

levelSets Problem Setup

Richard Raubertas

03 April 2026

Contents

1	Introduction	1
2	An example	2
3	Specifying the response function and its input space	2
3.1	The <code>fnObj</code> function and class	2
3.1.1	<code>respfn</code> and <code>...</code> arguments	3
3.1.2	<code>feasfn</code> , <code>feasbnds</code> and <code>...</code> arguments	3
3.1.3	<code>hasgrad</code> and <code>derivmethod</code> arguments	3
3.1.4	<code>inptol</code> and <code>resptol</code> arguments	3
3.2	<code>fnSpec</code> class and function	4
4	Specifying the search region for a level set	4
5	Specifying search rays	4
5.1	Rays and hyper-rectangles	5
6	Scaling of input space dimensions	5
7	Profiling a response function along rays	6
8	Level set boundary search	8
8.1	<code>bdryFromRays()</code> : Fixed set of rays	8
8.1.1	The search algorithm	8
8.1.2	<code>lsetSegs</code> objects	8
8.2	<code>bdrySearch()</code> : Adaptive selection of rays	9
8.3	Slices of input space	9

1 Introduction

This vignette goes beyond the *Introduction to the **levelSets** Package* vignette to provide more details about how to set up and run a level set problem. It uses a simple 2D problem for illustration. More extended examples are provided in the *Example* vignettes.

Recall that a level set of a response function $f(x)$ is the set of d -dimensional input points x for which the function value is greater than or equal to a specified threshold t :

$$L_f(t) = \{x : f(x) \geq t\}$$

If there are restrictions on the inputs that f will accept, we call the allowed portion of the input space the *feasible region* for f , and any point outside the feasible region is *infeasible*.

This package maps out the boundary of a level set by finding its intersections with collections of 1-dimensional *rays*. A ray is just a line with a defined origin (referred to as *position 0* of the ray) and orientation. A ray's orientation is specified by a second, distinct point that defines position 1 along the ray. Any other point on the ray has a *position* value obtained by interpolation or extrapolation from the origin and position 1 points. (Ray points on the opposite side of the origin from the position 1 point have negative position values.)

The package provides tools to generate rays, find intersections, and visualize results.

2 An example

The Rosenbrock “banana” function will be used for illustration. It has a 2-dimensional input space, and a minimum value of 0 at the point (1, 1). Since level sets are defined to have function values at or *above* a threshold, we use the negative of the function:

$$f(x_1, x_2) = -100((x_2 - x_1^2)^2) - (1 - x_1)^2$$

Contours of this function can be strongly curved, so level sets may not be convex. The level set threshold we use is $f \geq -50$.

The response function can be coded in R as

```
respfn <- function(x, inclGrad=FALSE) {
  y <- -1 * (100*((x[, 2] - x[, 1]^2)^2) + (1 - x[, 1])^2)
  if (inclGrad) {
    grad <- cbind(400*(x[, 1]*(x[, 2] - x[, 1]^2)) + 2*(1 - x[, 1]),
                  -200*(x[, 2] - x[, 1]^2))
    attr(y, "gradient") <- grad
  }
  y
}
```

thresh <- -50

The gradient calculation is simple so it is included, but gradients are not necessary in general.

Although this function is well-defined for any (x_1, x_2) , for illustration we define the feasible region to be $\{x : x_1 \geq -2\}$. A feasibility function can be coded as

```
feasfn <- function(x) {
  (x[, 1] >= -2)
}
```

In this package point coordinates are always represented as rows of a d -column matrix, one row per point. Both the response and feasibility functions must accept as their first argument a matrix of point coordinates, and return a vector with one value per point: a numeric value for the response function, and TRUE or FALSE for the feasibility function.

The following sections describe package functions and classes, in the order they are likely to be used in studying a level set.

