobj=fobj, posns=(1:4)/5)
table(chk$validIn)  # 8 invalid segments ...
table(chk$validOut)  # all appear to have been fixed by splitting into
                     # two valid segments
segs_fixed <- chk$segs  # could also use 'lsetSegs(chk)'
plot(segs_fixed, rect=rect0,
     main="Segments after processing by 'lsetSegsCheck()'")

lsetPkgOpt(oldopt)
```

`lsetThresh` *Response Function Value that Defines a Level Set*

Description

Extract the response function value that defines a particular level set, from a `respProfiles` or `lsetSegs` object.

Usage

```
lsetThresh(x)
```

Arguments

`x`	An object inheriting from class `respProfiles` (including `lsetSegs`), or NULL.

Value

Numeric scalar, possibly NA (including if `x` is NULL).

See Also

`respProfiles`, `lsetSegs`

Examples

```
Ex <- dbl_ellipse_X1bnd_2dEx  # see '?dbl_ellipse_2dEx'
segs <- Ex$segs  # 'lsetSegs' object
lsetThresh(segs)

# See '?respProfiles' for other examples
```

pairs.lsetSegs *'pairs' Method for Plotting 'lsetSegs' Objects*

Description

pairs method for lsetSegs objects. Base graphics are used to plot line segments in a level set, projected onto pairs of input space dimensions.

Usage

```
## S3 method for class 'lsetSegs'
pairs(x, dims=NULL, rect=NULL, pt0=NULL, ...)
```

Arguments

x	An lsetSegs object.
dims	Numeric or character vector indicating which of the input space dimensions are to be displayed. The plot will have a panel for each pair of the listed dimensions. The default is to use all input space dimensions.
rect, pt0, ...	Additional arguments passed to the plot method for lsetSegs objects; see the documentation for that method. Must not include oneD or add, which are set internally by this function.

Details

Projection of a point onto the plane spanned by a pair of coordinate axes just amounts to using the point's coordinates for those dimensions. Coordinates for other dimensions are ignored.

Value

NULL, invisibly.

See Also

lsetSegs, plot.lsetSegs

Examples

```
# Example with 3-dimensional input space.
Ex <- circuitFailure_3dEx  # see '?circuitFailure_3dEx'
bsrch <- Ex$bsrch  # list as returned by 'bdrySearch()'

segs <- lsetSegs(bsrch)  # 'lsetSegs' object, containing level set segments
pairs(segs, rect=bsrch$rect,
      main="95% confidence region, circuit failure example")
```

plot.inpRays *Plot Method for 'inpRays' Objects*

Description

Plot method for `inpRays` objects. Base graphics are used to display the projection of rays onto the plane defined by a pair of input space dimensions.

Usage

```
## S3 method for class 'inpRays'
plot(x, y=NULL, dims=1:2, showRect=TRUE, eqAxes=FALSE, rayPar=list(),
    add=FALSE, ...)
```

Arguments

`x`	An object inheriting from class `inpRays`, containing rays in an input space with at least two dimensions.
`y`	Ignored. (Only present for compatibility with the `plot` generic.)
`dims`	Numeric or character vector specifying two input space dimensions. The rays are projected onto the plane spanned by their coordinate axes.
`showRect`	Logical scalar. If TRUE and `x` includes a hyper-rectangle object, the projection of the edges of the hyper-rectangle onto `dims` are displayed.
`eqAxes`	Logical scalar. If TRUE, equal axes are used for both dimensions.
`rayPar`	Optional list with graphics parameters used to draw the rays. Available parameters are arguments to `graphics::arrows`: `length`, `angle`, `code`, `col`, `lty`, `lwd`.
`add`	Logical scalar. If TRUE, the display is added to the current plot rather than starting a new plot.
`...`	Additional arguments passed to `plot.default`. Must not include `type` or `pty`, which are set internally.

Details

Rays are shown as arrows extending from their origin to position 1.0 for each ray.

Projection of a point onto the plane spanned by a pair of input space dimensions just amounts to using the point's coordinates for those dimensions. Coordinates for other dimensions are ignored.

Value

NULL, invisibly.

See Also

`inpRays`, `rayOrigin`, `rayPosn1`, `rayRect`, `graphics::arrows`

Examples

```
fspec <- fnSpec(inpnames=c("X1", "X2", "X3"))

# Rays can be defined by specifying their position 1 points and origin:
pts <- rbind(c(0, 1, 3), c(2, 2, 4), c(5, -1, 2))
orig <- c(0, 1, 2)
rays <- inpRays(pts, origin=orig, spec=fspec)
names(rays) <- paste0("Ray", 1:3)
plot(rays, dims=c("X2", "X3"), eqAxes=TRUE, rayPar=list(col="blue"))
```

---

plot.inpRect *Plot Method for 'inpRect' Objects*

Description

Plot method for `inpRect` objects. Base graphics are used to display the border of the hyper-rectangle in the plane defined by a pair of input space dimensions.

Usage

```
## S3 method for class 'inpRect'
plot(x, y=NULL, dims=1:2, eqAxes=FALSE, showRefPt=NULL, lty=NULL,
    lwd=NULL, border=NULL, add=FALSE, ...)
```

Arguments

`x`	An `inpRect` object, representing a hyper-rectangle in an input space with at least two dimensions.
`y`	Ignored. (Only present for compatibility with the `plot` generic.)
`dims`	Numeric or character vector specifying two input space dimensions. The border of `x` is projected onto the plane spanned by their coordinate axes.
`eqAxes`	Logical scalar. If TRUE, equal axes are used for both dimensions.
`showRefPt`	Logical scalar. If TRUE the reference point of `x` will be plotted as an "O" character. Default depends on `add`.
`lty`, `lwd`, `border`	Graphical parameters used to display the border of `x`. Defaults depend on `add`.
`add`	Logical scalar. If TRUE, the display is added to the current plot rather than starting a new plot.
`...`	Additional arguments passed to `plot.default` when `add` is FALSE. Must not include `type` or `pty`, which are set internally.

Details

Defaults for display properties depend on add. If add is FALSE they are `showRefPt=TRUE, lty=1, lwd=par("lwd"),border=par("fg")`. If add is TRUE they are `showRefPt=FALSE,lty=2, lwd=par("lwd"), border="lightgray"`.

Projection of a hyper-rectangle onto the plane spanned by a pair of input space dimensions just amounts to using the its coordinates for those dimensions. Coordinates for other dimensions are ignored.

Value

NULL, invisibly.

See Also

inpRect, rectCorners, rectRefPt

Examples

```
fspec <- fnSpec(inpnames=c("X1", "X2", "X3"))
rect1 <- inpRect(rbind(c(0, 0, 1), c(1, 2, 4)), spec=fspec)
plot(rect1, dims=2:3)
plot(rect1, dims=2:3, eqAxes=TRUE)

# Add a 'rect' to an existing plot.  (Uses different default lty and color)
set.seed(1)
rays1 <- randomRays(25, origin=c(1, 1, 2), spec=fspec)
plot(rays1, dims=c("X2", "X3"), eqAxes=TRUE,
     main="Random 3D rays projected to 2D")
plot(rect1, dims=c("X2", "X3"), add=TRUE)

# Note that this is different from including the rectangle in the definition
# of the rays:
set.seed(1)
rays2 <- randomRays(25, origin=c(1, 1, 2), rect=rect1, spec=fspec)
plot(rays2, dims=c("X2", "X3"), eqAxes=TRUE,
     main="Random 3D rays with hyper-rect set, projected to 2D")
```

---

plot.lsetSegs *Plot Method for 'lsetSegs' Objects*

Description

Plot method for `lsetSegs` objects. Base graphics are used to display the projection of line segments in a level set onto individual coordinate axes of the input space, or onto the plane defined by a pair of input dimensions.

Usage

```
## S3 method for class 'lsetSegs'
plot(x, y=NULL, dims=y, oneD=NULL, showSegs=TRUE, rect=NULL,
    pt0=NULL, add=FALSE, ...)
```

Arguments

x	An lsetSegs object.
y	Ignored. (Only present for compatibility with the plot generic.)
dims	Numeric or character vector specifying the input space dimensions onto which the projection of the line segments are to be displayed. Must have length 2 if oneD is FALSE. The default is to use all dimensions; therefore if oneD is FALSE the number of dimensions must be 2.
oneD	Logical scalar. If TRUE, projections onto individual coordinate axes are displayed. If FALSE, projection onto the plane defined by a single pair of axes is displayed. The default is FALSE if length(dims) is 2 and TRUE otherwise.
showSegs	If FALSE only the endpoints of line segments are plotted. If TRUE the segment connecting the endpoints is added as a light gray line.
rect	Optional inpRect object containing a hyper-rectangle in input space, or 2-row matrix with the coordinates of the lower and upper corners of one. If provided, the edges of the hyper-rectangle, projected onto dims, are displayed.
pt0	Optional vector or 1-row matrix with the coordinates of a point in input space, to be displayed with the symbol "0". (For example the ray origin when x was derived using rays.)
add	Logical scalar. If TRUE, the display is added to the current plot rather than starting a new plot.
...	Additional arguments passed to plot.default. Must not include type, which is set internally.

Details

Segment endpoints on the level set boundary are displayed with a blue "*" symbol. Endpoints on the boundary of the feasible region are displayed with an orange "x". Endpoints that are in the level set but not on its boundary are shown with a blue "+" symbol. If showSegs is TRUE, endpoints of each segment are connected by a solid gray line. If rect is provided, the projected boundary of the hyper-rectangle is shown with a dashed gray line.

Projection of a point onto a coordinate axis, or the plane spanned by a pair of coordinate axes, just amounts to using the point's coordinates for that/those dimension(s). Coordinates for other dimensions are ignored.

Value

NULL, invisibly.

See Also

lsetSegs, summary.lsetSegs. pairs.lsetSegs shows projections onto multiple pairs of axes in a single plot.

Examples

