Introduction to the **levelSets** Package

Richard Raubertas

31 December 2025

A level set of a function $f(x)$ is the set of input points x for which the function value is greater than or equal to a specified threshold t[1]:

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

Applications of level sets include confidence or credible regions for parameters of statistical models (where x is a vector of parameter values and f is the likelihood or posterior density); regions where classification rules assign high probability to a given class (x contains feature values for an item and f is the class probability); and scientific or engineering models of physical phenomena (where one is interested in input regions where model output is above a threshold).

To avoid confusion with `R` functions, f will be referred to as the *response function*. Its inputs x are vectors with d elements, so are points in d-dimensional *input space*. Often the valid inputs to f are only a subset of input space, and that subset is called the *feasible region* for f.

This package maps out a level set by finding its intersection 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. The use of rays to map out the boundary of confidence regions for parameters of statistical models was proposed by Kim and Lindsay (2011). See the vignette *levelSets Example: Confidence Regions* for an example of that application. This package generalizes and implements Kim and Lindsay's idea.

The package provides tools to generate rays, find intersections, and visualize results. It makes few assumptions about f: it can be discontinuous, it may have a complicated feasible region, and the target level set may be non-convex or have multiple, disconnected parts. (Note that because f can be discontinuous, boundary points of a level set do not necessarily satisfy $f(x) = t$.) However it does assume that level sets of f have non-zero d-dimensional volume. For example in $d = 2$ dimensions, level sets should occupy a non-zero area of the (x_1, x_2) plane, rather than being a 1-dimensional line or curve.

An example

To illustrate the idea of using rays to map out a level set, we examine a simple response function with a 2-dimensional input space. Other examples that illustrate a wider range of package features are in separate vignettes.

The response function is

$$f(x_1, x_2) = \begin{cases} -(x_1^2 + 2x_1x_2 + 2x_2^2) & \text{if} \quad x_1 + 2x_2 < 15 \\ -((x_1 - 8)^2 + 2(x_1 - 8)(x_2 - 8) + 2(x_2 - 8)^2) & \text{if} \quad x_1 + 2x_2 \geq 15 \end{cases}$$

The response surface has two hills with elliptical contours. Although this function is well-defined for any (x_1, x_2), for illustration we define the feasible region to be $\{x : x_1 \geq 0\}$. The goal is to map out the level set where $f \geq -40$.

The response and feasibility functions can be coded in `R` as

[1] Terminology is not fully standardized. This definition is also referred to as an *upper* level set or an excursion set. For brevity the term level set is used throughout this package.

```
respf <- function(x) {
     c1 <- x[, 1]^2 + 2*x[, 1]*x[, 2] + 2*(x[, 2]^2)
     c2 <- (x[, 1]-8)^2 + 2*(x[, 1]-8)*(x[, 2]-8) + 2*((x[, 2]-8)^2)
     resp <- ifelse(x[, 1] + 2*x[, 2] < 15, -1*c1, -1*c2)
     resp
}
feasf <- function(x) {
     (x[, 1] >= 0)
}
```

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 response value, and TRUE or FALSE, respectively.

These functions are wrapped in an `fnObj` object that also contains specifications for the input space and possibly other specifications and arguments (see `?fnObj`).

```
library(levelSets)
fobj <- fnObj(c("X1", "X2"), respfn=respf, feasfn=feasf)
```

If we just wish to evaluate the response function at a set of points, the `respInfo` function will do that. It returns a data frame with one row per point. Column `feas` reports whether the point is feasible and column `lset` whether the point is in the level set.

```
pts <- rbind(c(2, 2), c(4, -1), c(-1, 4), c(3, 5))
respInfo(pts, fnobj=fobj, lsetthresh=-40)
```

```
##   resp  feas  lset
## 1  -20  TRUE  TRUE
## 2  -10  TRUE  TRUE
## 3   NA FALSE FALSE
## 4  -89  TRUE FALSE
```