3 Specifying the response function and its input space

3.1 The fnObj function and class

The primary function for this step is `fnObj()`, which creates objects of that class. `fnObj` objects contain the response and feasibility functions along with related information. This section covers the basics for the main arguments; see `?fnObj` for additional arguments and full details. An example of its use for the banana response function is

```
library(levelSets)
fobj <- fnObj(c("x1", "x2"), respfn=respfn, feasfn=feasfn, hasgrad=TRUE)
```

The first argument must be a character vector of names for the input space dimensions. Dimension names must be strings that would be valid names for R objects.

3.1.1 respfn and ... arguments

respfn is a user-supplied function that takes as its first argument a matrix of point coordinates, and returns a vector of response function values at those points. Any additional arguments in ... will be passed to this function on each call.

3.1.2 feasfn, feasbnds and ... arguments

feasfn is an optional user-supplied function that like **respfn** takes a matrix of point coordinates as its first argument, and returns a logical vector of values at those points. TRUE means the point belongs to the feasible region of the response function. Any additional arguments in ... are also passed to this function on each call.

feasbnds is an optional argument that allows simple box constraints on any or all of the input space dimensions to be specified without writing a separate function. In the example of section 2, the constraint $x_1 \geq -2$ could also have been specified by **feasbnds=list(x1=c(-2, Inf))** or **feasbnds=list(x1=c(-2, NA))**. (NA is interpreted as -Inf for lower bounds and Inf for upper bounds.) If both **feasfn** and **feasbnds** are specified, a point is considered feasible only if it satisfies both sets of constraints.

The algorithms in this package assume the feasible region of the response function is a *closed* (although possibly unbounded) set. So, for example, one can impose a non-negativity constraint ($x_1 \geq 0$), but not a strict positivity constraint ($x_1 > 0$).

respfn will never be called with points that are infeasible according to **feasfn** or **feasbnds**. Instead the response function value at those points will be set to NA.

3.1.3 hasgrad and derivmethod arguments

hasgrad is an optional logical scalar that indicates whether **respfn** is able to calculate the gradient of the response function at feasible points. The default is FALSE.

derivmethod is an optional 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 "gradient" option will calculate analytic derivatives. It requires that **respfn** has the ability to calculate the gradient of the response function, and is the default when **hasgrad** is TRUE. See ?fnObj for details. "finite difference" will use a numerical approximation for derivatives.

Note that derivative calculations are entirely optional, and in fact play only a minor role in finding level set boundary points.¹ If the response function is discontinuous, or if one is confident that the level set has only a single part with a fairly smooth boundary, it is quite reasonable to set **derivmethod** to "none" to save some computation.

3.1.4 inptol and resptol arguments

These optional arguments specify numerical tolerances for point coordinates in input space and for response function values, respectively. The idea is that coordinates or response values that differ by no more than the tolerance can be considered equivalent for practical purposes. Tolerances also allow for the fact that computer calculations inevitably introduce roundoff and representation errors. These cause values that

¹Briefly: When the search along a ray finds multiple points where the ray intersects the boundary of the level set, it can check the signs of the derivatives for consistency. Two consecutive intersections with the same sign suggest that there is an intermediate intersection that has been missed.

theoretically should be identical to differ slightly. Without allowing some slack in comparisons of such values, the algorithms in this package will fail.

Tolerance specifications consist of both a relative (or fractional) tolerance applied to the magnitude of a value, and a minimum absolute tolerance, independent of value magnitude. The actual absolute tolerance for a coordinate or response value is the larger of the two. See `?fnObj.character`, `?lsetPkgOpt`, and `?absTol` for more details.

There are default tolerances, which can be seen or changed using the `lsetPkgOpt()` function.

```
str(lsetPkgOpt(c("inptol", "resptol")))
```

```
## List of 2
## $ inptol : num [1:2] 1.00e-06 1.49e-08
## $ resptol: num [1:2] 1.00e-05 1.49e-08
```

3.2 fnSpec class and function

`fnSpec` objects contain a subset of the information in an `fnObj` object, including `inpnames`, `feasbnds`, `inptol` and `resptol`. They are created or extracted from other objects by the `fnSpec()` function. Their main use is to provide a way to specify enough information about an input space to create `inpRect` and `inpRays` objects that are not tied to a specific response function (next two sections).