```
Ex <- dbl_ellipse_X1bnd_2dEx  # see '?dbl_ellipse_2dEx'
segs <- Ex$segs_checked
rect0 <- Ex$rect0
plot(segs, rect=rect0, main="Segments found along 50 random rays")

# 1D projections
plot(segs, oneD=TRUE, rect=rect0, main="Segments found along 50 random rays")
```

---

plot.respProfiles *Plot Profiles of a Response Function along Lines/Rays*

Description

Plot method for respProfiles objects. Uses ggplot2 graphics.

Usage

```
## S3 method for class 'respProfiles'
plot(x, y, groupBy=NULL, wrap=list(ncol=ceiling(sqrt(nprofs))), ...)
```

Arguments

x	A respProfiles object.
y	Ignored. (Only present for compatability with the generic.)
groupBy	Optional vector or factor with one element per line or ray in x. If specified, lines will be grouped by their value for this variable, and each group will be plotted in a separate panel. The default is to put all profiles in a single panel if they are *ray* profiles, and to put each in a separate panel for more general line profiles.
wrap	List that will be passed as arguments to ggplot2::facet_wrap, to control the arrangement of panels on the page. (For example nrow, ncol, scales, as.table, drop, dir, etc.)
...	Ignored, with a warning. (Only present for compatability with the generic.)

Details

The x-axis for the plot panels is *position* along the lines/rays. (See the definition of position in ?inpRays.) In general any given position value will represent different Euclidean distances from the line origin for different lines or rays.

A dashed horizontal line indicates the response threshold defining a level set, if one was specified in x. Any intersections of a profile line with the boundary of the level set are marked with a blue plotting symbol. Plotting symbols are an upward-pointing triangle if response is increasing along the line at that point, a downward-pointing triangle if it is decreasing, and a circle if it is flat or the trend is NA. An orange vertical line and plotting symbol are used for feasible region boundary intersections found in x.

To show ray profiles in separate panels, set groupBy to have a distinct value for each ray, such as 1:length(x), or names(x) if x has ray names.

Value

A ggplot object containing the plot.

See Also

respProfiles, inpLines, inpRays

Examples

```
# See the examples in '?respProfiles.fnObj'.
```

---

plotSegsList *Plot Sets of Level Set Segments in Separate Panels*

Description

Given a list of lsetSegs objects, plot their projections onto the same 2-D plane, each in a separate panel. ggplot2 graphics are used.

Usage

```
plotSegsList(x, dims, showSegs=TRUE, rect=NULL, ptSize=2,
    wrap=list(ncol=ceiling(sqrt(nx)), drop=FALSE))
```

Arguments

x	List of lsetSegs objects (or objects that contain an lsetSegs object, such as the return value of functions slicedBdryFromRays() and slicedBdrySearch()).
dims	Numeric or character vector of length 2, specifying the pair of input space dimensions onto which the line segments will be projected.
showSegs	If FALSE only the endpoints of line segments are plotted. If TRUE the segment connecting the endpoints is added as a light gray line.
rect	Optional 2-row matrix or inpRect object specifying a hyper-rectangle in input space, or a list of inpRect objects parallel to x. If provided, the edges of the hyper-rectangle, projected onto dims, are displayed in each panel.

ptSize	Size of point plotting symbols. Nominally in millimeters, but in practice displayed sizes are smaller. This size applies to level set boundary points; feasible region boundary points are 0.6 times as large.
wrap	List that will be passed as arguments to ggplot2::facet_wrap, to control the arrangement of panels on the page. (For example nrow, ncol, scales, as.table, drop, dir, etc.)

Details

Each lsetSegs object in x is plotted in a separate panel. Names of components in list x are used to label the panels.

Segment endpoints on the level set boundary are displayed with a blue "*" symbol. Endpoints on the boundary of the feasible region are displayed with an orange "x". Endpoints that are in the level set but not on its boundary are shown with a blue "+" symbol. If showSegs is TRUE, endpoints of each segment are connected by a solid gray line. If rect is provided, the projected boundary of the hyper-rectangle is shown with a dashed gray line.

Value

A ggplot object containing the plot.

See Also

lsetSegs. pairs.lsetSegs shows projections of a single lsetSegs object onto multiple pairs of dimensions in a single plot. slicedBdryFromRays and slicedBdrySearch return lists where each component contains an lsetSegs object, one per slice; those lists can be plotted using this function.

Examples

```
Ex <- circuitFailure_3dEx  # see '?circuitFailure_3dEx'
slsrch <- Ex$slsrch  # list returned by 'slicedBdrySearch()'

# Number of level set segments found per slice:
slsegs <- lapply(slsrch, lsetSegs)  # list of 'lsetSegs' objects
sapply(slsegs, length)

if (use("ggplot2")) {  # package 'ggplot2' required
  plt <- plotSegsList(slsrch, dims=c("dfrac", "shape"))
  print(plt + labs(title="Confidence region sliced by 'scale'"))
}
```

---

posnToCoord *Convert Positions on Lines or Rays to Point Coordinates*

Description

Convert positions on lines or rays to point coordinates.

Usage

```
posnToCoord(posn, lineidx, lines)
```

Arguments

posn	Vector of position values. NA is allowed, and will produce a point with NA coordinates.
lineidx	Vector of line or ray numbers (indices into lines). Its length must equal the number of positions in posn, or 1 to specify the same line for all positions.
lines	An object inheriting from class inpLines, representing lines or rays in input space.

Details

Lines and rays are defined by pairs of distinct points in input space, as represented by objects inheriting from class inpLines. The first point in a pair defines position 0 along the line, and is called the line origin. The second point defines position 1.0. Positions of other points on a line are based on linear interpolation or extrapolation from those points.

Value

A matrix with number of rows equal to the length of posn, containing point coordinates.

See Also

inpLines, inpRays

Examples

```
fspec <- fnSpec(inpnames=c("X1", "X2", "X3"))
rays <- inpRays(rbind(c(1, 1, 1), c(1, 2, 3)), origin=c(0, 0, 0),
                spec=fspec)  # two rays
posnToCoord(lineidx=c(1, 1, 1, 2, 2), posn=c(0, 0.5, NA, -0.5, 2),
            lines=rays)
```

---

print.inpLines *'print' Method for 'inpLines' and 'inpRays' Objects*

Description

print method for inpLines objects (including inpRays objects). Package users should call the generic print function.

Usage

```
## S3 method for class 'inpLines'
print(x, vecForm=inherits(x, "inpRays"), includeNames=inherits(x,
    "inpRays"), ...)
```

Arguments

x	An object inheriting from class inpLines (including inpRays).
vecForm	Logical scalar. If TRUE lines/rays are presented as position 0 points/origin point and direction vectors away from position 0. If FALSE, they are presented as position 0 points/origin and position 1 points. The default is TRUE for inpRays objects and FALSE for inpLines.
includeNames	Logical scalar. If TRUE, line/ray names, if any, will be included in the output. The default is TRUE for inpRays objects and FALSE for inpLines.
...	Additional print-related arguments that will be passed to the default print method for matrices.

Value

x, invisibly.

See Also

rayOrigin, rayPosn1, rayVec to extract individual components of inpRays objects.

Examples

```
fspec <- fnSpec(inpnames=c("a", "b", "c"))
pts <- rbind(c(0, 1, 3), c(2, 2, 4), c(5, -1, 2))
rays <- inpRays(pts, origin=c(0, 1, 2), spec=fspec)
print(rays)
print(rays, vecForm=FALSE)
names(rays) <- paste0("ray", seq_len(length(rays)))
print(rays)

lines <- inpLines(rbind(rayOrigin(rays), pts), spec=fspec,
                  pairidxs=cbind(c(1, 2), c(1, 3), c(1, 4)))
print(lines)
print(lines, vecForm=TRUE)
```

---

print.respProfiles *'print' Method for 'respProfiles' and 'lsetSegs' Objects*

Description

print method for respProfiles and lsetSegs objects. Package users should call the generic print function.

Usage

```
## S3 method for class 'respProfiles'
print(x, inclGrad=FALSE, ...)
```

Arguments

x	An object inheriting from class respProfiles (which includes lsetSegs objects).
inclGrad	Logical scalar. If TRUE and x includes gradient information, that matrix will also be printed.
...	Additional print-related arguments that will be passed to the default print methods for matrices and data frames.

Details

Point coordinates in x are printed as a matrix, and the response function values at those points are printed as a data frame. If x is an lsetSegs object, the two endpoints for each segment are printed in consecutive rows.

Value

x, invisibly.

Examples

