Modelling 3D utilization distributions for acoustic telemetry data in R

Eneko Aspillaga, Kamran Safi, Bernat Hereu, and Frederic Bartumeus

May 2018

Introduction

This document is a step-by-step guide to estimate 3D utilization distributions (3D-UDs) from passive acoustic telemetry data in `R`, following the new numerical method proposed in Aspillaga *et al.* (2018) and implemented in the `fishtrack3d` package. The main novelty of this method is that it takes into account the detection probability around receivers, which is empirically determined, and integrates the depth information from transmitters and the local topography. The method simulates large numbers of stochastic trajectories (synthetic paths) that are then assembled to estimate space use probabilities. All the procedure explained here is carried out with an openly available sample dataset.

Sample dataset

All the neccesary `R` functions and a sample dataset are distributed together with the `fishtrack3d` package (https://github.com/aspillaga/fishtrack3d). The last version of the package can be installed with the following code:

```r
library(devtools)
github_install("aspillaga/fishtrack3d")
```

The following data objects are provided within the package:

- **tracking:** Passive acoustic telemetry data from two common dentex (*Dentex dentex*) individuals ('dentex18' and 'dentex43') in the Medes Islands marine protected area (Catalonia, NW Mediterranean Sea). The tracking period took place from 01/Oct/2007 to 31/Dec/2007. Movements were tracked with a fixed array of 17 acoustic receivers.

- **receivers:** Geographical coordinates of the 17 acoustic receivers used in the acoustic telemetry study (see image below). Coordinates are in UTM, referred to the datum WGS84 zone 31N.

- **bathymetry:** Bathymetric raster map (class `RasterLayer`) of the study area (see image below). It has a resolution of 10x10 m and a total extension of 221x221 cells. Land areas are denoted by `NA` values. The coordinates are in UTM, referred to the datum WGS84 zone 31N.

- **range_test:** Data from an acoustic range test performed in the study area. The data frame contains the hourly percentages of signals detected by receivers placed at different distances from one acoustic transmitter during a 24 h period.

```
library(fishtrack3d)
library(raster)
library(plyr)
library(ks)

plot(bathymetry, col = terrain.colors(100))
points(receivers$long.utm, receivers$lat.utm, pch = 16, cex = 0.8)
text(receivers$long.utm, receivers$lat.utm, labels = receivers$id, pos = 4)
```
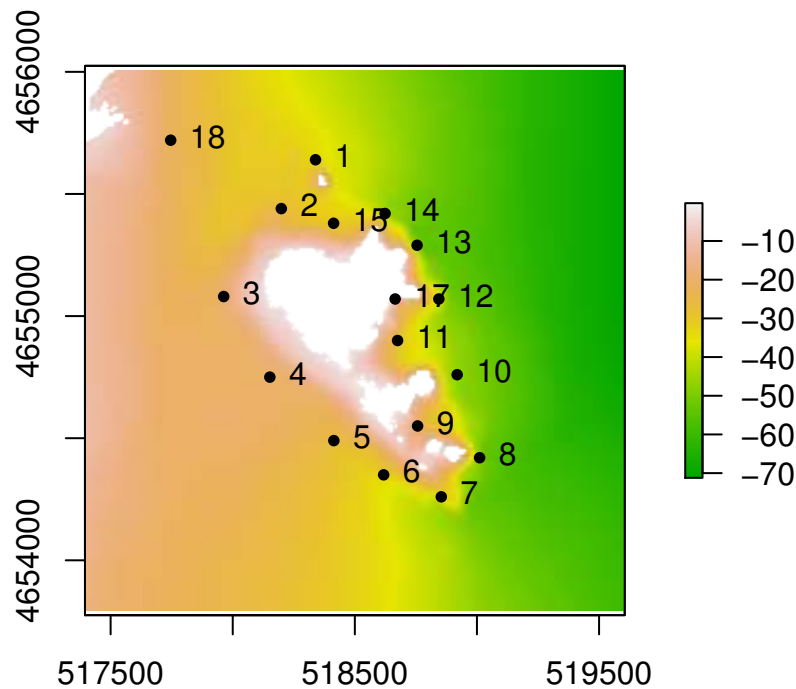

Some of the steps of the method are time consuming, so they obstruct a quick run through the vignette. If you just want to have a general overview, you can skip these steps by directly loading their outputs, which can be downloaded in the following file:

```
path <- "https://github.com/aspillaga/fishtrack3d/raw/master/vignettes/data.zip"

# Download and unzip the file in the working directory
download.file(path, destfile = "/data.zip")
unzip("./data.zip", exdir = "./data", junkpaths = TRUE)
```

General overview of the method

The process to estimate 3D-UDs is divided in three sequential steps:

1. **Empirical characterization of detection probabilities:** Data from an acoustic range test and the local topography are used to characterize the spatial probability of being detected by each receiver.

2. **Simulation of synthetic path realizations:** A large number of stochastic trajectories are simulated from the acoustic telemetry data, taking into account the characterization of the acoustic performance, the depth values, and the local topography.

3. **Assemblage of utilization distribution volumes:** 3D-UDs are estimated from the spatial occurrence of all the synthetic path realizations within a 3D grid in which the study site has been divided.

Step 1: Empirical characterization of detection probabilities

In this step, we are going to characterize the acoustic performance of the receiver array, by modelling the detection probability depending on the distance to the receiver and detecting acoustic shadow areas using the viewshed analysis.

1.1. Modelling the acoustic range

The probability of detecting an acoustic signal decreases when increasing the distance between the transmitter and the receiver. This is because acoustic signals gradually lose their power while travelling through water, making them more susceptible to be interferred by environmental noise, and finally becoming undetectable for acoustic sensors. There is a critical distance, known as the detection range, at which most of the emitted signals are no longer detected by receivers. Assesing the average detection range is highly important to design a receiver array before the experiment, but also for the posterior interpretation of the data. This range is often tested by placing receivers at increasing distances from a receiver and calculating the ratio of signals that arrives to each distance.