4 Specifying the search region for a level set

The user must specify a finite region of input space where the search for level set boundary points will be focused. The search region takes the form of a d -dimensional box or hyper-rectangle, represented by objects of class `inpRect`, and created by the `inpRect()` function. See the documentation for the latter for details.

Ideally the search region should be slightly larger than the smallest hyper-rectangle that contains the level set. However if an initial guess at the search region turns out to be too small, too large, or simply in the wrong part of input space, the search can be resumed with an updated search region. Functions `boundingRect()` and `segBoundingRect()` may help with choosing that.

Note that the search region is distinct from the feasible region of the response function. The search region is allowed to include infeasible points. In fact, when the level set touches the boundary of the feasible region, it actually helps the search algorithm if the search region extends beyond the feasible region in that direction.

An example search region for the banana function:

```
rect0 <- inpRect(cbind(c(-3, 3), c(-3, 6)), spec=fobj)
```

5 Specifying search rays

Points on the boundary of a level set are identified by finding where 1-dimensional rays intersect the boundary. Collections of rays, with a shared origin, are represented by `inpRays` objects. As mentioned in the Introduction, points along a ray are identified by their *position* relative to the ray origin (position 0) and second defining point for the ray (position 1). Rays are in principle infinite so there are points at any position value, positive or negative. There is no requirement that the Euclidean distance between positions 0 and 1 be the same for different rays in an `inpRays` object.

Degenerate rays, where the position 0 and position 1 points are equal or nearly equal, are not allowed. Whether two points are considered nearly equal is determined by the numerical tolerances for point coordinates, as described for `inptol` in section 3.

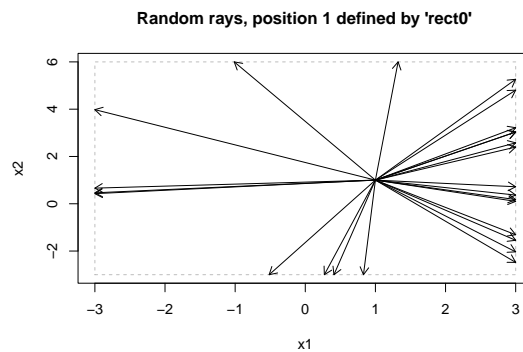
A set of rays can be created with the `inpRays()` function. The `randomRays()` and `axisRays()` functions create rays with specific properties; see their documentation.

5.1 Rays and hyper-rectangles

The ray-creating functions have an optional argument `rect` that accepts an `inpRect` object. If specified, the ray origin must belong to the hyper-rectangle, and positions along each ray are (re)defined so that position 1 is the ray's intersection with the boundary of the hyper-rectangle. This was illustrated in the first figure of the *Introduction* vignette.

The following creates 25 rays with random directions from an origin point (1, 1), and defined so that position 1 for each ray is its intersection with `rect0`:

```
set.seed(1)
rays0 <- randomRays(25, origin=c(1, 1), rect=rect0)
plot(rays0, main="Random rays, position 1 defined by 'rect0'")
```



6 Scaling of input space dimensions

For some problems the 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. In these cases it is important to specify that package algorithms should rescale and/or weight the dimensions when they calculate distances or angles in input space. `inpScale` objects provide a way to specify such scaling. (This is different from the redefinition of position 1 along rays discussed in the previous section.)