```
Ex <- dbl_ellipse_X1bnd_2dEx  # see '?dbl_ellipse_2dEx'
segs <- Ex$segs_checked  # 'lsetSegs' object
print(segs[1:5])
```

---

ptCoord *Extract a Matrix of Point Coordinates*

Description

Extract a matrix containing coordinates of points in input space from an object.

Usage

```
ptCoord(x, ...)
```

Arguments

x	An object containing point coordinates (for example, a respProfiles or lsetSegs object or one inheriting from that class).
...	Additional arguments for specific methods. In particular, when x is an lsetSegs object, this can include a which argument; See DETAILS.

Details

This is an S3 generic function. There are methods to handle `inpLines`, `respProfiles`, and `lsetSegs` objects. (Package users should call the generic however.)

For a `respProfiles` object, points are ordered by the line/ray being profiled, and then by position within each line/ray.

For an `inpLines` object (which also includes `inpRays` objects), there are two points per line/ray, in consecutive rows of the returned matrix: odd-numbered rows are position 0 points and even-numbered rows are position 1 points.

For an `lsetSegs` object, a `which` argument is available. It is a character string indicating the form in which points are to be extracted. "1st" and "2nd" extract just the first or just the second endpoint of each segment. "both" returns the coordinates of the first endpoints of all the segments followed by those of the second endpoints. "paired" (the default) returns coordinates of first and second endpoints of each segment in consecutive rows (i.e., odd-numbered rows are first endpoints, even-numbered rows are second endpoints).

For both `respProfiles` and `lsetSegs` objects the number and order of the points matches rows of the data frame returned by `respInfo` for the same object and value of `which`.

Value

Matrix with one row per extracted point. Columns will be named with the input space dimension names.

See Also

`respInfo`; `rayOrigin`, `rayPosn1` extract coordinates of specific points for rays.

Examples

```
Ex <- dbl_ellipse_X1bnd_2dEx  # see '?dbl_ellipse_2dEx'
segs <- Ex$segs_checked  # 'lsetSegs' object
pts1 <- ptCoord(segs, which="1st")  # first endpoint of each segment
pts2 <- ptCoord(segs, which="2nd")  # second endpoint of each segment
pts12 <- ptCoord(segs, which="both")
identical(pts12, rbind(pts1, pts2))
# Default is 'which="paired"', endpoints for each segment in consecutive rows
head(ptCoord(segs), 10)
```

---

randomRays *Generate Rays in Input Space*

Description

Generate rays with random directions in input space.

Usage

```
randomRays(n, spec=NULL, scale=NULL, origin=NULL, rect=NULL,
    slice=NULL, warnDegenerate=TRUE)
```

Arguments

n	The number of rays to generate.
spec	An fnObj or fnSpec object with specifications for the input space. If omitted it will be taken from rect or scale, one of which must be specified in that case.
scale	Optional inpScale object that defines scaling and/or rotation transformations for the dimensions of the input space. If rect is specified, the default for scale is based on the lengths of its sides. See DETAILS.
origin	Vector or 1-row matrix with the coordinates of a single point in input space. All rays start from this point (i.e., it is position 0 for each ray). origin must belong to both rect and slice if they are specified. The default is rectRefPt(rect) if rect is specified, and otherwise the point with all coordinates equal to 0.
rect	Optional inpRect object, an axis-aligned hyper-rectangle in input space. Rays will be scaled so that position 1.0 on each ray is its intersection with the boundary of rect.
slice	Optional vector or 1-row matrix specifying one slice of input space. (See ?sliceMat for information about slices.) All the generated rays will lie in the slice. If specified it must intersect rect, and origin must lie in the slice.
warnDegenerate	Integer or logical scalar. When rect is specified and origin is on its boundary, it is possible for certain ray directions to lead to degenerate rays (i.e., positions 0 and 1 are the same point). Degenerate rays are not allowed. warnDegenerate controls whether they are silently dropped (0 or FALSE), dropped with a warning (1 or TRUE), or treated as an error (2).

Details

Let d denote the number of input space dimensions. Rays are generated following the same steps as those described in ?axisRays, with the exception of the initial set of unit vectors. Rather than being based on coordinate axes, here n vectors are sampled from the uniform distribution on the surface of the unit hypersphere in d_s dimensions. (d_s is the number of NA coordinates in slice, or equal to d if no slice is specified.)

Note that if some sides of rect are much longer than others, scale should probably be specified to reflect that. Otherwise few rays are likely to sample the long dimensions of rect. See the examples. Thus the default is scale=inpScale(rectSize(rect),spec=spec).

Value

An inpRays object. The origin will be the point specified by origin. The position 1.0 point for each ray will by default be distance 1 from the origin, but distances may differ if scale or rect is specified.

See Also

inpRays, axisRays, inpScale

Examples

```
oldopt <- lsetPkgOpt(reset=TRUE)  # default options

layout(matrix(1:4, nrow=2, byrow=TRUE))
#----- Rays pass through random points on circle centered at (3,2) with
# radius 1:
spec <- fnSpec(inpnames=c("A", "B"))
set.seed(1)
rslt <- randomRays(50, spec=spec, origin=c(3, 2))
plot(rslt, eqAxes=TRUE, main="Random rays, unit length")

# Effect of 'scale':
# Ellipsoid with wider range for A axis relative to B (semi-major and
# semi-minor axis lengths given by sqrt(diag(mat))).
scl <- inpScale(matrix(c(5, 0, 0, 1), nrow=2), spec=spec)
set.seed(1)
rslt <- randomRays(50, origin=c(3, 2), scale=scl)
plot(rslt, eqAxes=TRUE, main="Ray lengths scaled to an ellipse")

# Ellipsoid with wider range for A axis relative to B, and rotated:
scl <- inpScale(matrix(c(5, 2, 2, 1), nrow=2), spec=spec)
set.seed(1)
rslt <- randomRays(50, origin=c(3, 2), scale=scl)
plot(rslt, eqAxes=TRUE, main="Rays scaled and rotated")

# ... adjust lengths to put position 1.0 on the boundary of a rectangle:
rect <- inpRect(rbind(c(1, 1), c(5, 2.5)), spec=spec)
set.seed(1)
rslt <- randomRays(50, origin=c(3, 2), scale=scl, rect=rect)
plot(rslt, eqAxes=TRUE, main="Ray position 1 on a 'rect' boundary")

#-- When the sides of 'rect' have very different lengths, 'scale' makes
# a big difference.
layout(matrix(c(1:3, 0), nrow=2, byrow=TRUE))
# To highlight the effect, create a long, narrow rect, and use an off-center
# 'origin'.
corners <- rbind(c(0.5, 3), c(5.5, 3.5))
rect <- inpRect(corners, spec=spec)
# Default for 'scale' adjusts for the side lengths, so rays fill 'rect':
set.seed(1)
rslt <- randomRays(50, origin=c(1, 3.25), rect=rect)
plot(rslt, eqAxes=FALSE, main="Default 'scale' adjusts for 'rect' sides")

# If we intentionally ignore differences in side lengths, rays may be sparse
# in the long directions:
set.seed(1)
rslt <- randomRays(50, origin=c(1, 3.25), rect=rect,
                   scale=inpScale(spec=spec))
```

```
plot(rslt, eqAxes=FALSE, main="No adjustment for 'rect' sides")

#-- When 'origin' is itself on the boundary of 'rect', direction vectors
# that point outside 'rect' and lead to degenerate rays are instead reversed.
set.seed(1)
rslt <- randomRays(50, origin=c(2, 3.5), rect=rect)
plot(rslt, eqAxes=FALSE, main="Ray origin on boundary of 'rect'")

layout(matrix(1))
lsetPkgOpt(oldopt)  # restore starting options
```

`rayOrigin` *Coordinates of the Origin of a Set of Rays in Input Space*

Description

Extract the coordinates of the origin from an `inpRays` object. The origin is the point that defines position 0 along each ray.

Usage

```
rayOrigin(x)
```

Arguments

`x`	An `inpRays` object, representing rays in input space.

Value

One-row matrix with the coordinates of the ray origin.

See Also

`inpRays`, `rayPosn1`, `rayVec`, `rayRect`

Examples

```
# See the examples in '?inpRays'
```

rayPosn1 *Coordinates of the Position 1 Points of a Set of Rays*

Description

Extract the coordinates of the points that define position 1 for a set of rays in input space, from an `inpRays` object.

Usage

```
rayPosn1(x)
```

Arguments

`x`	An `inpRays` object, representing rays in input space.

Details

`inpRays` objects do not allow degenerate rays, so `rayPosn1(x)` will always be different from `rayOrigin(x)`.

Value

Matrix with one row per ray. Row names will be the ray names, if any.

See Also

`inpRays`, `rayOrigin`, `rayVec`, `rayRect`

Examples

```
# See the examples in '?inpRays'
```

rayRect *Hyper-Rectangle Associated with a Set of Rays*

Description

Extract the hyper-rectangle in input space associated with an `inpRays` object.

Usage