The `range_test` data frame contains data from one range test performed in the study site in 2013. The data frame represents the ratio of signals detected (`det.ratio`) by receivers (`rec.id`) at different distances (`dist.m`) during one hour intervals.

```
head(range_test)
>               date.time tag.id rec.id dist.m rec.depth.m detected emitted
> 1 2013-09-04 13:00:00    534   7423      0          10       40      40
> 2 2013-09-04 14:00:00    534   7423      0          10       46      46
> 3 2013-09-04 15:00:00    534   7423      0          10       46      46
> 4 2013-09-04 16:00:00    534   7423      0          10       45      45
> 5 2013-09-04 17:00:00    534   7423      0          10       41      41
> 6 2013-09-04 18:00:00    534   7423      0          10       40      40
>   det.ratio
> 1         1
> 2         1
> 3         1
> 4         1
> 5         1
> 6         1
```

We adjust a logistic regression model to get the probability distribution of being detected as function of the distance.

```
range.mod <- glm(det.ratio ~ dist.m, data = range_test,
                 family = quasibinomial(logit))
```

We can plot the data to see the fit of the function:

```
plot(det.ratio ~ dist.m, data = range_test, xlim = c(0, 400),
     ylab = "Percentage of detections",
     xlab = "Distance from receiver (m)")
lines(1:400, predict(range.mod, data.frame(dist.m = 1:400), type = "response"),
      type = "l", col = "firebrick", lwd = 2)
```
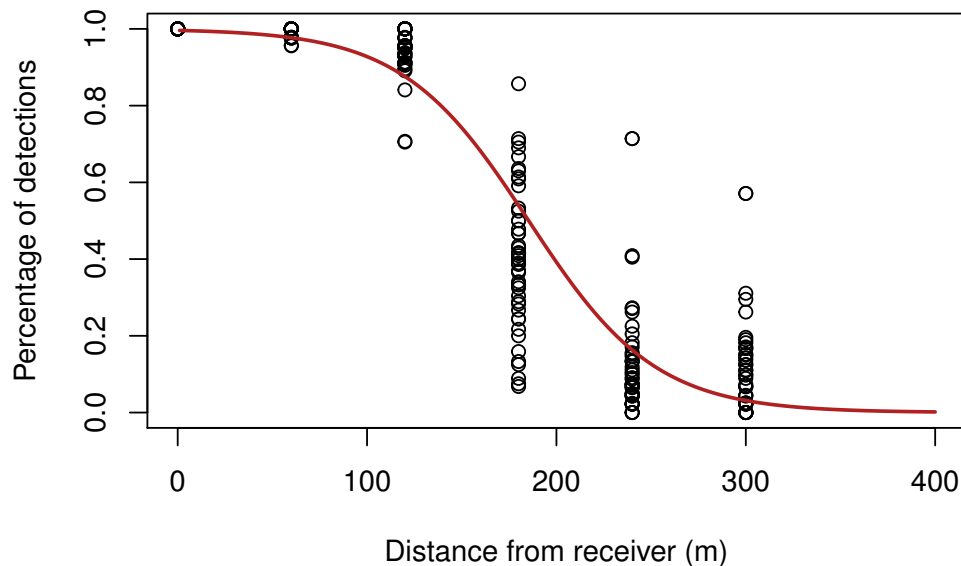

1.2. Acoustic shadows

In this step, we are going to characterize acoustic shadows, i.e. areas from which acoustic signals can not be detected by a receiver, within our receiver array. Acoustic shadows are generated by the presence of prominent physical obstacles in the study area, which impede the signal transmission between transmitters and receivers. Here, we will identify the most evident acoustic shadows: those hindered by emerged landmasses. This is done by applying a *viewshed* analysis to the `bathymetry` raster. The viewshed analysis determines the areas of the terrain that are visible from a specific location, but it is also applicable to the propagation of acoustic signals in water (in our case, the specific locations are the positions of the receivers). As we are interested in only excluding the acoustic shadows generated by emerged landmasses, we will magnify the height of land cells in the `bathymetry` raster before computing the viewshed.

At the same time, we will calculate the lineal distance (in 2D) between each receiver and all the raster cells outside acoustic shadows. This distance will be used later, (during the simulation of synthetic paths in *Step 2.2*), together with the acoustic range model (created in *Step 1.1*) to obtain the probabilities of being detected around each of the receivers.

NOTE: Setting GRASS to work within R might not be very straightforward. You can skip this step if you are following this vignette with the example dataset, as the result of this step, the object `viewshed`, is already provided in the **fishtrack3d** package. But if you want to use your own telemetry and topography data, I encourage you (and wish you good luck) to make R, GRASS and **rgeos** understand each other!

```r
library(rgrass7)

# In the bathymetry raster, NA values correspond to emerged areas.
# We will exaggerate the height of these areas to make sure
# everything behind them is removed in the viewshed analysis
elevation <- bathymetry
elevation[is.na(elevation)] <- 1e+10
elevation <- as(elevation, "SpatialPixelsDataFrame")

# Initialize GRASS session
initGRASS("/Applications/GRASS-7.0.app/Contents/MacOS/",
          home = tempdir(), override = TRUE)

# Load the elevation raster in GRASS
writeRAST(x = elevation, vname = "elevation", overwrite = TRUE)

# Set the region for the analysis
execGRASS("g.region", parameters = list(raster = "elevation"))

# Loop for each receiver
viewshed <- lapply(receivers$id, function(i) {

  # Coordinates of the receivers
  coord <- as.numeric(receivers[receivers$id == i, 2:3])

  # Execute the 'viewshed' analysis
  execGRASS("r.viewshed", flags = c("overwrite", "b", "quiet"),
            parameters = list(input = "elevation",
                              output = "viewshed",
                              coordinates = coord,
                              target_elevation = 500))

  # Export raster from GRASS and assign NA values and a projection
  rast.tmp <- raster(readRAST("viewshed"))
  proj4string(rast.tmp) <- proj4string(elevation)
  rast.tmp[rast.tmp == 0 | is.na(bathymetry)] <- NA


  # Calculate distances from receivers to each raster cell
  distances <- sqrt((coordinates(rast.tmp)[, 1] - coord[1])^2 +
                    (coordinates(rast.tmp)[, 2] - coord[2])^2)
  rast.tmp[!is.na(rast.tmp)] <- distances[!is.na(values(rast.tmp))]

  return(rast.tmp)
})

raster::names(viewshed) <- raster::validNames(receivers$id)
viewshed <- raster::stack(viewshed)
```
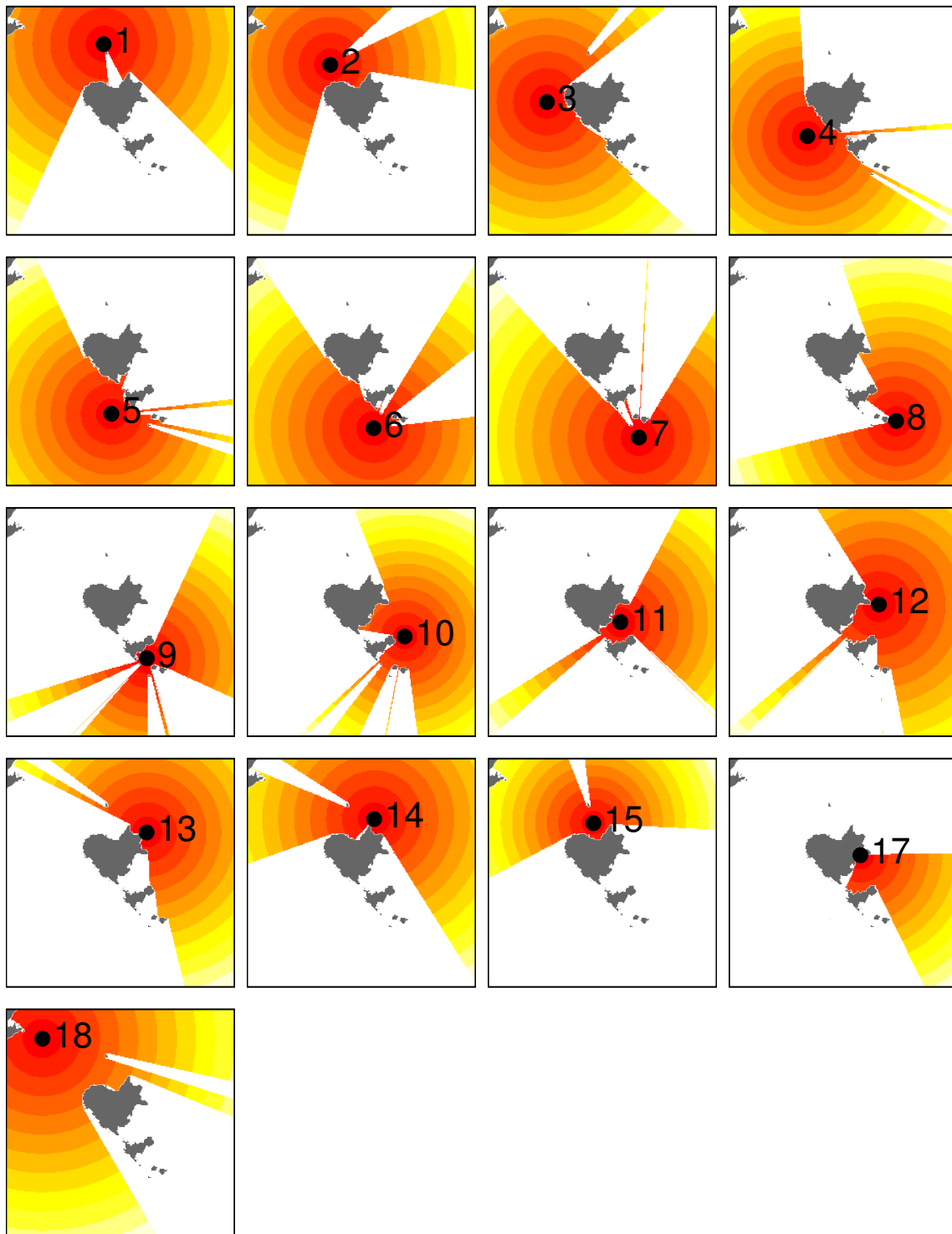
This is how the viewshed analysis looks like for each receiver. The yellowish the colour, the greater the distance from the receiver. White parts of the plots represent acoustic shadows caused by emerged landmasses (in grey):