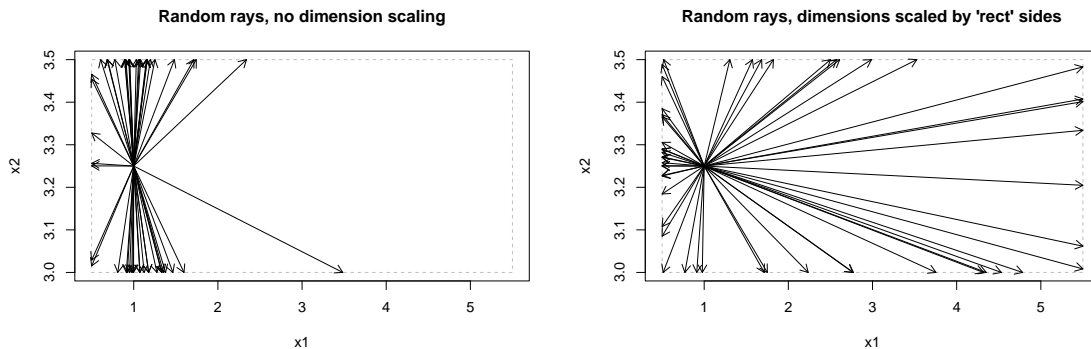
The following example in two dimensions illustrates the issue. Suppose the level set is believed to have a wider range of x_1 values than x_2 values, and the search region reflects this:

```
fspec <- fnSpec(inpnames=c("x1", "x2"))
rect1 <- inpRect(rbind(c(0.5, 3), c(5.5, 3.5)), spec=fspec)
```

By default `randomRays()` creates ray vectors that are uniformly distributed around the unit circle; i.e. it treats both dimensions equally and symmetrically. When such rays are used with an asymmetric search region we can end up undersampling the long dimension of the region, as seen in the first plot below. Scaling each dimension in inverse proportion to the lengths of the sides of the search region gives a better distribution (second plot).

```
set.seed(1)
# Rays with uniformly distributed directions and equal position 1 lengths:
rays_u <- randomRays(50, origin=c(1, 3.25), spec=fspec)
# Position 1 redefined by intersection with a hyper-rectangle:
rays1 <- update(rays_u, rect=rect1)
plot(rays1, main="Random rays, no dimension scaling")
# Specify scaling for input space dimensions:
scl2 <- inpScale(rectSize(rect1), spec=fspec)
# Same rectangle, but ray generation now reflects input dimension scaling:
```

```
set.seed(1)
rays2 <- randomRays(50, origin=c(1, 3.25), scale=scl2, rect=rect1)
plot(rays2, main="Random rays, dimensions scaled by 'rect' sides")
```



Scaling factors should be chosen so that division by them standardizes point coordinates: coordinates for different dimensions should have roughly similar magnitude and spread, preferably with a magnitude not too far from 1.

`inpScale` objects in fact allow for more general types of scaling, such as correlation between input space dimensions. See `?inpScale` for details.

7 Profiling a response function along rays

The `respProfiles()` function allows one to evaluate the response function at an arbitrary set of positions along each of a set of rays. Returning to the banana response function of section 2:

