How to easily split transects into segments using the dshm R-package

Filippo Franchini

2019-01-18

Summary

In this tutorial you will learn how to use the `dshm` R-package to:

- Split transects into segments and apply a buffer to them using the `dshm_split_transects` function.
- Check if segments have overlapping areas using the `dshm_check_segments` function. Correct the segments using the function `dshm_correct_segments` and re-check to prove all the overlapping areas have been deleted.
- Finalize the segments for spatial analyisis using the function `dshm_finalize_segments`. This process includes:
  - Taking out overlapping areas between buffered segments and land.
  - Calculating covariate statistics for each segment.

Getting started

In order to use `dshm` you have to install the `countreg` R-package that is (unfortunately) not available on CRAN. You can install it using the following code:

```
install.packages("countreg", repos="http://R-Forge.R-project.org")
```

Alternatively, `countreg` can be found and downloaded as .tar file in the `dshm /local_repository` on GitHub:

`https://github.com/FilippoFranchini/dshm/blob/master/local_repository/countreg_0.2-0.tar`

You can then install `countreg` in R studio: go to Packages → Install → In the `Install from` dropdown menu select `Package Archive File` → Search for the downloaded `countreg_0.2-0.tar` file. After sucessfully installing `countreg` you can install the `dshm` package by running the following code:

```
devtools::install_github("FilippoFranchini/dshm")
```

The `dshm` installer will check if you already have the required packages to make `dshm` work properly. Required packages will be automatically installed if you do not have them in your repository. After installing the package you can load it by typing:

```
library(dshm)
```

Now `dshm` is loaded and you have access to all functions and datasets. You can explore all datasets in `dshm` by typing `data(package="dshm")`.

Splitting transects into segments

Basics

The object `transects` is a `SptialLinesDataFrame` containign 15 lines (i.e. transects) with data associated with each line that you can access by typing:

```
transects@data
```

You can show the `transects` spatially by typing:

```
raster::plot(transects)
```

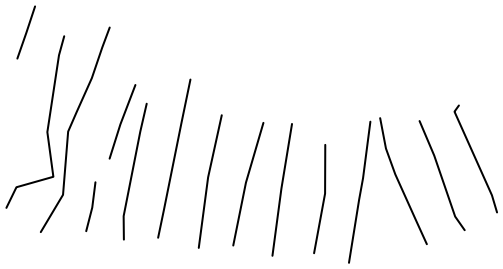

Figure 1: Transect lines.

As you can see these are spatial lines that represent the transect lines covered by a ship during a survey. Now we would like to split those lines into segments of a certain length and we would also like to convert those segments into polygons by adding a buffer. You can easily do that using the funtion `dshm_split_transects` as follows:

```
segments <- dshm_split_transects(transect.data = transects,
            lwr = 5000,
            search.time = 30,
            w = 1500)
```

The R console will display a percentage bar and the time needed to complete the operation. You can show the obtained segments spatially by typing:

```
raster::plot(segments)
```

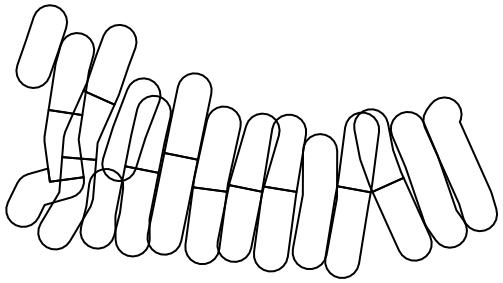

Figure 2: Segments.

Key Arguments

Let's explore each of the arguments of the funtction `dshm_split_transects`:

- `transect.data` is the `SpatialLinesDataFrame` that you have, i.e. the object containing the transects you would like to split.

- `inter.dist` is the distance between segments in meters once the transects will be split. Usually such distance is short, but if you need to, you can set it larger. Default is 10 cm.
- `lwr` is the lower limit for the length (in meters) of the segments. Segments (originating from split transects) will never be shorter than the `lwr` value.
- `search.time` is the time in seconds the algorithm is searching for the solution. This is equivalent to algorithm precison. Note that this does not correspond to the time required to find the solution! You will see this later. Default is 15 seconds.
- `w` is the strip width, i.e. the buffer in meters applied around each segment to convert them into polygons.

What you get

Segments have data associated with them. You can show segment data by typing:

```
head(segments@data, 6)
```

This will print the first 6 rows of the segment data:

Transect.Label	Sample.Label	length	area
1049102	1	4.077627	19.18522
1049103	1	6.384895	26.10751
1049104	1	5.087868	22.07933
1049104	2	6.213111	18.71048
1049105	1	7.286680	21.17518
1049105	2	5.910528	25.29813

In the table you can see four main columns. The `Transect.Label` is the ID of the original transect that has been split, while the `Sample.Label` is the ID of the segments originating from each transect. For each segment we also have `length` (km) and `area` (km^2). You can see that the first two transects (i.e. 1049102 and 1049103) all have 1 segment, meaning that they were not split. This is due to the fact that we set a `lwr` value of 5 km and those two transects are 4.07 and 6.38 km long, i.e. impossible split them into two segments of 5 km each. The third segment (i.e. 1049104) was split into two segments of 5.08 and 6.21 km, respectively. In total the object `segments` contains 26 segments that are >5km in length (except for those transects that were not split!). Note that segment count may vary since the splitting algorithm is based on random process (for more information see documentation for *spsample*).