Step 2: Simulation of synthetic path realizations

The object `tracking` contains acoustic telemetry data of two common dentex (*Dentex dentex*) individuals in the Medes Islands marine protected area. The object is a data frame where each row corresponds to one signal. The `rec.id` column indicates the ID of the receiver that detected each signal, and the `tag.id` column the ID of the detected individual (in this case there are two individuals, `dentex18` and `dentex43`). `time.stamp` and `depth` columns indicate, respetively, the date and time (in UTC) and the depth of the fish when the detection occurred. In total, the data frame contains 61,528 detections that took place between October 1 and December 31 2007. For this experiment, transmitters were programmed to emit signals at random intervals between 80 and 180 seconds (to reduce the probability of collision between signals).

```
head(tracking)
>       tag.id          time.stamp rec.id depth
> 1 dentex18 2007-10-01 00:03:03      9   10.6
> 2 dentex18 2007-10-01 00:04:42      9   10.6
> 3 dentex18 2007-10-01 00:06:36      9   11.4
> 4 dentex18 2007-10-01 00:09:02      9   11.0
> 5 dentex18 2007-10-01 00:12:57      9   11.0
> 6 dentex18 2007-10-01 00:15:50     10   11.4


# We will split the data into a list to make it easier to apply the
# method to both individuals at the same time
tracking.list <- split(tracking, tracking$tag.id)
names(tracking.list)
> [1] "dentex18" "dentex43"
```

2.1 Thin telemetry data

In this first step, we will thin acoustic telemetry data into 30-minute intervals using the `thinData` function. This function commits a receiver to each time interval, randomly sampled according to the proportion of signals detected by each receiver, and assigns a depth value sampled from the probability distribution of the depths measured within that interval. This procedure allows to reduce the number of data to be processed (and hence, the computation time), while maintaining the variability of the original data.

```
t30min <- lapply(tracking.list, thinData, time.int = "30min")


# We thin the data twice to later check the variability
t30min.2 <- lapply(tracking.list, thinData, time.int = "30min")


head(t30min[["dentex18"]])
>       tag.id rec.id          time.stamp depth
> 1 dentex18     10 2007-10-01 00:00:00   11.5
> 2 dentex18      9 2007-10-01 00:30:00   11.2
> 3 dentex18      9 2007-10-01 01:00:00   10.2
> 4 dentex18      9 2007-10-01 01:30:00   11.4
> 5 dentex18      9 2007-10-01 02:00:00   11.9
> 6 dentex18      9 2007-10-01 02:30:00   12.1
```

```
head(t30min.2[["dentex18"]])
>     tag.id rec.id           time.stamp depth
> 1 dentex18      9 2007-10-01 00:00:00  11.2
> 2 dentex18      9 2007-10-01 00:30:00  11.3
> 3 dentex18      9 2007-10-01 01:00:00  10.9
> 4 dentex18      9 2007-10-01 01:30:00  12.4
> 5 dentex18      9 2007-10-01 02:00:00  11.9
> 6 dentex18      9 2007-10-01 02:30:00  12.4
```

We can make some plots to compare the receivers in two different thinned data frames.

```
par(mfrow = c(2, 1), mar = c(3.1, 4.1, 3.1, 5.1))
for (i in names(tracking.list)) {
  t30.tmp <- list(t30min[[i]], t30min.2[[i]])
  for(x in 1:length(t30.tmp)) {
    spatChronPlot(time.stamp = t30.tmp[[x]]$time.stamp,
                  rec.id = factor(t30.tmp[[x]]$rec.id, levels = receivers$id))
    title(main = paste(i, "- Simulation", x))
  }
}
```