```
rayRect(x)
```

Arguments

x	An inpRays object, representing rays in input space.

Value

An inpRect object, or NULL if no hyper-rectangle is associated with x. When not NULL, the ray origin rayOrigin(x) will belong to the hyper-rectangle, and position 1 for each ray (rayPosn1(x)) will lie on its boundary.

See Also

inpRays, rayOrigin, rayVec, rayPosn1, inpRect

Examples

```
# See the examples in '?inpRays'
```

---

rayVec *Ray Direction Vectors from an 'inpRays' Object*

Description

Extract the vectors that define ray directions from an inpRays object. The vectors point away from the ray origin. They are simply the differences between position 1 points and the ray origin, and are not normalized.

Usage

```
rayVec(x)
```

Arguments

x	An inpRays object.

Value

A matrix with one row per ray. Row names will be the ray names, if any.

See Also

inpRays, rayOrigin, rayPosn1

Examples

```
# See the examples in '?inpRays'
```

rectCorners *Coordinates of the Corner Points of a Hyper-Rectangle*

Description

Extract the coordinates of the lower and upper corners of a hyper-rectangle in input space, from an `inpRect` object.

Usage

```
rectCorners(x)
```

Arguments

`x`	An `inpRect` object.

Value

Two-row matrix with the coordinates of the lower and upper corners of the hyper-rectangle.

See Also

`inpRect`, `rectRefPt`, `rectSize`

Examples

```
# See examples '?inpRect'
```

rectRefPt *Coordinates of the Reference Point of a Hyper-Rectangle*

Description

Extract the coordinates of the reference point of a hyper-rectangle in input space, from an `inpRect` object.

Usage

```
rectRefPt(x)
```

Arguments

`x`	An `inpRect` object.

Value

One-row matrix with the coordinates of the reference point for hyper-rectangle `x`. The reference point is an arbitrary point in the hyper-rectangle, defined when an `inpRect` object is created or modified.

See Also

`inpRect`, `rectCorners`, `rectSize`

Examples

```
# See examples '?inpRect'
```

`rectSize` *Lengths of the Sides of a Hyper-Rectangle*

Description

The lengths of the sides of a hyper-rectangle in input space, from an `inpRect` object.

Usage

```
rectSize(x)
```

Arguments

`x`	An `inpRect` object.

Value

Vector containing the lengths of the sides of the hyper-rectangle.

See Also

`inpRect`, `rectCorners`, `rectRefPt`

Examples

```
# See examples '?inpRect'
```

`respInfo` *Response Function Values at Input Points*

Description

Evaluate a response function at a set of points, or extract previous evaluations from another object. The result is a data frame with response function values and related information for each point.

Usage

```
respInfo(x, ..., inclGrad=FALSE)
```

Arguments

`x`	Matrix of points at which the response function is to be evaluated, or an object from which existing response function values will be extracted (for example a `respProfiles` or `lsetSegs` object).
`...`	Additional arguments for specific methods.
`inclGrad`	Logical scalar. If TRUE and the response function supports gradient calculations, gradient vectors will be included with the return value.

Details

This is an S3 generic function. In addition to the default method when `x` is a matrix of evaluation points, there are methods to extract response values from `respProfiles` and `lsetSegs` objects. (Package users should call the generic however.)

See `?respInfo.default` for information about arguments used when evaluating the response function at the points in matrix `x`.

When extracting response values from a `respProfiles` object, points are ordered by the line/ray being profiled, and then by position within each line/ray.

For an `lsetSegs` object, a `which` argument is available. It is a character string indicating the form in which points are to be extracted. "1st" and "2nd" extract responses for just the first or just the second endpoint of each segment. "both" returns responses at the first endpoints of all the segments, followed by those at the second endpoints. "paired" (the default) returns responses at first and second endpoints of each segment in consecutive rows (i.e., odd-numbered rows are for first endpoints, even-numbered rows are second endpoints).

For both `respProfiles` and `lsetSegs` objects the number and order of the points matches rows of the matrix returned by `ptCoord` for the same object, and the same `which` value.

Membership in the level set. A point is considered to belong to the specified level set if its response function value is not NA and is greater than or equal to `lsetthresh - atol`, where the absolute tolerance `atol` is calculated as `absTol(lsetthresh,tol=respTol(fnobj))`.

Value

Data frame with one row per evaluation point. Columns will include:

`resp`	Value of the response function at the point. NA for infeasible points.
`feas`	Logical, TRUE if the point is in the feasible region of input space for the response function. FALSE if and only if `resp` is NA. Never NA itself.
`lset`	Logical. For feasible points, TRUE if the point belongs to the level set (with allowance for a numerical tolerance, see DETAILS); FALSE if it does not; and NA if no threshold has been specified. Always FALSE for infeasible points.

Other columns may be present in the data frame, depending on the class of `x`. In particular, for `respProfiles` and `lsetSegs` objects there will be columns:

`line`	Integer index of the line or ray along which the point lies.
`posn`	Position of the point along the line/ray.
`feasbdry`	Logical, TRUE if the point has been identified as one where the line/ray crosses the *boundary* of the feasible region. FALSE means the point is not *known* to be a boundary crossing along the line/ray, or that it is not feasible. (The feasible region is assumed to be a closed set, so `feasbdry` must be FALSE if `feas` is.) `feasbdry` may be NA if no boundary search was done at that point.
`lsetbdry`	Logical. TRUE if the point has been identified as one where the response function along the line/ray crosses from at or above the threshold (minus a tolerance) to below, or vice versa. FALSE means the point is not *known* to be a threshold crossing along the line/ray, or that it is not in the level set. (Level sets are assumed to be closed, so `lsetbdry` must be FALSE if `lset` is, including for infeasible points.) For feasible points, `lsetbdry` will be NA if no level set threshold was specified, or if no boundary search was done at that point.
`deriv`	Derivative of the response function with respect to position along the line/ray, evaluated at the point. May be NA.
`trend`	Character string categorizing the local trend in response in the direction of the line/ray: "increasing", "decreasing", or "flat". May be NA.

`lsetSegs` objects will have an additional column:

`lsetcens`	Logical indicating whether the level set appears to extend beyond this point in the direction of the line/ray containing the segment. Defined as `!(lsetbdry | (lset & feasbdry))`. Never NA.

In practical terms the level set of a response function is the set of *feasible* points for which response is greater than or equal to a threshold. Therefore boundary points of the feasible region should also be considered boundary points of the level set, provided response is at or above the threshold. However the `lsetbdry` column of the returned data frame uses a narrower definition: it is TRUE only if the point is known to represent a *transition* of response values across the threshold. So the practical definition of a level set boundary point is actually `lsetbdry | (lset & feasbdry)`, which is just `!lsetcens`.

If `inclGrad` is TRUE, the returned data frame will have an attribute `gradient`. This is a matrix of gradient vectors for the response function at the evaluation points (one gradient vector per row). It is an error to request *calculation* of gradient values if the response function does not support them.

However if x is an existing object from which information is being extracted and it does not contain gradient information, the gradient attribute will be a matrix of NA values.

See Also

respInfo.default, ptCoord, lsetThresh

Examples

```
# Define a response function and evaluate it at a set of points.
respf <- function(x, inclGrad=FALSE) {
           resp <- x[, "X1"] + x[, "X2"]^2 + x[, "X3"]^3
           if (inclGrad) {
             grad <- cbind(1, 2*x[, "X2"], 3*x[, "X3"]^2)
             structure(resp, gradient=grad)
           } else  resp
         }
fobj <- fnObj(c("X1", "X2", "X3"), respfn=respf, hasgrad=TRUE)
pts <- rbind(c(0, 1, 3), c(2, 2, 4), c(5, -1, 2))  # three input space points
respInfo(pts, fnobj=fobj, lsetthresh=28)  # default is not to calc gradients
resp <- respInfo(pts, fnobj=fobj, lsetthresh=28, inclGrad=TRUE)
attr(resp, "gradient")

# Extract response function values from an existing object.
Ex <- dbl_ellipse_X1bnd_2dEx  # see '?dbl_ellipse_2dEx'
segs <- Ex$segs_checked  # 'lsetSegs' object
resp1 <- respInfo(segs, which="1st")  # first endpoint of each segment
resp2 <- respInfo(segs, which="2nd")  # second endpoint of each segment
resp12 <- respInfo(segs, which="both")
identical(resp12, rbind(resp1, resp2))
# Default is 'which="paired"', endpoints for each segment in consecutive rows
head(respInfo(segs), 10)
```

---

respInfo.default *Response Function Values at Input Points*

Description

Evaluate a response function at a set of points in input space, returning a data frame with response function values and related information. Package users should call the generic respInfo function.

Usage

```
## Default S3 method:
respInfo(x, fnobj, lsetthresh=NA_real_, ..., inclGrad=FALSE)
```

Arguments

x	Matrix or data frame, each row containing the coordinates of one point in input space. NA values are not allowed. A vector will be treated as a one-row matrix.
fnobj	fnObj object containing the response and feasibility functions.
lsetthresh	Scalar, the value of the response function that defines the level set of interest. NA (the default) means no level set is specified.
...	Ignored, with a warning. (Only present for compatibility with the generic.)
inclGrad	Logical scalar, whether to also calculate gradients at the input points. Must be FALSE (the default) if the response function in fnobj does not support gradient calculations.

Value

Data frame with one row per point. See the documentation for the generic function for details. If inclGrad is TRUE, the data frame will have an attribute gradient, a matrix with the same shape as x containing the gradient vector of the response function at each point.

See Also

fnObj, hasGrad

Examples