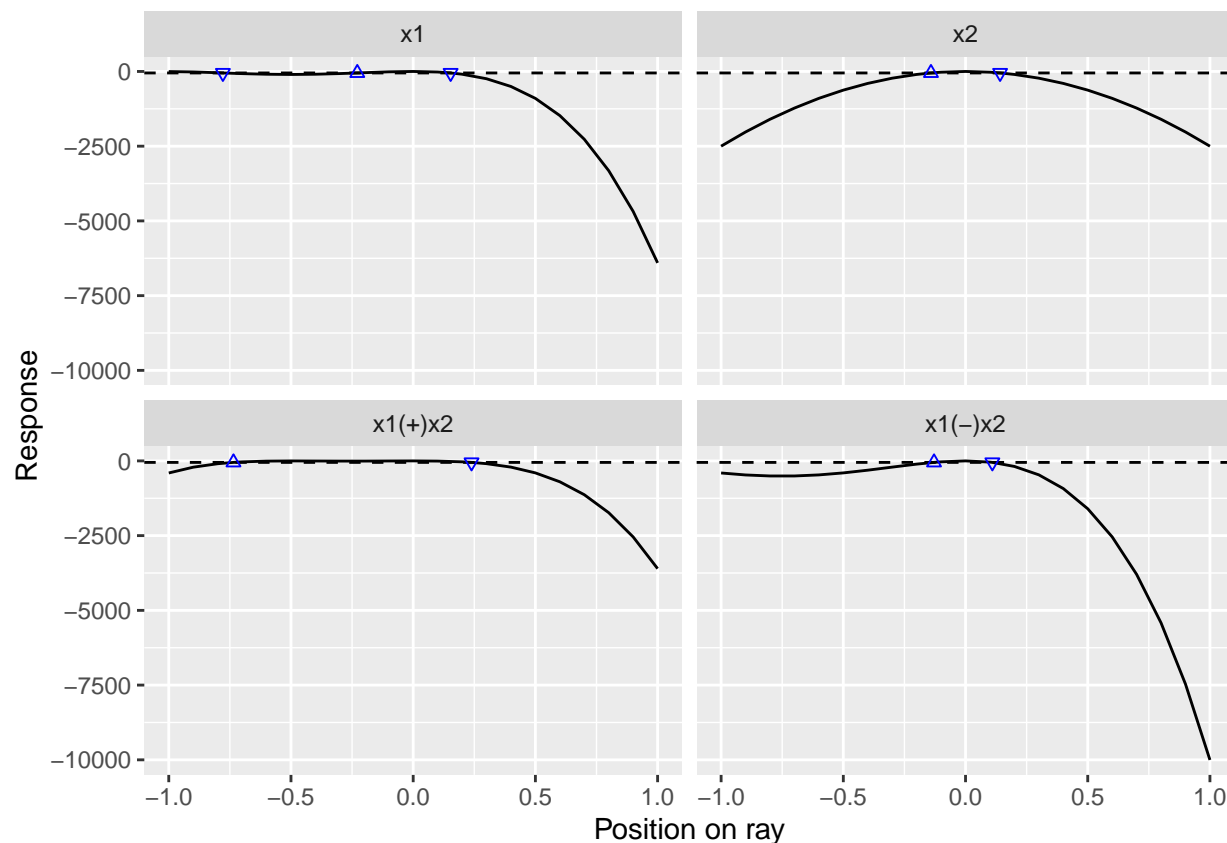
```
rays3 <- axisRays(degree=2, origin=c(1, 1), rect=rect0)
prof3 <- respProfiles(fobj, rays=rays3, lsetthresh=-50, posns=(-10:10)/10)
```

`axisRays()` with `degree=2` generates four rays: one parallel to each of the two coordinate axes, and two parallel to the diagonals with slopes of ± 1 . The `posns` argument to `respProfiles` specifies the evaluation positions (the same positions are used for each ray). The optional `lsetthresh` argument specifies the level set of interest. It triggers a search for level set boundary points along each ray within the range of `posns`. There are `summary` and `plot` methods for the returned `respProfiles` object. (The `plot` method requires the `ggplot2` package.)

```
summary(prof3)
```

```
## Response function profiled along 4 rays in a 2-dimensional input space
## Level set threshold: -50
## Number of profile points over all rays: 93
##   In feasible region= 93 (on its bdry= 0)
##   In level set= 32 (on its bdry= 9)
## Range of response values over all rays: -10004, 0
```

```
if (requireNamespace("ggplot2", quietly=TRUE)) plot(prof3, groupBy=names(rays3))
```



Function `respInfo()` applied to a `respProfiles` object returns a data frame with information about response at each of the positions along each ray. Function `ptCoord()` returns a matrix with the corresponding point coordinates.

```
head(respInfo(prof3))
```