dentex43 - Simulation 1

dentex43 - Simulation 2
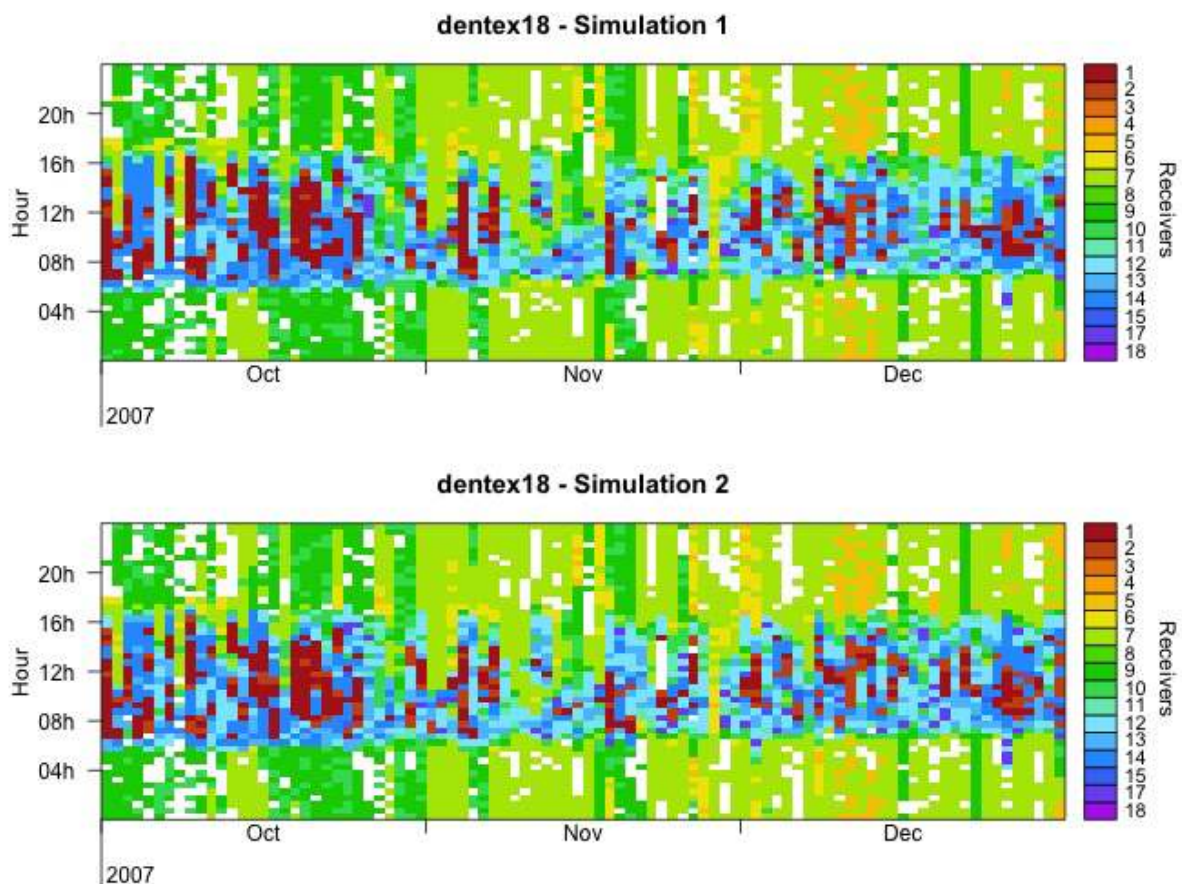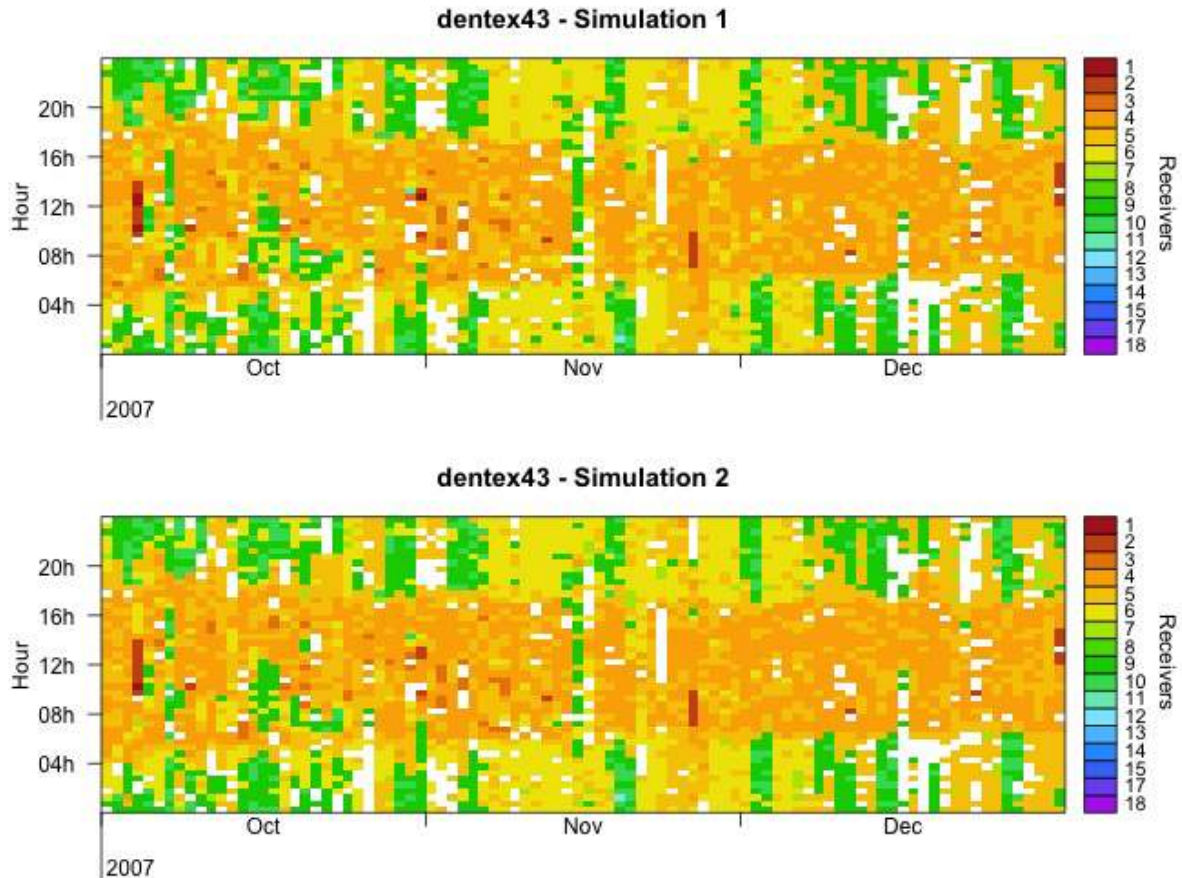
We can also see the comparison between depth values from the two thinned data frames.

```r
par(mar = c(3.1, 4.6, 2.1, 2.1))
for (i in names(tracking.list)) {
  plot(t30min[[i]]$time.stamp, -t30min[[i]]$depth, type = "l", main = i,
       xlab = "Date", ylab = "Depth (m)", col = tCol("steelblue", 30))
  lines(t30min.2[[i]]$time.stamp, -t30min.2[[i]]$depth,
        col = tCol("firebrick", 40))
  legend("bottomleft", legend = c("Simulation 1", "Simulation 2"), cex = 0.9,
         col = tCol(c("steelblue", "firebrick"), 30), lty = 1, bty = "n")
}
```