```
# Define a response function and evaluate it at a set of points.
respf <- function(x, inclGrad=FALSE) {
           resp <- x[, "X1"] + x[, "X2"]^2 + x[, "X3"]^3
           if (inclGrad) {
             grad <- cbind(1, 2*x[, "X2"], 3*x[, "X3"]^2)
             structure(resp, gradient=grad)
           } else  resp
         }
fobj <- fnObj(c("X1", "X2", "X3"), respfn=respf, hasgrad=TRUE)
pts <- rbind(c(0, 1, 3), c(2, 2, 4), c(5, -1, 2))  # three input space points
respInfo(pts, fnobj=fobj, lsetthresh=28)  # default is not to calc gradients
resp <- respInfo(pts, fnobj=fobj, lsetthresh=28, inclGrad=TRUE)
attr(resp, "gradient")
```

---

respProfiles *Profiles of a Response Function along Lines or Rays*

Description

Create a respProfiles object, containing response function values evaluated at positions along a set of 1-dimensional lines or rays in input space.

Usage

```
respProfiles(x, ...)
```

Arguments

x	fnObj object containing the response function, or another object from which profiles are to be extracted.
...	Additional arguments for specific methods.

Details

This is an S3 generic function, but the only user-level method is for fnObj objects. See ?respProfiles.fnObj for additional arguments.

Each point and its response function value in the respProfiles object x is associated with a line or ray, included in an inpLines or inpRays object within x. Use inpLines(x) or inpRays(x) to extract those lines/rays.

Points are identified by their position on the associated line. Positions are relative to the two points that define positions 0 and 1 for that line/ray: the position of any other point on it is calculated by linear interpolation or extrapolation from those two points. (See ?inpLines for more details.) Note that the Euclidean distance between position 0 and position 1 may differ between lines, so a given difference in positions may represent different Euclidean distances on different lines.

Function respInfo() will extract information about individual points from a respProfiles object, as a data frame. Coordinates for the points can be extracted with the ptCoord() function.

There are length, [, names, c, print, summary, and plot methods for respProfiles objects. length returns the number of lines/rays profiled, [extracts entire line/ray profiles (not individual points), and names extracts the names of the lines/rays that were profiled. If a level set threshold was specified when profiles were created, lsetSegs() will extract segments of lines/rays that lie within the level set.

Value

An object with S3 class respProfiles. It will contain one profile for each line/ray.

See Also

inpLines, inpRays, plot.respProfiles, summary.respProfiles, respInfo, ptCoord

Examples

```
# See examples in '?respProfiles.fnObj'
```

`respProfiles.fnObj` *Profiles of a Response Function along Lines or Rays*

Description

Evaluate a response function at positions along a set of 1-dimensional lines or rays in input space. For each line optionally find positions where it crosses the boundary of a specified level set or the boundary of the feasible region of the response function.

Usage

```
## S3 method for class 'fnObj'
respProfiles(x, lines, rays=NULL, posns, lsetthresh=NA_real_,
    feasbdrySrch=!is.na(lsetthresh), ...)
```

Arguments

`x`	`fnObj` object containing definitions of the response function and its input space.
`lines`	`inpLines` or `inpRays` object defining a set of lines or rays in input space.
`rays`	Alternative to `lines` argument, accepting only an `inpRays` object.
`posns`	Vector of positions along each line/ray at which the response function is to be evaluated. All must be finite. See DETAILS.
`lsetthresh`	Optional response function value that defines a level set of interest. A search will be made for positions along each line where it crosses the boundary of the level set. The search will be internal to the range of `posns`. The default is NA; i.e. no level set is specified.
`feasbdrySrch`	Logical scalar. If TRUE a search will be made for positions along each line where it intersects the boundary of the feasible region of the response function. The search will be internal to the range of `posns`. If `lsetthresh` is not NA, `feasbdrySrch` will be forced to TRUE.
`...`	Ignored, with a warning. (Only present for compatibility with the generic.)

Details

Finding level set boundary crossings along a line/ray relies on finding pairs of positions such that one position has a response greater than or equal to `lsetthresh` and the other has a response less than `lsetthresh`. Similarly, finding feasible region boundary intersections relies on finding pairs of positions such that one position is feasible and the other is not. Therefore the ability of this function to find level set or feasible region boundary intersections depends in part on a suitably wide and dense set of `posns`. See `?bdryFromRays` for more details about finding level set and feasible region boundary intersections.

Value

An object with S3 class `respProfiles`. See the documentation for the generic for more details.

See Also

fnObj, inpLines, inpRays, bdryFromRays

Examples

```
oldopt <- lsetPkgOpt(reset=TRUE)

#----- 1D example.
respf <- function(x, ...) { x*cos(x) }
xx <- seq(-3, 10, by=0.25)
plot(xx, respf(xx), type="l")
abline(h=0.2, lty=2)

fobj <- fnObj(c("X1"), respfn=respf)
ray1 <- inpRays(c(1), spec=fobj)  # unit vector
prof <- respProfiles(fobj, lines=ray1, posns=seq(-3, 10), lsetthresh=0.2)

# Methods for 'respProfiles' objects:
length(prof)  # 1, the number of profiled lines/rays
summary(prof)
if (use("ggplot2")) {  # plot method requires 'ggplot2'
  print(plot(prof))
}

# Extract and show level set segments.
lssegs <- lsetSegs(prof)
summary(lssegs)
plot(lssegs)
  # 3 segments.  Note that the left endpoint of the lowest segment is
  # marked as censored, because the level set extends further to the
  # left, outside the range of 'posns'.

#----- Multimodal example in 2D (surface with two elliptical contours).
Ex <- dbl_ellipse_X1bnd_2dEx  # see '?dbl_ellipse_2dEx'
fobj <- Ex$fobj

# Calculate ray profiles in four standard directions.
rays1 <- axisRays(fobj, degree=2)
prof1 <- respProfiles(fobj, rays=rays1, posns=seq(-8, 15), lsetthresh=-40)
summary(prof1)  # 77 of 101 profiled points are in the feasible region,
                # including 3 on its boundary.  48 of the feasible points
                # lie in the level set, including 8 on its boundary.

if (use("ggplot2")) {
  print(plot(prof1))  # Default is all ray profiles in one panel
  print(plot(prof1, groupBy=names(prof1)))  # each ray in a separate panel
    # Note that no points with X1 < 0 are evaluated (not in the feasible
    # region).
}

# Extract and show level set segments.
lssegs <- lsetSegs(prof1)
```

```
summary(lssegs)  # 6 segments:  two rays (1st and 4th) have one segment,
                 # two rays (2nd and 3rd) have two segments.
plot(lssegs)     # One segment endpoint is marked as censored:  the segment
                 # does not extend far enough in that direction to reach
                 # the level set boundary.

# Rather than rays (with a single, shared origin) one can also profile
# along arbitrary lines.
# Position 0, 1 points for three lines:
coords0 <- rbind(c(0, 4), c(8, 7), c(6, 2))
coords1 <- rbind(c(7, 9), c(4, -2), c(8, 2))
lines <- inpLines(x=coords1, x0=coords0, spec=fobj)
names(lines) <- c("(X1,X2) = (0,4) + posn*(7, 5)",
                  "(X1,X2) = (8,7) + posn*(-4,-9)",
                  "(X1,X2) = (6,2) + posn*(2,0)")
prof2 <- respProfiles(fobj, lines=lines, posns=(-10:10)/10,
                      lsetthresh=-40)
if (use("ggplot2")) {
  print(plot(prof2, groupBy=names(prof2)))  # each ray in a separate panel
}

lsetPkgOpt(oldopt)  # restore settings
```

`respTol` *Numerical Tolerance Specification for Response Values*

Description

Return the tolerance specification for response function values.

Usage

```
respTol(x)
```

Arguments

`x`	An `fnSpec` object, or object containing one. (That includes `fnObj`, `inpRays`, `inpRect`, `inpScale`, `respProfiles` and `lsetSegs` objects.)

Value

Two-element vector. The first element is the relative tolerance and the second is the minimum absolute tolerance.

See Also

`lsetPkgOpt` and `absTol` describe how tolerances are specified in this package. `fnSpec`, `fnObj.character`

Examples

```
# See examples in '?fnSpec'
```

---

segBoundingRect *Hyper-Rectangle Bounding a Set of Level Set Segments*

Description

Create an axis-aligned hyper-rectangle just large enough to contain all the level set points in an `lsetSegs` object. Optionally expand or contract the hyper-rectangle from that size.

Usage