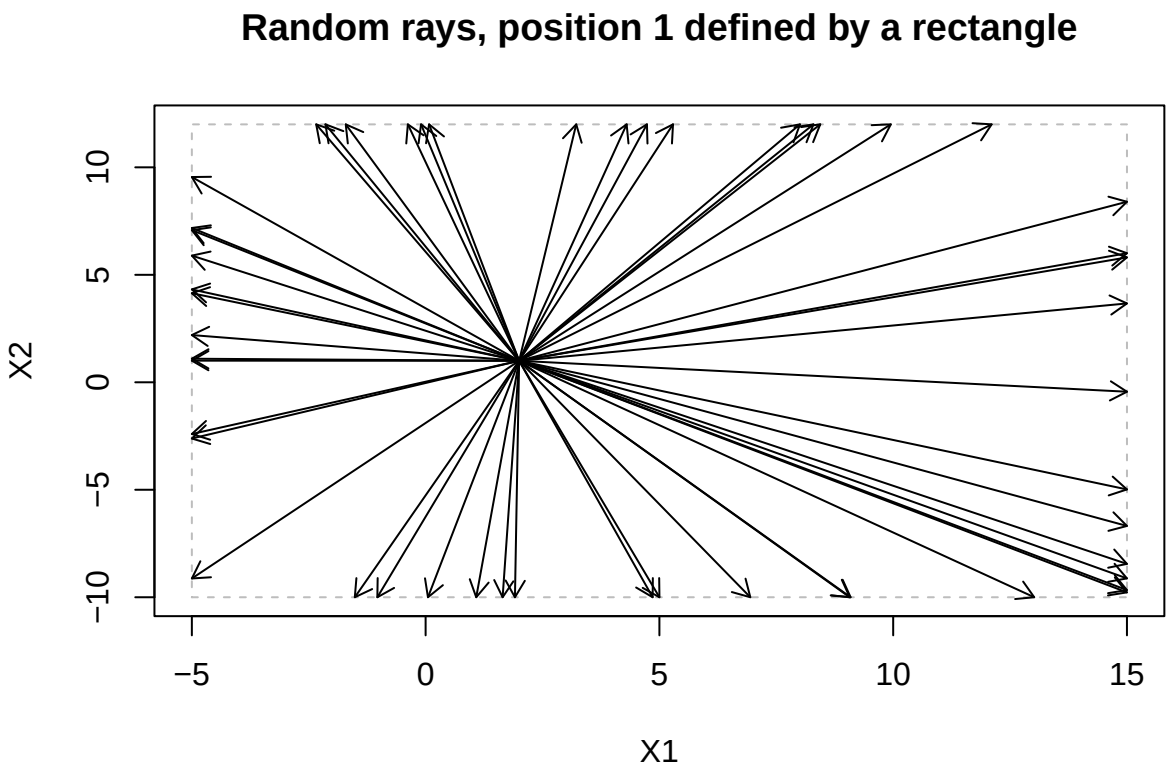
Because `fnObj` objects are aware of the feasible region of the response function, they ensure that the latter is never called with an infeasible point, such as `pts[3, ]` here. The response value is automatically set to `NA` for such points.

For a level set boundary search we need to specify a region of input space where the search will be focused. Search regions are represented by `inpRect` objects, which are axis-aligned hyper-rectangles (d-dimensional "boxes"). They are created by the `inpRect` function by specifying the coordinates of the lower and upper corners of the box.

```
rect1 <- inpRect(rbind(c(-5, -10), c(15, 12)), spec=fobj)
```

And finally we create a set of rays along which boundary intersections will be searched for.

```
set.seed(1)
rays1 <- randomRays(50, origin=c(2, 1), rect=rect1)
plot(rays1, main="Random rays, position 1 defined by a rectangle")
```



Because of the key role played by rays in this package, it is important to understand how they are parameterized. A set of rays is represented by an `inpRays` object. All rays in the object have the same *origin* point, which defines *position* 0 along the rays. The direction of a ray is defined by a second, distinct point that defines position 1 along that ray. The position of any other point on the ray is defined by linear 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.)