dentex18

dentex43
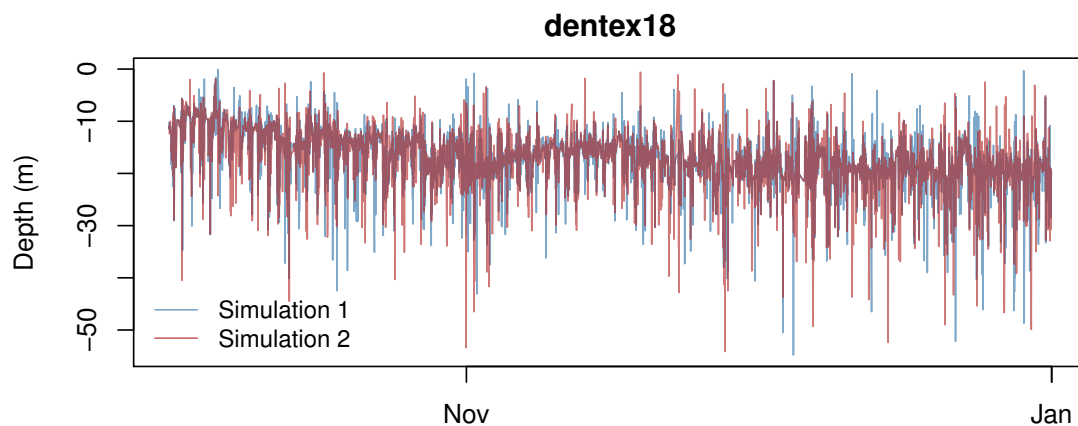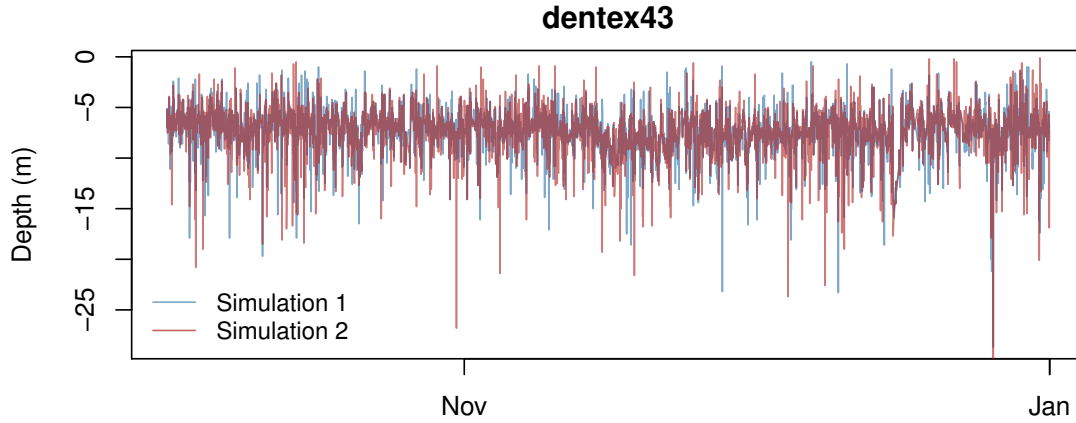
2.2. Sampling synthetic path realizations

The `syntPath` function generates synthetic path realizations by sampling the geographic coordinates for each detection in the acoustic telemetry data, taking into account the following points:

- **Distance from the receiver:** The function calculates the probability of being detected by each receiver in each of the cells of the topographic raster. This is done by applying the acoustic-range logistic model (generated in *Step 1.1*) to the distances between receivers and raster cells that were calculated in the `viewshed` object (generated in *Step 1.2*). The vertical distance is also considered by adding the difference between the depth of the receiver and the depth of each detection before applying the logistic model.

- **Distance from the previous sampled coordinates:** The sampling of a pair of coordinates is restricted to a certain distance from the previous position, in order to avoid unlikely movements between distant locations in short periods of time. This is especially useful if the fish is moving halfway between two receivers and detections rapidly alternate from one to the other, which could led to extremely rapid movements between the extremes of the two acoustic ranges. In this case, by restricting the maximum distance, the position of the individual will be forced to be closer to the space between receivers. The maximum distance is defined using the `max.vel` argument of the `syntPath` function, which limits the average maximum speed (in $m \cdot s^{-1}$) that the fish is assumed to reach along the synthetic trajectory. Taking into account the `max.vel` value and the elapsed time from the previous detection, a maximum distance is determined, beyond wich the probability of detecting the individual is set to zero. To consider this distance, the shortest distances from the previous location to the rest of the raster cells is calculated avoiding emerged landmasses.

- **Shortest path between sampled coordinates:** After sampling two pairs of coordinates, the function finds the shortest path connecting them taking into account the topography. By default, the only restriction to compute the shortest path is not to cross emerged landmasses (the `depth.cost.dist` argument is set as NULL). In this case, the `syntPath` function will internally generate the `TransitionLayer` object that is required to compute the shortest path, excluding the cells that correspond to emerged landmasses, using the `gdistance` package. However, we can also restrict the synthetic paths so that they do not cross cells at shallower depths than each pair of points to join. To do this, we must provide the `depth.cost.list` argument, which is a list of `TransitionLayer` objects. Each consecutive `TransitionLayer` of the list excludes a wider range of depths, from the surface to an increasing depth, at 1 m intervals between 0 and the maximum depth of the study area. If the `depth.cost.list` argument is provided, the `syntPath` function will pick the

`TransitionLayer` that corresponds to the minimum depth of the points to join to compute the shortest path. The `depth.cost.list` object can be created using the `leastCostMap` function as follows.

NOTE: Running this piece of code is quite slow, if you want to run through this vignette more quickly, one option is to directly load the `depth.cost.list` object from the files downloaded in the beginning of the vignette. Another option is to skip this step and run the `syntPath` function setting the `depth.cost.list` argument as `NULL`. By doing this, the resulting synthetic paths will avoid emerged land areas, but not the submerged ones.

```r
# Create the list of transition matrices
depths <- 0:70 # Vector of minimum depths to compute the matrices

depth_cost_list <- lapply(depths, function(x) {
  leastCostMap(topo = bathymetry, min.depth = x)
})

names(depth_cost_list) <- depths

# Load the 'depth.cost.list' object from the downloaded files (OPTIONAL)
# load("./data/depth_cost_list.rda")
```