##	resp	feas	lset	line	posn	feasbdry	lsetbdry	deriv	trend
## 1	-4.00000	TRUE	TRUE	1	-1.0000000	NA	NA	8.000	increasing
## 2	-16.20000	TRUE	TRUE	1	-0.9000000	FALSE	FALSE	-223.200	decreasing
## 3	-43.52000	TRUE	TRUE	1	-0.8000000	FALSE	FALSE	-300.800	decreasing
## 4	-49.99996	TRUE	TRUE	1	-0.7784998	FALSE	TRUE	-301.125	decreasing
## 5	-72.52000	TRUE	FALSE	1	-0.7000000	FALSE	FALSE	-263.200	decreasing
## 6	-93.60000	TRUE	FALSE	1	-0.6000000	FALSE	FALSE	-148.800	decreasing

```
head(ptCoord(prof3))
```

##	x1	x2
## [1,]	-1.0000000	1
## [2,]	-0.8000000	1
## [3,]	-0.6000000	1
## [4,]	-0.5569996	1
## [5,]	-0.4000000	1
## [6,]	-0.2000000	1

8 Level set boundary search

The emphasis of `respProfiles()` is evaluation of the response function at a specified set of ray positions. Finding level set boundary points is optional and secondary. When finding boundary points is the primary goal, other functions are available.

8.1 `bdryFromRays()`: Fixed set of rays

This function finds boundary points along a single set of user-supplied rays. It takes as arguments the response function (`fnObj` object), a set of rays (an `inpRays` object, usually with an associated hyper-rectangle defining the search region), and the response threshold that defines the level set of interest (`lsetthresh`). It also has a `control` argument, a list of options to control the search (see `?srchControl` for full details). Of these, the most important are `initPosns` and `initPosns2`.

8.1.1 The search algorithm

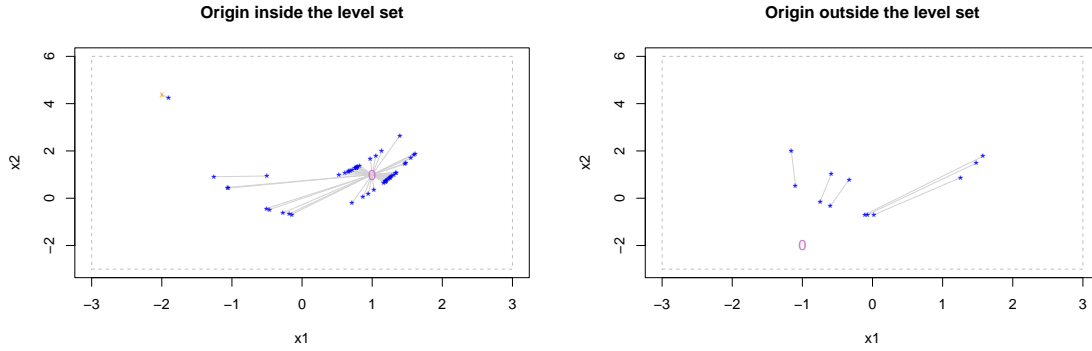
Along a given ray, a boundary of the level set is either (a) a point where the response function transitions from greater than or equal to `lsetthresh` to less than `lsetthresh`, or vice versa; or (b) a point where the ray crosses the boundary of the feasible region of the response function, with a response value 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`.

For a given ray the response function is evaluated at positions specified in `initPosns`. If any of those positions belongs to the level set, in general there will be a non-trivial segment of the ray, containing that position, that lies in the level set. (Recall that it is assumed the level set has non-negligible volume in the input space.) The function searches for the limits of that segment by looking for an `initPosns` position in both directions that is *not* in the level set, and using bisection to find the point(s) between them where the boundary is crossed. If a segment appears to extend beyond the range of `initPosns`, or none of the positions in `initPosns` belongs to the level set, positions in `initPosns2` are added.

8.1.2 `lsetSegs` objects

This approach to searching for level set boundary points yields not just points but *line segments* that belong to the level set, with boundary points as the segment endpoints. That is the form in which `bdryFromRays()` returns its result: an object of class `lsetSegs`. There are `summary` and `plot` methods for these objects as shown in the example in the *Introduction* vignette. Information about segments can be extracted with the `segInfo()` function, and about segment endpoints with the `respInfo()` and `ptCoord()` functions. Segment validity can be checked with the `lsetSegsCheck()` function.