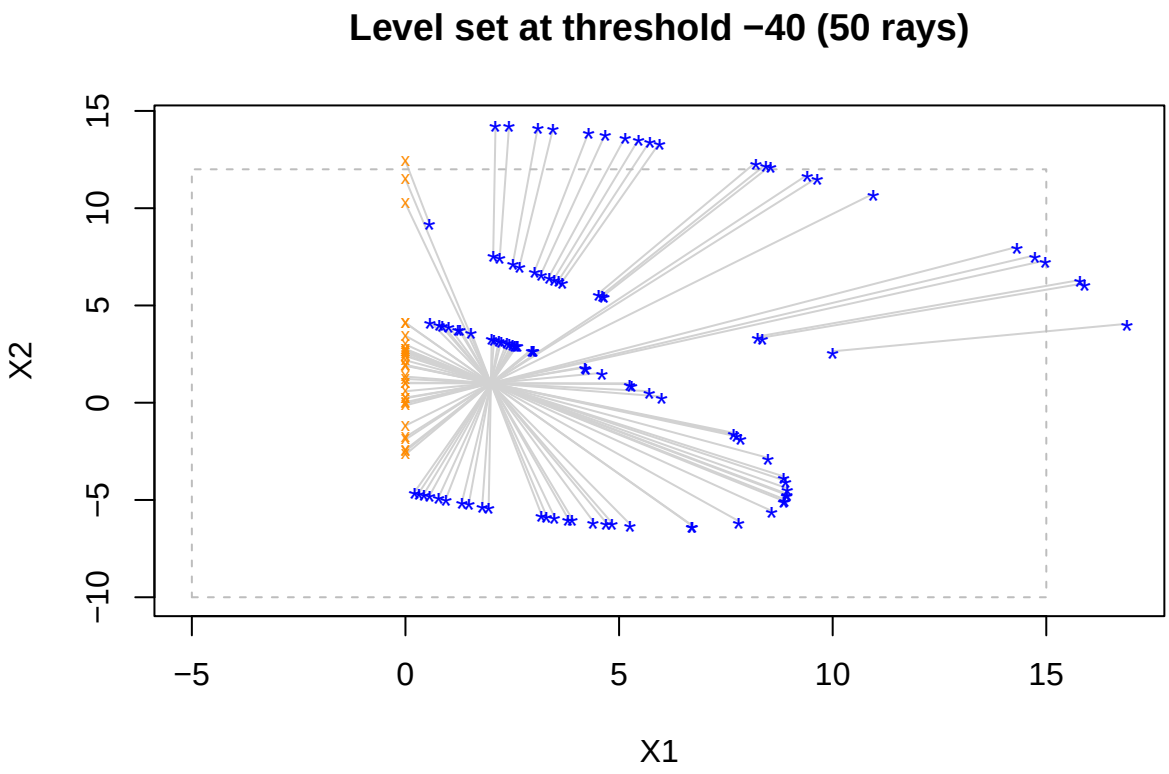
Note that there is no requirement that the Euclidean distance between positions 0 and 1 be the same for all rays, and thus any given position value may represent different Euclidean distances from the origin for different rays. In particular, when the `rect` argument is used when creating an `inpRays` object, position 1 for each ray is defined as its intersection with the boundary of that hyper-rectangle, as for `rays1` above. This turns out to be useful and convenient for level set boundary searches with a search region defined by `rect`.

The function `bdryFromRays` finds intersections of rays with the level set.

```
segs1 <- bdryFromRays(fobj, rays=rays1, lsetthresh=-40)
```

It returns an object with class `lsetSegs`. These objects actually contain *pairs* of level set boundary points. The points in each pair lie along the same ray, and the search algorithm is designed to find point pairs for which the entire line *segment* connecting the pair belongs to the level set. (Such segments are sometimes called *chords* of the level set.) There are `plot` and `summary` methods for `lsetSegs` objects.

```
plot(segs1, rect=rect1, main="Level set at threshold -40 (50 rays)")
```




```
summary(segs1)
```

```
## Level set segments along 50 rays in a 2-dimensional input space
## Level set threshold: -40
## Number of segments over all rays: 67
##   Segment endpoints by type:
##     nfeasbdry nlsetbdry     ncens
##            34       100         0
##   Number of rays with 0, 1, >1 segments: 0, 33, 17
## Bounding box for level set segments:
##                   X1        X2
##   [1,] 1.459264e-08 -6.312632
##   [2,] 1.688813e+01 14.310640
##   Box volume= 348.2885, # of intersecting rays= 50
```

This example shows how ray-boundary intersections can outline the shape and size of a level set. There are several things to note.

- 34 segments extend to the boundary of the feasible region ($x_1 = 0$). Segment endpoints on the feasible region boundary are plotted with a different color and shape.
- 17 rays had more than one level set segment found. Whenever there are two or more segments along a single ray, it indicates that the level set is not convex and/or has multiple parts.
- However segments along a number of other rays cross the gap between the two ellipses. This leads us to suspect that they are *invalid segments*: The claim that all points on the segment connecting two boundary points also belong to the level set appears to be false. Checking and correcting this is described below.
- Default parameters for the search algorithm allow the search to go modestly beyond the specified search

region `rect1`, as indicated by segments that extend outside the dashed gray rectangle. This can be controlled by the `initPosns` and `initPosns2` control parameters (see `?srchControl`).