Now, to see how the `syntPath` function works, we will apply it to the first 50 positions of our thinned tracks.

```r
synthetic.path <- lapply(t30min, function(data.frame) {

  # Remove receiver NAs and missing depths (if there are NAs).
  # We will run the example with only the first 50 positions
  df <- subset(data.frame, !is.na(depth))[1:50, ]

  path <- syntPath(track.data = df, topo = bathymetry, dist.rec = viewshed,
                   ac.range.mod = range.mod, depth.cost.list = depth_cost_list,
                   max.vel = 1, check = F)
})
```

```r
head(synthetic.path[[1]])
>   loc   tag.id rec.id             time.stamp      x       y depth     type
> 1   1 dentex18      9 2007-10-01 00:00:00 518920 4654510  10.7 original
> 2  NA dentex18   <NA> 2007-10-01 00:07:32 518920 4654570  10.8   interp
> 3   2 dentex18      9 2007-10-01 00:30:00 518840 4654730  11.0 original
> 4  NA dentex18   <NA> 2007-10-01 00:34:52 518840 4654650  11.0   interp
> 5  NA dentex18   <NA> 2007-10-01 00:49:51 518730 4654430  10.9   interp
> 6   3 dentex18      9 2007-10-01 01:00:00 518820 4654290  10.8 original
```

The resulting data.frame adds the sampled `x` and `y` coordinates to the acoustic telemetry data. Moreover, when the shortest path between two locations is not straigh, it interpolates the additional coordinates that define the path around land barriers. The `type` column indicates if the location corresponds to one of the receivers in the initial dataset (`original`) or if it has been interpolated (`interp`) to avoid barriers.

Now we can take a look at the path in two and three dimensions:

```r
# Plot in two dimensions
par(mar = rep(0.6, 4))
image(bathymetry, col = terrain.colors(50), asp = 1, ann = FALSE, axes = FALSE)
lines(synthetic.path[[1]][, c("x", "y")], col = "firebrick", lwd = 1.2)
lines(synthetic.path[[2]][, c("x", "y")], col = "dodgerblue", lwd = 1.2)
legend("topright", legend = names(synthetic.path), lty = 1, bg = "white",
       col = c("firebrick", "dodgerblue"), cex = 0.8)
```
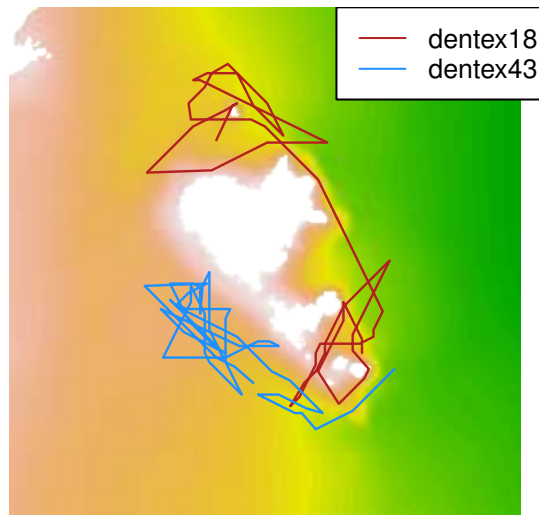


```r
# Plot in three dimensions (with the 'rgl' package)
library(rgl)

# Create the matrix to plot the bathymetry
x <- unique(coordinates(bathymetry)[, 1])
y <- unique(coordinates(bathymetry)[, 2])
z <- matrix(values(bathymetry), ncol = length(x))
z[is.na(z)] <- 0

# Prepare the color scale for bathymetry
zlim <- range(z, na.rm = TRUE)
zlen <- round(zlim[2] - zlim[1] + 1)
col <- c(terrain.colors(zlen)[z - zlim[1] + 1])

# Set the display matrix (manually obtained to get the desired view)
mat <- matrix(c(0.48, 0.88, -0.06, 0.00, -0.30, 0.23, 0.93, 0.00,
                0.83, -0.42, 0.37, 0.00, 0.00, 0.00, 0.00, 1.00),
              nrow = 4, byrow = TRUE)

# Open device and set other visualization options
open3d(scale = c(1, 1, 10), windowRect = c(0, 0, 600, 400))
rgl.viewpoint(userMatrix = mat, zoom = 0.65, fov = 30)
rgl.pop("lights")
light3d(specular = "black")

surface3d(x, y, z, col = col)
```
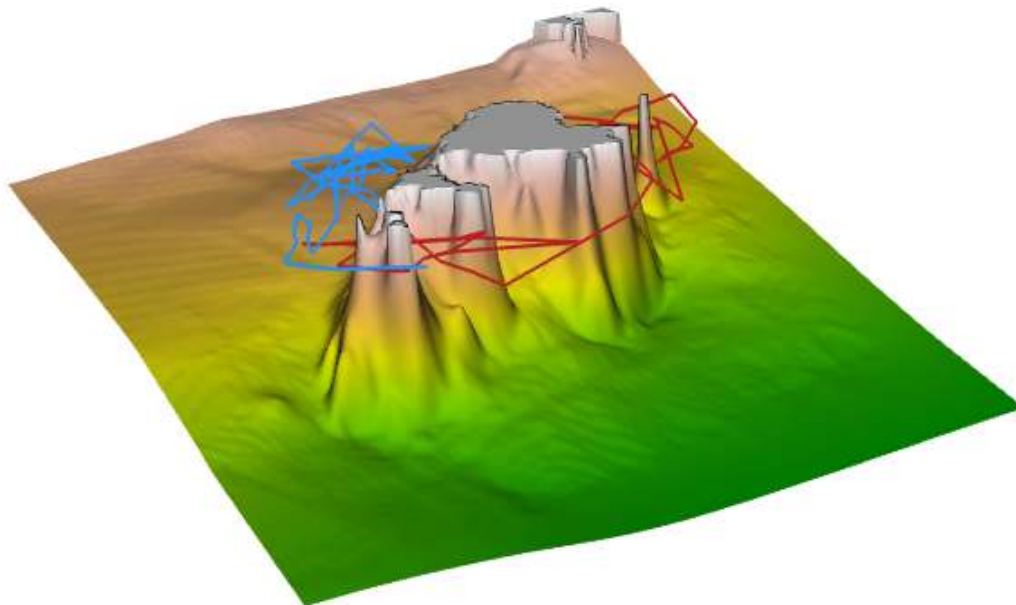
```
lines3d(x = synthetic.path[[1]]$x, y = synthetic.path[[1]]$y,
        z = -synthetic.path[[1]]$depth, lwd = 2, col = "firebrick")
lines3d(x = synthetic.path[[2]]$x, y = synthetic.path[[2]]$y,
        z = -synthetic.path[[2]]$depth, lwd = 2, col = "dodgerblue")
```


Arrived to this point, we are interested in generating a large number of stochastic trajectories from our original radiotracking data. We will do this by applying the `thinData` and the `synthPath` functions sequentially, as many times as synthetic trajectories we want to generate. This step can be easily parallelized to make it faster using the `llply` function of the `plyr` package.