It is a huge help to the boundary search if the ray origin is a point inside the level set. In that case every ray is guaranteed to have a segment belonging to the level set. Ray origins that are outside the level set are allowed, but there is a chance that few (or even no) rays will intersect the set, yielding little information about it. The following two figures illustrate this for the banana response function. Twenty-four random rays are used in both cases, but only 6 yield level set segments when the origin (plotted as 0) is outside the level set.

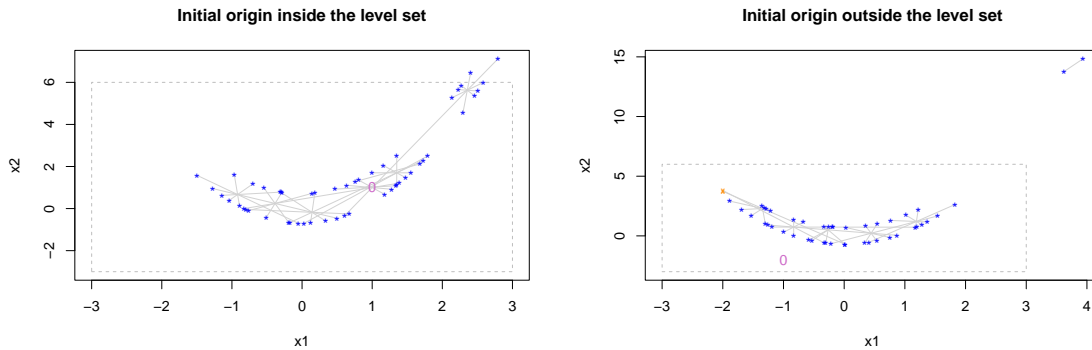


For simple, convex level sets, with a ray origin inside the level set, the default `initPosns` and `initPosns2` will often be adequate. However when the level set has a more complicated shape or multiple parts, or the ray origin is not in the level set, a more closely-spaced set of initial positions may be necessary for informative results. See `?bdryFromRays` and `?srchControl` for details.

8.2 `bdrySearch()`: Adaptive selection of rays

In some cases more information about the boundary of a level set can be obtained by using multiple sets of rays, with different origins, than by using the same total number of rays with a single origin. The function `bdrySearch()` performs an adaptive boundary search. At each step a new ray origin is selected that attempts to explore portions of the boundary that have not been well-sampled at previous steps. The user can control the number of steps and the number of rays used per step. See `?bdrySearch` for details.

The following plots show `bdrySearch()` results for the banana function. There are 24 total rays as in the previous subsection, but now distributed over 6 adaptively selected origins. Results are more informative about the overall shape and extent of the level set. See the *Example* vignettes for more examples.



8.3 Slices of input space

When the input space has more than two dimensions, it may be useful to explore the boundary of the level set separately within lower-dimensional *slices* of the space. A slice is just an axis-aligned hyperplane within the input space, which is equivalent to holding a subset of the d coordinates at fixed values while the other coordinates are free to vary. Therefore a set of k slices can be represented as a matrix with k rows and d columns, where each row contains specific values for the fixed coordinates and `NA` for the free coordinates. For example in a 3-dimensional input space

```
slices <- rbind(c(1, NA, NA), c(2, NA, 3), c(NA, NA, NA))
```

represents three slices. The first is a 2-dimensional slice consisting of all points with $x_1 = 1$, the second is a 1-dimensional slice (i.e., a line) consisting of all points with $x_1 = 2$ and $x_3 = 3$, and the third is the *identity*

slice that allows all coordinates to vary and so is equivalent to the whole input space. See `?sliceMat` for more information about creating slices.

Functions `slicedBdryFromRays()` and `slicedBdrySearch()` are extensions of `bdryFromRays()` and `bdrySearch()` respectively, that carry out level set boundary searches separately for one or more slices. See their help pages for more information, and the *Example* vignettes for illustrations of their use.