```
segBoundingRect(x, refpt=NULL, expand=c(1.0, 1.0), warnEmpty=TRUE)
```

Arguments

`x`	An `lsetSegs` object.
`refpt`	A vector or one-row matrix containing the coordinates of the point to use as the reference point for the new hyper-rectangle. The new hyper-rectangle will always include it. The default is the centroid of the segment endpoints in `x`.
`expand`	One or two-element numeric vector of expansion factors for the hyper-rectangle, relative to what is required to just contain all the points in `x` and `refpt`. If specified, the second element is only used for segment endpoints that are marked as censored in `x`. See DETAILS.
`warnEmpty`	Logical scalar. If TRUE, a warning will be raised if `x` does not contain any segments.

Details

The midpoint of each line segment in `x` is found, and the endpoints are adjusted so that their distances from the midpoint are multiplied by factor `expand[1]` (for uncensored endpoints) or `expand[2]` (for censored endpoints). The smallest axis-aligned hyper-rectangle containing all of the adjusted endpoints and `refpt` is calculated. Censoring is defined by the variable `lsetcens` in the data frame returned by `respInfo(x)`.

`inpRect` objects are required to have positive d-dimensional volume, so any sides of the hyper-rectangle that have near-zero length will be increased enough to pass a check based on `inpTol(x)`.

Value

An `inpRect` object containing the calculated hyper-rectangle, or NULL if `x` does not contain any segments.

See Also

`inpRect`, `lsetSegs`, `boundingRect`

Examples

```
oldopt <- lsetPkgOpt(reset=TRUE)

Ex <- dbl_ellipse_2dEx  # see '?dbl_ellipse_2dEx'
segs <- Ex$segs_checked

bnding_rect <- segBoundingRect(segs, expand=1.0)
plot(segs, rect=bnding_rect, main="Level set segs with bounding rectangle")

lsetPkgOpt(oldopt)
```

`segInfo` *Information About Level Set Line Segments*

Description

Extract a data frame of information about each line segment in an `lsetSegs` object.

Usage

```
segInfo(x)
```

Arguments

`x`	An `lsetSegs` object.

Value

Data frame with one row per segment in `x`. The returned data frame has columns:

`line`	Index of the line/ray (i.e., in `inpLines(x)`) along which the segment lies.
`posn1`, `posn2`	Positions of the segment endpoints along their line.
`feasbdry1`, `feasbdry2`	Logicals indicating whether a segment endpoint is known to be on the boundary of the feasible region of the response function.
`lsetcens1`, `lsetcens2`	Logicals indicating *censoring* of the line segment with respect to the boundary of the level set. That is, whether the level set appears to extend beyond each endpoint of the segment, in the direction of the line containing it. Will never be NA.

See Also

`lsetSegs`

Examples

```
Ex <- dbl_ellipse_X1bnd_2dEx  # see '?dbl_ellipse_2dEx'
segs <- Ex$segs_checked
summary(segs)
head(segInfo(segs), 10)
```

`slicedBdryFromRays` *Ray-Based Exploration of the Boundary of a Level Set*

Description

Identify the boundary of a level set by finding points where a set of rays intersect it. A separate set of rays is used for each of the specified slices of the input space.

Usage

```
slicedBdryFromRays(x, lsetthresh, slices, rect, scale=NULL,
    origins=NULL, currentBdry=NULL, control=list())
```

Arguments

`x`	`fnObj` object containing definitions of the response function and its input space.
`lsetthresh`	Response function value that defines the level set of interest.
`slices`	Matrix specifying one or more slices of input space. See `?sliceMat` for details. All slices must intersect `rect`.
`rect`	`inpRect` object specifying the hyper-rectangle in input space where the search for boundary points is focused. Rays will be defined such that position 1 along a ray is its intersection with the boundary of `rect`.
`scale`	Optional `inpScale` object specifying scaling to be applied to input space dimensions when calculating distances and generating rays. May also be a vector or a square matrix from which an `inpScale` object will be created; see `?inpScale` for details. The default is to use the lengths of the sides of `rect` as scaling factors for the input space dimensions.
`origins`	Optional matrix specifying the point to use as the ray origin for each slice. The number of rows must match the number of slices. Each origin must belong to `rect`, and to its corresponding slice. Default value: If `currentBdry` is specified and is not empty, origins are derived rom it according to `control$originCrit` (see `?bdrySearch` for details). Otherwise the reference point of `rect`, projected to each slice, is used.
`currentBdry`	Optional object containing existing information about the level set boundary: either an `lsetSegs` object (e.g. from a call to `bdryFromRays()`) or the result of a previous call to `bdrySearch()`, `slicedBdrySearch()` or `slicedBdryFromRays()`.
`control`	Optional list with parameters to control the search process. See `?srchControl` for the available parameters and their defaults. Any value specified here overrides the default from `srchControl()`.

Details

When the rays used in each slice are random and `control$seedRays` is not NA (which it is not by default; see `?srchControl`), the state of R's random number generator is reset between slices, to its value for the first slice. This is done so that results for a given slice do not depend on the number or order of other slices.

Value

List with one component per slice. Each component is itself a list, with the same structure as the value returned by `bdrySearch()`, plus slice information. All boundary points and origins will lie in their slices.

Note that the rays used for slice `i` can be extracted by `inpRays(rslt[[i]]$segs)`, where `rslt` is the list returned by this function.

List names will be taken from `rownames(slices)`.

See Also

sliceMat, bdryFromRays, srchControl, plotSegsList

Examples

```
oldopt <- lsetPkgOpt(reset=TRUE)

Ex <- circuitFailure_3dEx  # see '?circuitFailure_3dEx'
fobj <- Ex$fobj
thresh <- Ex$thresh
theta_mle <- Ex$theta_mle
theta_vcov <- Ex$theta_vcov

# Initial level set search without slicing.
rect1 <- inpRect(rbind(theta_mle - 3*sqrt(diag(theta_vcov)),
                       theta_mle + 3*sqrt(diag(theta_vcov))),
                 spec=fobj)
scl1 <- inpScale(theta_vcov, spec=fobj)
set.seed(1)
rays1 <- randomRays(100, rect=rect1, scale=scl1)
segs1 <- bdryFromRays(fobj, rays=rays1, lsetthresh=thresh)

# The three input dimensions are 'dfrac', 'shape', and 'scale'.
# Consider slices defined by six values of 'scale'.
slices <- sliceMat(list(scale=c(10, 30, 50, 70, 90, 110)), spec=fobj)

# To apply `slicedBdryFromRays()` we need to specify a search region that
# intersects all the slices, such as
rect2 <- boundingRect(rect1, slices, expand=1.1)

# Use 'currentBdry=segs1' as a "warm start" for searching within slices.
slsrch <- slicedBdryFromRays(fobj, lsetthresh=thresh, slices=slices,
                             rect=rect2, scale=scl1, currentBdry=segs1)
if (use("ggplot2")) {
```

```
  plt <- plotSegsList(slsrch, dims=c("dfrac", "shape"), rect=rect1)
  print(plt + theme(panel.background=element_blank()) +
        labs(title="Level set sliced by 'scale'"))
}

lsetPkgOpt(oldopt)  # restore previous settings
```

`slicedBdrySearch` *Ray-Based Exploration of the Boundary of a Level Set*

Description

Identify the boundary of a level set by finding pairs of boundary points, such that the line segment connecting each pair appears to lie entirely in the level set. The search is carried out separately within specified slices of the input space. For each slice an adaptive, ray-based search is carried out, as described for function `bdrySearch()`. A list is returned, where each component is the result returned by `bdrySearch()` for a single slice.

Usage

```
slicedBdrySearch(x, lsetthresh, slices, rect, scale=NULL,
    initOrigins=NULL, currentBdry=NULL, control=list())
```

Arguments

`x`	An `fnObj` object, containing definitions of the response function and its input space.
`lsetthresh`	Response function value that defines the level set of interest.
`slices`	Matrix specifying one or more slices of input space. See `?sliceMat` for details. All slices must intersect `rect`.
`rect`	`inpRect` object specifying the hyper-rectangle in input space where the search for boundary points is focused. Search rays extending from each origin will be defined such that position 1 along a ray is its intersection with the boundary of `rect`.
`scale`	Optional `inpScale` object specifying scaling to be applied to input space dimensions when calculating distances and generating rays. May also be a vector or a square matrix from which an `inpScale` object will be created; see `?inpScale` for details. The default is to use the lengths of the sides of `rect` as scaling factors for the input space dimensions.
`initOrigins`	Optional matrix specifying the point to use as the initial ray origin for each slice. The number of rows must match the number of slices. Each origin must belong to `rect`, and to its corresponding slice. Default value: If `currentBdry` is specified and is not empty, the inital origin is derived from it according to `control$originCrit` (see `?bdrySearch` for details). Otherwise the reference point of `rect`, projected to each slice, is used.

currentBdry — Optional object containing existing information about the level set boundary: either an lsetSegs object (e.g. from a call to bdryFromRays()) or the result of a previous call to bdrySearch(), slicedBdrySearch() or slicedBdryFromRays().

control — Optional list with parameters to control the search process. See ?srchControl for the available parameters and their defaults. Any value specified here overrides the default from srchControl().

Details

By default the results for a given slice do not depend on the number or order of other slices in slices; see ?srchControl.

Value

List with one component per slice. Each component is itself a list, with the same structure as the value returned by bdrySearch(), plus slice information. All boundary points and origins will lie in their slice.

List names will be taken from rownames(slices).

See Also

sliceMat, bdrySearch, srchControl, plotSegsList

Examples