The function `lsetSegsCheck` checks the validity of segments and attempts to fix invalid ones by splitting them into two or more shorter, valid segments.

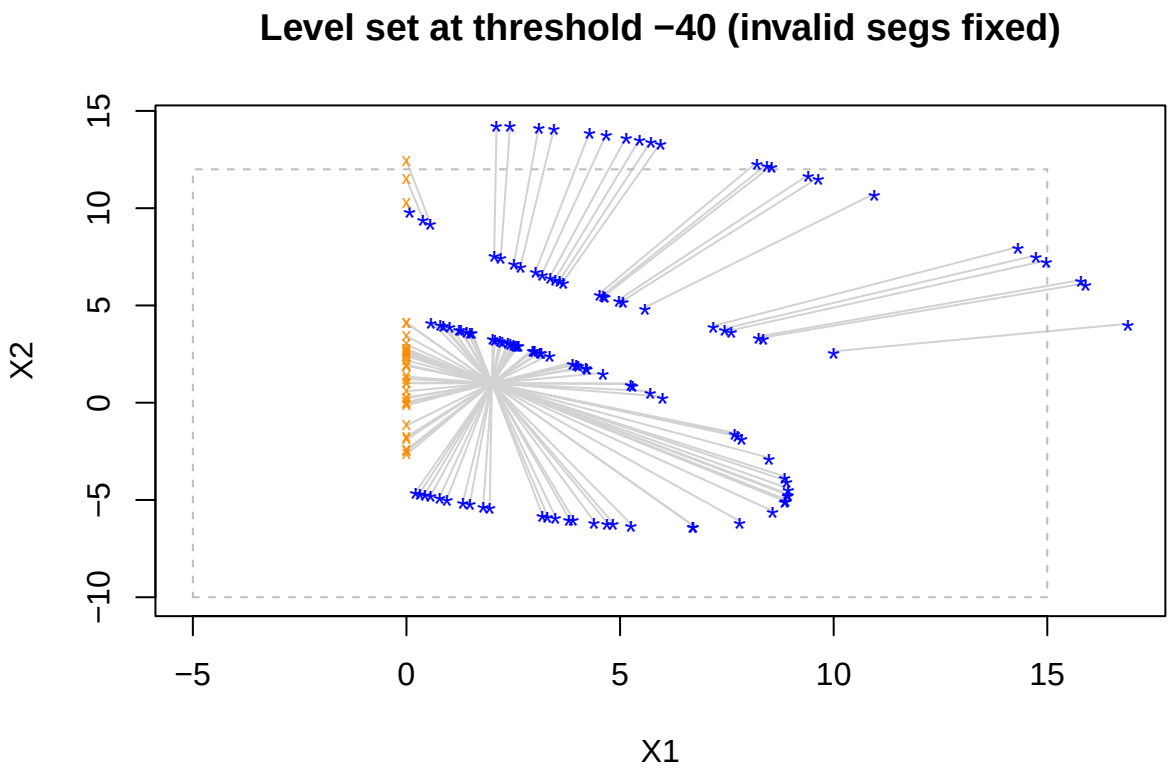
```
chk1 <- lsetSegsCheck(segs1, fnobj=fobj, posns=(1:4)/5)
table(chk1$validIn)
```

```
##
## FALSE  TRUE
##     8    59
```

```
table(chk1$validOut)
```

```
##
## TRUE
##   75
```

```
segs1 <- lsetSegs(chk1)  # all TRUE, so update 'segs1'
plot(segs1, rect=rect1, main="Level set at threshold -40 (invalid segs fixed)")
```