NOTE: This part of the code can take a long time. You can reduce the number of iterations or the number of locations to process to make it faster, or you can load the `synt_path_x100` object from the downloaded files to diectly skip this step.

```
# Set the parallel back end (only for UNIX systems)
library(doMC)
doMC::registerDoMC(cores = 18) # Set number of cores to parallelize
parallel = TRUE # Set to false if the parallel back end is not used

synt_path_x100 <- lapply(tracking.list, function(df) {

  sim.ind <- plyr::llply(1:100, .parallel = parallel, function(x) {
    # Print individual and current simulation number
    cat(df$tag.id[1], "synthetic track no.", x, "\n")

    t30min <- thinData(df, time.int = "30min", depth.range = c(0, 69))
    t30min <- subset(t30min, !is.na(depth))
```

```
    synthetic.path <- syntPath(track.data = t30min, topo = bathymetry,
                               dist.rec = viewshed, ac.range.mod = range.mod,
                               depth.cost.list = depth_cost_list, max.vel = 1)
    return(synthetic.path)
  })
})
```

Step 3. Assemblage of utilization distribution volumes

In this step, we are going to merge all the synthetic paths to generate the 3D utilization distribution (3D-UD). We will do this with the `voxelize` function, which divides our study area in a 3D grid, and then calculates the average time that the synthetic trajectories spend in each one of the resulting voxels. Voxels are defined with the horizontal resolution (x and y) of a raster dataset, which will usually be the same topographic raster that has been previously used (in our case, it has a resolution of 10 x 10 m). The vertical (z) resolution is provided as a vector with the depth values that define the breaks into which the z axis will be partitioned (in this case, we will use a resolution of 1 m).

The `voxelize` function first divides each path into a large number of points, in order to ensure that there will be points falling in each of the voxels that have been crossed by the trajectory. Then, it calculates the proportion of points that fell within each voxel for each synthetic trajetory. Finally, it calculates the average proportions for all the synthetic paths. The resulting object is a `RasterStack` object, with one layer for each depth interval.

```
rast3d.list <- lapply(synt_path_x100, function(s) {
  voxelize(synt.list = s, raster = bathymetry, depth.int = 0:69, max.lag = 24)
})
```

The next image shows the result for six different depth ranges:
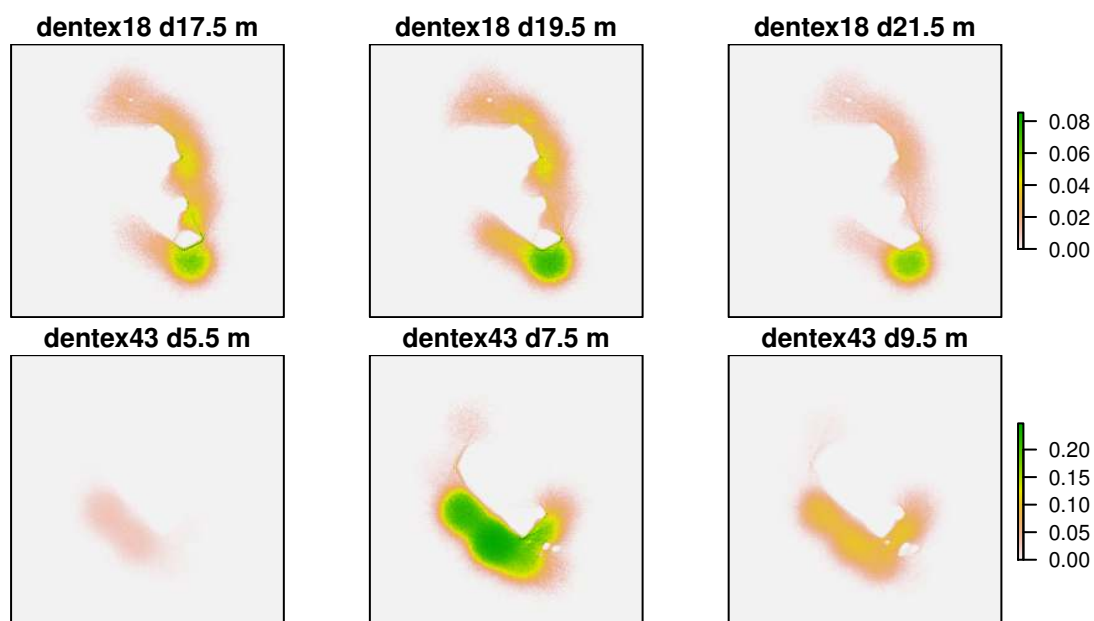
In this last step, we are going to generate smooth UD contours by applying a 3D kernel density estimation to the values calculated for each voxel. First, we have to convert the `RasterStack` object into a data frame with the x, y, and z coordinates and the average time spent by the path in each voxel. Then, we apply the `kde` function from the `ks` package to get the three-dimensional kernel density distribution. We let the `kde` function to set the kernel bandwidth values according to its optimization algorithms.

```
library(ks)

kde.list <- lapply(rast3d.list, function(r) {

  # Convert the RasterStack into a table
  table <- as.data.frame(r, xy = TRUE)
  table <- ldply(3:ncol(table), function(n) {
    return(cbind(table[, 1:2], depth = colnames(table)[n], val = table[, n]))
  })
  table <- table[table$val > 0, ]
  table$depth <- -as.numeric(substr(table$depth, start = 2, stop = 6))

  # Kernel density estimation
  kde.tmp <- kde(x = table[, 1:3], w = table$val, compute.cont = FALSE)
  return(kde.tmp)
})
```