```
oldopt <- lsetPkgOpt(reset=TRUE)

Ex <- circuitFailure_3dEx  # see '?circuitFailure_3dEx'
fobj <- Ex$fobj
thresh <- Ex$thresh
theta_mle <- Ex$theta_mle
theta_vcov <- Ex$theta_vcov

# Initial level set search without slicing.
rect0 <- inpRect(rbind(theta_mle - 3*sqrt(diag(theta_vcov)),
                       theta_mle + 3*sqrt(diag(theta_vcov))),
                 spec=fobj)
scl0 <- inpScale(theta_vcov, spec=fobj)
bsrch <- bdrySearch(fobj, lsetthresh=thresh, rect=rect0, scale=scl0,
                    initOrigin=theta_mle,
                    control=list(axisDegree=2, updateScale=2))

# The three input dimensions are 'dfrac', 'shape', and 'scale'.
# Consider slices defined by four values of 'scale'.
slices <- sliceMat(list(scale=c(10, 40, 70, 110)), spec=fobj)

# To apply `slicedBdrySearch()` we need to specify a search region that
# intersects all the slices, such as
rect1 <- boundingRect(bsrch$rect, slices, expand=1.1)

# Use 'currentBdry=bsrch' as a "warm start" for searching within slices.
```

```
slsrch <- slicedBdrySearch(fobj, lsetthresh=thresh, slices=slices,
                           currentBdry=bsrch, rect=rect1, scale=bsrch$scale,
                           control=list(axisDegree=2, maxSteps=6))

# If desired, plot results, one slice per panel:
#if (use("ggplot2")) {
#  plt <- plotSegsList(slsrch, dims=c("dfrac", "shape"), rect=rect1)
#  print(plt + theme(panel.background=element_blank()) +
#        labs(title="Level set sliced by 'scale'"))
#}

lsetPkgOpt(oldopt)  # restore previous settings
```

`sliceMat` *Define Slices of Input Space*

Description

Create a matrix that specifies slices of input space.

Usage

```
sliceMat(x, spec, names=NULL)
```

Arguments

`x`	Matrix or list specifying slices. See DETAILS. If omitted, a matrix with 0 rows (i.e., no slices) is returned.
`spec`	`fnObj` or `fnSpec` object with specifications for the input space.
`names`	Character vector or logical scalar. FALSE means the slices will have no names. TRUE constructs default names of the form `tag=value,tag=value, ..., tag=value` from the non-NA elements of each row of `x`. A character vector should have one element per slice. If `x` is a matrix, the default is to use its row names, or TRUE if those are NULL. If `x` is a list, the default is TRUE.

Details

A slice of `d`-dimensional input space is defined as an axis-aligned hyperplane with `d_s` dimensions, where `1 <= d_s <= d`. Points in a slice have fixed values for `d - d_s` of their coordinates, while the coordinates for the remaining `d_s > 0` dimensions take any values. This is reflected in how slices are represented internally: As rows of a `d`-column matrix, where in each row `d - d_s` elements are set to fixed values and `d_s` elements are NA. (The slice dimension `d_s` can vary between rows.)

The case when `d_s = d` (no dimensions have fixed values) is called the identity slice, and represents the whole input space. To specify the identity slice, `x` must be provided in matrix form with a row of all NA values.

The list form of argument x should consist of components named by a subset of input space dimensions. Each component should be a vector of values to be used for the corresponding dimension. Values in each component will be crossed with all values in the other components, in the style of base::expand.grid. The result will be a matrix with prod(lengths(x)) rows (slices). Matrix elements for any dimension not mentioned in x will be set to NA.

Value

A matrix with one row per slice.

Examples

```
fspec <- fnSpec(inpnames=c("X1", "X2", "X3"))

# Slices specified directly in matrix form:
slices <- sliceMat(rbind(c(4, 3, NA), c(4, NA, NA), c(NA, 3, NA),
                         c(NA, NA, NA)), spec=fspec)
slices

sliceMat(spec=fspec)  # no slices

# Slices specifed in list form:
slices2 <- sliceMat(list(X2=c(3, -1), X1=c(4, 0, 1)), spec=fspec)
slices2  # 3 x 2 = 6 slices
```

---

srchControl *Default Control Parameters for Boundary Search Functions*

Description

Return a list with default settings for various control parameters used by level set boundary search functions.

Usage

```
srchControl(d)
```

Arguments

d	Dimension of the input space, or vector of slice dimensions if slicing is being used.

Details

Functions bdryFromRays, bdrySearch, and their sliced* versions each have a control argument that specifies certain parameters used to control the search for level set boundary points. This function documents the parameters and their default values. Note that not all parameters are used by a given function, and default values can be overridden in the control argument.

The control parameters are categorized into four groups:

Finding boundary points along given rays. (These parameters are used by all search functions.):

initPosns Vector of non-negative positions along each ray that will be used as starting points to search for intersections with the level set boundary. Position 0 (the ray origin) will always be included, and at least one other strictly positive value is required. Default: c(0, 1).

initPosns2 Optional vector of additional positive positions to use as starting points for finding boundary crossings. These additional positions are only tried for rays for which no or only censored boundary crossings are found using initPosns. The default is c(0.25, 0.5, 2, 4).

bothDirections Logical scalar. If TRUE (the default), the search for boundary intersections is also done along a reversed version of each ray, and combined with those from the original ray.

Generating rays to search along, given a ray origin. (Used by bdrySearch(), slicedBdrySearch(), slicedBdryFromRays().):

rayType Character string. random means rays will be generated by function randomRays(); axis means axisRays() will be used. Default is axis for bdrySearch() and slicedBdrySearch(), random for slicedBdryFromRays().

nRays Vector containing the number of rays to generate for each slice when rayType is random. Default is 1 if the slice dimension is 1, and pmin(5^d, 1000) for larger d (i.e., 1000 for d > 4).

axisDegree When rayType is axis, vector of positive integers less than or equal to slice dimensions in d. For slice i, rays are generated from unit vectors along each free coordinate axis and combinations of those vectors up to axisDegree[i] (see ?axisRays). A scalar will be repeated for all slices. Default: 1.

rotateRays Logical scalar. If TRUE and rayType is axis, the set of axis-based rays will be randomly rotated after each step (new origin). Default is TRUE.

seedRays Integer used to initialize the RNG stream for random rays when rayType is random, or the random rotations when rayType is axis and rotateRays is TRUE. If NA, the current state of R's random number stream will be used, and is never reset. If not NA, the current state of R's RNG will be saved and restored at the end, as if random number generation had not been used. Default: 1.

warnDegenerate Integer or logical scalar. When a ray origin is on the boundary of a search region (hyper-rectangle), it is possible for certain ray directions to lead to degenerate rays (i.e., positions 0 and 1 are the same point). Degenerate rays are not allowed. warnDegenerate controls whether they are silently dropped (0 or FALSE), dropped with a warning (1 or TRUE), or treated as an error (2). The default is 0.

Generating initial or new ray origins. (Used by bdrySearch(), slicedBdrySearch(), slicedBdryFromRays().):

maxSteps Vector setting the maximum number of steps (ray origins) the search will use to explore the level set boundary for each slice. A scalar value will be repeated for all slices. Default: `pmin(d^2, 100)`.

originBias Scalar in [0, 1] used during the generation of new candidate origins. It specifies where along each level set segment the candidate origins are placed relative to the segment's midpoint (0) and endpoints (1). Values close to 1 bias origin selection toward existing level set boundary points, while values close to 0 prefer new origins toward the interior of the level set. Default: 0.6.

originCrit Character string specifying how the new origin at each step is selected from among the candidate origins. One of `maximindist`, `minimaxdist`, or `maxproj`. The default is `maximindist`. See `?bdrySearch`.

lsetFirst Logical scalar. If TRUE (the default), candidate origins that belong to the level set are always preferred over those that do not, regardless of their criterion value.

srchDirection Vector, 1-row matrix, or 1-element `inpLines` object specifying a direction in input space. Required if `originCrit` is `maxproj` and ignored otherwise. The boundary search will select the candidate origin with maximum projection onto this vector.

maxCandOrigins Positive integer. New ray origins are selected from among a potentially large number of candidates generated from previously-found level set segments. Depending on the selection criterion `originCrit`, the distance from each candidate to each existing origin or existing boundary point must be calculated, which can result in a large distance matrix. This parameter limits the number of candidates that will be considered. Default is 1000.

maxDistances Positive integer. Soft limit on the total number of elements allowed for a distance matrix, during the selection of new origins. Default is 1e7.

Initializing or updating the search region (`rect`) and input scaling (`scale`) during a boundary search, based on current information about the level set. (Used by `bdrySearch()`, `slicedBdrySearch()`, `slicedBdryFromRays()`.):

updateRect1, updateRect2 Numeric scalars. The first specifies whether the search region `rect` will be updated before each step (new ray origin) to include all previously found boundary points. The update is done by `segBoundingRect()` with expansion factors 1.2 for uncensored boundary points and 2.0 for censored points. A value of 0 means no update, > 0 enables updates; the default is 1. `updateRect2` is the maximum number of times to allow `rect` to be expanded *within* each step (fixed origin and ray directions), to try to resolve censored level set segments along its rays. 0 means no expansion; the default is 2 times.

updateScale Logical or numeric scalar. If FALSE or 0, the same `scale` is used at every step. If TRUE or 1 (the default), scaling factors for input space dimensions are updated at each step to the lengths of the sides of the bounding hyper-rectangle for previously found boundary points. The dimension correlation structure specified by argument `scale`, if any, is unchanged. If `updateScale` is 2, both scaling factors and correlation structure are updated at each step, using previously found boundary points.

Value

List with default values for the `control` argument of functions `bdryFromRays`, `bdrySearch`, and their `sliced*` versions.

See Also

bdryFromRays, bdrySearch, slicedBdrySearch, slicedBdryFromRays, axisRays, randomRays

Examples