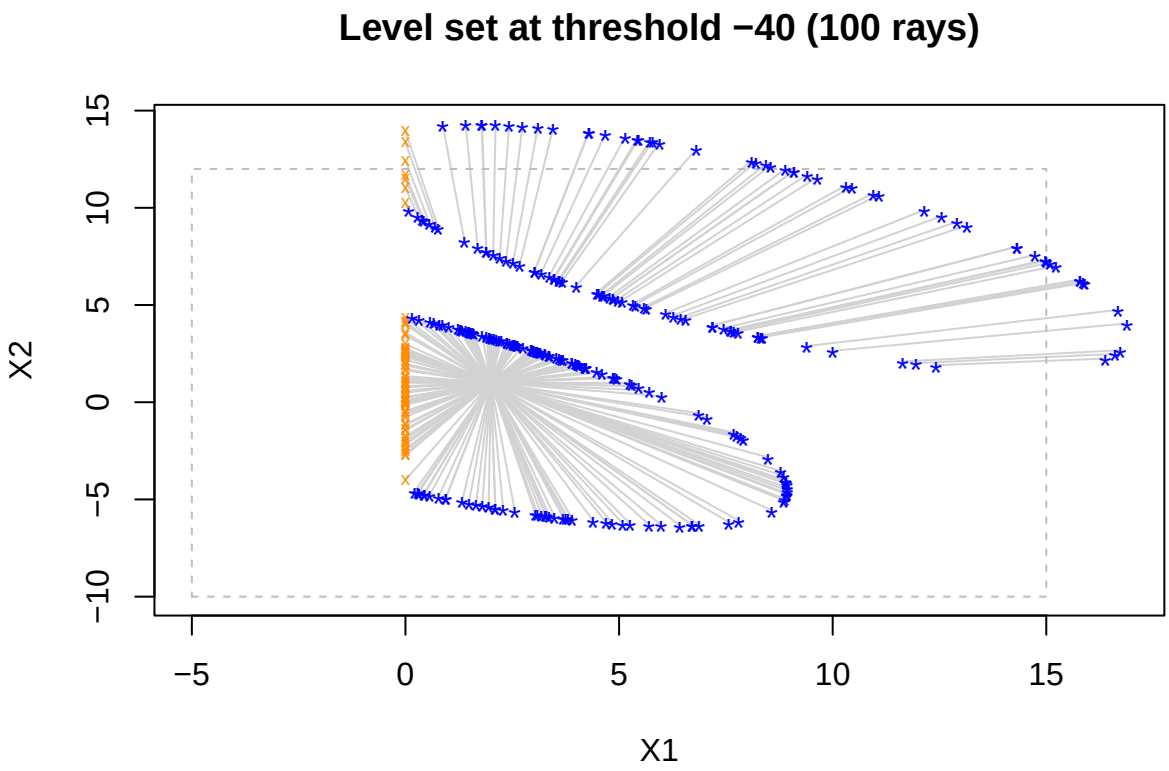


Increasing the number of rays will give a clearer picture of the level set, at the cost of more computation. We can generate another set of rays and their level set intersections

```
rays2 <- randomRays(50, origin=c(2, 1), rect=rect1)
segs2 <- bdryFromRays(fobj, rays=rays2, lsetthresh=-40,
                      control=list(initPosns=(0:5)/5))
```

and combine them with the first

```
segs12 <- c(segs1, segs2)  # 'c' method for 'lsetSegs' objects
plot(segs12, rect=rect1, main="Level set at threshold -40 (100 rays)")
```




```
summary(segs12)
```

```
## Level set segments along 100 rays in a 2-dimensional input space
## Level set threshold: -40
## Number of segments over all rays: 159
##   Segment endpoints by type:
##     nfeasbdry nlsetbdry     ncens
##            71       247         0
##   Number of rays with 0, 1, >1 segments: 0, 41, 59
## Bounding box for level set segments:
##                   X1       X2
##   [1,] 5.482811e-09 -6.32386
##   [2,] 1.688813e+01 14.32369
##   Box volume= 348.6985, # of intersecting rays= 100
```

Main search functions

There are four main functions in the package to find level set boundary points and segments.

- `bdryFromRays()`. The user defines a single set of rays (origin, directions, and, implicitly or explicitly, an associated search region). A search is made for level set segments along those rays in that region. Not every ray needs to intersect the level set, and in fact the search might not find any segments. But when the user has a good idea of the location of the level set and it does not have a complicated shape,

this approach is simple and works well.

- `bdrySearch()`. An algorithm adaptively selects multiple ray origins in an attempt to explore more of the level set than a single set of rays can. Useful when the level set has a complicated shape.
- `slicedBdryFromRays()`, `slicedBdrySearch()`. When the input space has more than two dimensions, it may be useful to determine boundary points and segments within lower-dimensional *slices* of input space. A slice is just a (hyper)plane in which a subset of the inputs are held at fixed values. These functions apply the algorithms from `bdryFromRays` and `bdrySearch`, respectively, to each slice separately.

In addition there is a function `respProfiles()` that simply evaluates the response function at a fixed set of positions along a user-specified set of rays.

The vignette *Problem Setup* goes into more detail about how to use this package. There are also two vignettes with extended examples of how to use the package for different types of level set problems.

References

Kim, Daeyoung and Lindsay, Bruce G. 2011. Using confidence distribution sampling to visualize confidence sets. *Statistica Sinica* 21:923–948.