We can compute and plot the 3D-UD contours using the `rgl` package. This example shows the 50% and the 90% UD contours (inner and outer volumes) for the two individuals in the example dataset (`dentex18` in purple-blue and `dentex43` in yellow-orange).

```
library(rgl)

# Generate a matrix to plot the bathymetry
x <- unique(coordinates(bathymetry)[, 1])
y <- unique(coordinates(bathymetry)[, 2])
z <- t(as.matrix(bathymetry))
z[is.na(z)] <- 0

# Set the color scale for depth
zlim <- range(z, na.rm = TRUE)
zlen <- round(zlim[2] - zlim[1] + 1)
col <- c(gray.colors(zlen)[z - zlim[1] + 1])

# Set the display matrix
mat <- matrix(c(0.48, 0.88, -0.06, 0.00, -0.30, 0.23, 0.93, 0.00, 0.83, -0.42,
                0.37, 0.00, 0.00, 0.00, 0.00, 1.00), nrow = 4, byrow = TRUE)

open3d(scale = c(1, 1, 10), windowRect = c(0, 0, 600, 400))
rgl.viewpoint(userMatrix = mat, zoom = 0.65, fov = 30)
rgl.pop("lights")
light3d(specular="black")
```
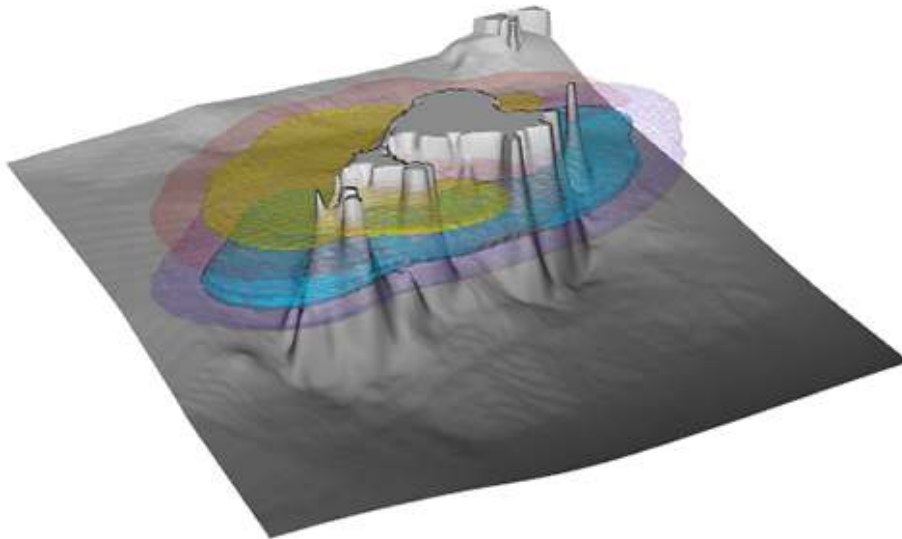
```
surface3d(x, y, z, col = col)
plot(kde.list[[1]], cont = c(25, 75), add = TRUE, axes = FALSE,
     col.fun = topo.colors, box = FALSE)
plot(kde.list[[2]], cont = c(25, 75), add = TRUE, axes = FALSE,
     col.fun = heat.colors, box = FALSE)
```


In the next part of code we apply the `volumeUD` function to compute the UD volumes from 3D-UD estimations. Before, we have to predict the values of the `kde` to our spatial grid using the `predictKde` function.

```
# Compute UD volumes
ud.vol.list <- llply(kde.list, function(k) {
    rast.tmp <- predictKde(kde = k, raster = bathymetry, depths = 0.5:69.5)
    rast.tmp <- volumeUD(rast.tmp)
    return(rast.tmp)
})
```

Plotting and exploring 3D contours with the `rgl` package is quite slow and visualizations do not look very good. I recomend to export the UD probability volumes to visualize them in other 3D visualization software such as `Paraview`.

```
# Tables to import into paraview
contour.table <- lapply(ud.vol.list, function(r) {

  # Convert the RasterStack into a table
  cont.tab <- as.data.frame(r, xy = TRUE)
  cont.tab <- ldply(3:ncol(cont.tab), function(n) {
    return(cbind(cont.tab[, 1:2], depth = colnames(cont.tab)[n],
                vol = cont.tab[, n]))
  })
```

```
  cont.tab$depth <- -as.numeric(substr(cont.tab$depth, start = 2, stop = 6))
  return(cont.tab)
})

# Save the tables to import in paraview
for (i in names(contour.table)) {
  write.csv(contour.table[[i]], row.names = FALSE,
            file = paste0("../", i, ".csv"))
}
```

The following picture represents a `Paraview` capture showing the same probability contours as in the previous figure. Red contours correspond to `dentex18`, and green contours to `dentex43`. 50% and the 90% UD contours are represented by the inner and outer volumes, respectively.
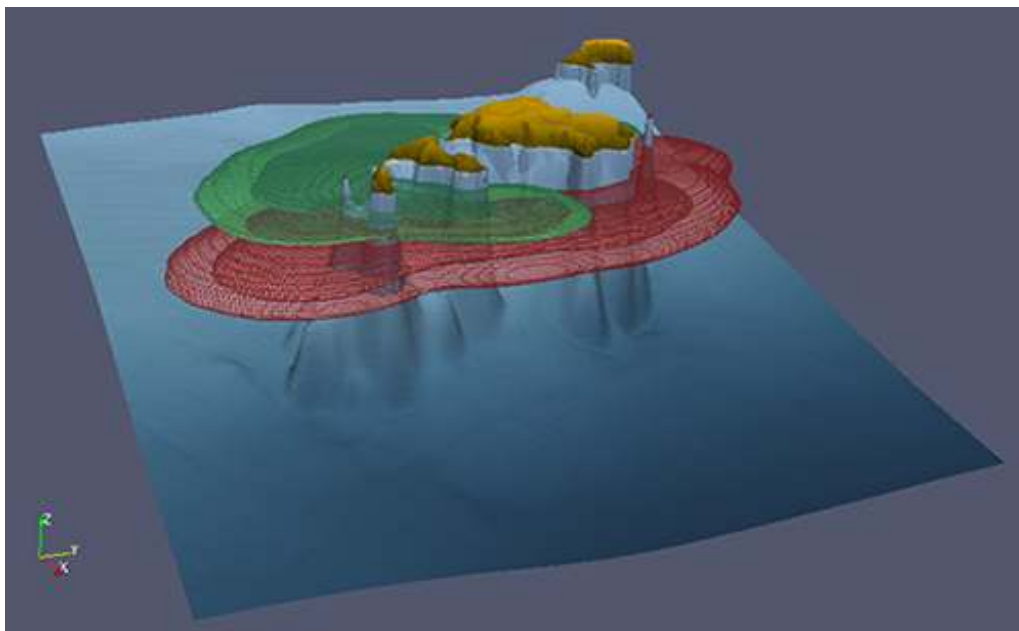
Overlap between 3D-UD volumes

In the next step, we show how to calculate the overlap index between two 3D-UD volumes with the `overlap3d` function. This function takes a list of 3D-UD volumes and calculates, for each pair of elements, the proportion of voxels that they have in common within a certain contour level. When the `symmetric` argument is `TRUE`, the overlap is calculated referred to the joint volume of the UDs to compare, so the upper and lower triangles of the resulting matrix are equal. If the `symmetric` argument is set to `FALSE`, the overlap is calculated referred to the volume of each single UD, so the upper and lower triangles of the resulting matrix might no be equal.

```
overlap.50 <- overlap3d(ud.vol.list, level = 0.5, symmetric = FALSE)
overlap.95 <- overlap3d(ud.vol.list, level = 0.95, symmetric = FALSE)
```

In this case, UD volumes are not overlapped at the 50% contour level:

```
overlap.50
>         dentex18 dentex43
> dentex18        1        0
> dentex43        0        1
```

But they slightly overlap at the 95% contour level:

```
overlap.95
>         dentex18 dentex43
> dentex18    1.000    0.029
> dentex43    0.082    1.000
```

This values mean that, at the 95% contour level, 8.2% of the UD volume of `dentex43` is overlapped with the volume of `dentex18`, but that only the 2.9% of the UD of this second individual is overlaped with the UD of the first one.