```
# The default control parameters for input space dimensions of 1, 2, 3:
str(srchControl(1:3))
```

---

summary.lsetSegs *'summary' Method for 'lsetSegs' Objects*

Description

summary method for lsetSegs objects. Package users should call the generic summary function.

Usage

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

Arguments

object	An lsetSegs object.
...	Ignored, with a warning. (Only resent for compatibility with the generic.)

Details

If object is empty (no level set segments), respRange, bBox, and bBoxVol will contain only NA values.

There is a print method for the objects returned by this function.

Value

An object with class summary.lsetSegs. This is a list with components:

fnSpec	fnSpec object containing information about the response function and its input space.
nlines	Number of lines/rays to which segments in object belong.
inpLines	inpLines or inpRays object containing the lines/rays to which segments in object belong.
lsetThresh	Response function value that defines the level set to which segments belong.
respRange	Vector with minimum and maximum response function values over all segment endpoints. (The minimum will be greater than or equal to lsetThresh.)

bBox — 2 x d matrix containing the lower and upper corners of the bounding box for the level set segments in x: The smallest, axis-aligned hyper-rectangle that contains all the level set boundary points in x.

bBoxVol — Volume of the bounding box.

lineInfo — Data frame with one row per line/ray in inpLines, and columns:

nsegs: Total number of level set segments on the line/ray. (May be 0.)

nfeasbdry: Number segment endpoints marked as being on the boundary of the feasible region.

nlsetbdry: Number segment endpoints marked as being on the boundary of the level set.

ncens: Number of segment endpoints marked as being censored (i.e., the level set appears to extend beyond the endpoint in the direction of the segment).

bBoxLen: Length of the intersection of the line with the bounding box bBox. (May be 0.)

lsetLen: Combined length of all the level set segments on the line within the bounding box.

lsetLenCens: Logical indicating whether lsetLen may be an underestimate for the line. TRUE if either endpoint of a level set segment lies in the interior of the bounding box and is marked as censored in object (see ?lsetSegs).

See Also

lsetSegs

Examples

```
oldopt <- lsetPkgOpt(reset=TRUE)

Ex <- dbl_ellipse_X1bnd_2dEx  # see '?dbl_ellipse_2dEx'
segs <- Ex$segs_checked
summary(segs)  # 75 segments along 50 rays (25 rays have 2 segs each)
               # 34 of 150 segment endpoints are on the foundary of the
               # feasible region.

lsetPkgOpt(oldopt)
```

---

summary.respProfiles *'summary' Method for 'respProfiles' Objects*

Description

summary method for respProfiles objects. Package users should call the generic summary function.

Usage

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

Arguments

object	A respProfiles object.
...	Ignored, with a warning. (Only present for compatibility with the generic.)

Value

An object with class summary.respProfiles. This is a list with components:

fnSpec	fnSpec object containing specifications for the response function.
nlines	Number of lines/rays profiled in object.
inpLines	inpLines or inpRays object containing the lines/rays that were profiled.
lsetThresh	Response function value that defines a level set for which line intersections with its boundary were searched. May be NA.
ptCounts	Matrix with one row per line/ray profile, and columns of point counts for each: **npts:** Total number of profile points. **nfeas:** Number of points in the feasible region. **nfeasbdry:** Number points marked as being on the boundary of the feasible region. **nlset:** Number of points in the level set (NA if lsetThresh is NA). **nlsetbdry:** Number points marked as being on the boundary of the level set (NA if lsetThresh is NA).
respRange	Matrix with one row per line/ray profile, and two columns containing the minimum and maximum response function values in the profile for each line/ray. Values will be NA if nfeas is 0.

See Also

respProfiles

Examples

```
# See the examples in '?respProfiles'.
```

update.fnObj *Update Certain Specifications in an 'fnObj' Object*

Description

Update certain specifications in an fnObj object. Package users should call the generic update function.

Usage

```
## S3 method for class 'fnObj'
update(object, inptol=NULL, resptol=NULL, derivmethod=NULL,
    warnNAresp=NULL, warnInfresp=NULL, ...)
```

Arguments

object	fnObj object to be updated.
inptol, resptol, derivmethod, warnNAresp, warnInfresp	New values for specifications in object. See ?fnObj and ?fnObj.character for their meanings. Any that are NULL will be left unchanged in object.
...	Ignored, with a warning. (Only present for compatibility with the generic.)

Details

The number and names of input space dimensions, the response function, and its feasible region cannot be updated.

Value

An fnObj object.

See Also

fnObj, fnObj.character

Examples

```
fnobj <- fnObj(c("X1", "X2"), function(x) { colSums(x) },
               inptol=c(1e-6, 1e-8))
inpTol(fnobj)

fnobj2 <- update(fnobj, inptol=c(1e-4, 2e-8))
inpTol(fnobj2)
```

update.inpRays *Rays in Input Space*

Description

Update an inpRays object with a new origin and/or hyper-rectangle. Package users should call the generic update function.

Usage

```
## S3 method for class 'inpRays'
update(object, origin=NULL, rect=NULL, warnDegenerate=TRUE, ...)
```

Arguments

object	inpRays object.
origin, rect	Optional values to update the ray origin and/or the hyper-rectangle associated with the rays. (Rays will be scaled so that position 1.0 on each ray is its intersection with the boundary of rect.) After any updating, the ray origin must be in the interior or on the boundary of the associated hyper-rectangle.
warnDegenerate	Integer or logical scalar. If 1 or TRUE (the default), a warning will be raised if any rays are dropped because their position 1.0 point is (near-)equal to the ray origin. If 0 or FALSE, degenerate rays are dropped silently. If 2, degenerate rays will be treated as an error.
...	Ignored, with a warning. (Only present for compatibility with the generic.)

Details

Ray directions are preserved. However rays may be dropped during the update if they become degenerate (their position 1.0 point is equal to the ray origin, to within a numerical tolerance). This can happen if a new rect is specified, or with a new origin if object already has an associated hyper-rectangle.

Value

An object with S3 class inpRays, inheriting from inpLines.

Examples

```
fspec <- fnSpec(inpnames=c("X1", "X2"))
set.seed(1)
rays <- randomRays(10, spec=fspec)  # ray vectors have unit length,
                                    # origin at (0, 0)
rect2 <- inpRect(rbind(c(0, 0), c(1, 1)), spec=fspec)  # unit square
rays2 <- update(rays, origin=c(0.3, 0.4), rect=rect2)

plot(rays, eqAxes=TRUE, main="Updating ray origin and rect")
```

```
plot(rays2, rect=rect2, add=TRUE, rayPar=list(col="blue"))
```

`update.inpRect` *Update Method for 'inpRect' Objects*

Description

Update an `inpRect` object with a new reference point or size. Package users should call the generic `update` function.

Usage

```
## S3 method for class 'inpRect'
update(object, refpt=NULL, expand=NULL, ...)
```

Arguments

`object`	`inpRect` object.
`refpt`	A vector or one-row matrix containing the coordinates of a single point. The point must belong to the hyper-rectangle, before applying `expand`. (Note however that if it is only slightly outside, within a numerical tolerance, then the hyper-rectangle itself will be adjusted so that `refpt` is exactly on its boundary.) The default is to leave the current eference point unchanged.
`expand`	Positive scalar. The hyper-rectangle will be expanded away from/ contracted toward `refpt`, such that its size along each dimension is multiplied by `expand`. It is an error to specify an expansion factor so small that the resulting hyper-rectangle is reduced to (nearly) zero volume. The default is to leave the size of `object` unchanged.
`...`	Ignored, with a warning. (Only present for compatibility with the generic.)

Details

Hyper-rectangles represented by `inpRect` objects must be finite and have non-zero `d`-dimensional volume. See `?inpRect` for how the latter condition is defined.

Values of `expand` less than 1 will contract the hyper-rectangle toward `refpt`.

Value

An `inpRect` object.

Examples

```
fspec <- fnSpec(inpnames=c("X1", "X2"))
rect <- inpRect(rbind(c(-1, 3), c(2, 5)), refpt=c(0, 4), spec=fspec)
rect2 <- update(rect, refpt=c(1, 3), expand=1.5)
  # Size increases from 3 x 2 to 4.5 x 3.  Expansion is relative to new
  # reference point.
plot(rect2, showRefPt=TRUE, border="red", main=
  "Original (blue) and updated (red) 'inpRect' objects")
plot(rect, showRefPt=TRUE, border="blue", add=TRUE)
```

Index