Parallelization: Speeding-up Calculations

You are probably confused about the argument `search.time`. Let's say we would like to split our transects into segments with minimum length of 2 km with a search time of 15 s. We can run the following code (this will take a few minutes!):

```
segments_2km <- dshm_split_transects(transect.data = transects,
                lwr = 2000,
                w = 1500)
```

The time required to find the solution in my case is 2.3 minutes. This is because we are asking the function to split the transects into small segments, and to be very precise in doing it. Decreasing the search time from 15 s to 1 s lowers the time to reach the solution to 20 seconds. However, we lost precision since with 15 s we got 64 segments while with 1 s only 58.

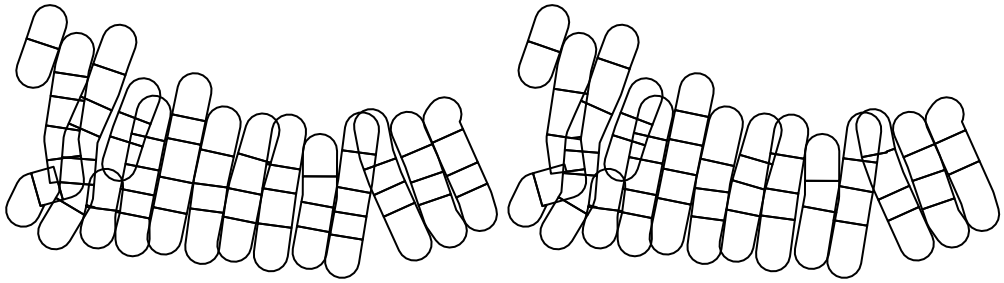

Figure 3: (*left*) Search time of 15 seconds. (*right*) Search time of 1 second.

Idieally, we would like to have the highest precision using the lowest amount of time. This is very useful in situations where we have a lot of transects or very long transects to split into small segments. With `dshm` this is possible by parallelizing the function `dshm_split_transects`. The parallelization process divides the splitting task into samller jobs that are assigned to differert cores on your computer. The code is the same as the previous one, you just have to specify the arguments `prallel = TRUE` and `ncores` (number of cores you would like to use). For the `ncores` value you have to enter the right amount according to the technical specification of your computer. You can know how many cores has your machine by typing `parallel::detectCores()`. I recommend to leave at least one core free.

You can parallelize the splitting task using the following code:

```
segments_2km_par <- dshm_split_transects(transect.data = transects,
                    lwr = 2000,
                    w = 1500,
                    parallel = TRUE,
                    ncores = 7)
```

Without parallelization it took 2.3 minutes while with parallelization on 7 (virtual) cores only 48 seconds. Please note that the percentage bar is not displayed for parallel execution. I suggest monitoring the process with task manager: CPUs that are working will be busy while those that have finsihed the job will be free. You will usually notice a cascade-like pattern with some cores finishing quickly and others later. Another sign that your computer is doing calculations on many CPUs is just the noise of the fan(s). If the parallelization is working then the temperature will increase and thus also the rpm of the fan(s). Do not worry your computer will not die!

Capping

Capping is the option to add a cap to each of the two segments at the ends of a transect. Capping is controlled by the argument `cap` and by default it is enabled (i.e. `TRUE`). You can disable it by setting `cap = FALSE`.

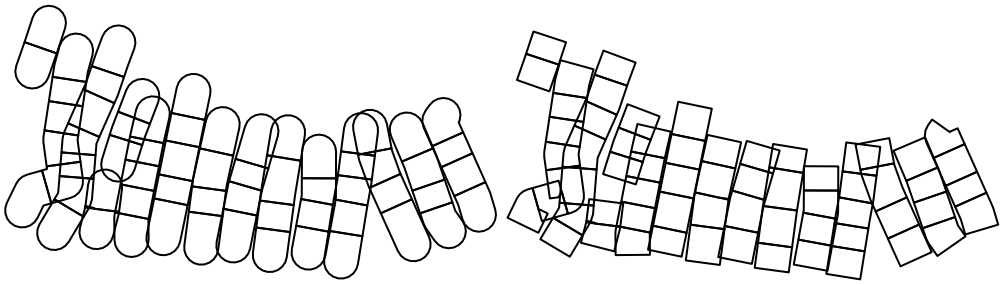

Figure 4: (*left*) With cap. (*right*) Without cap.

Check and Correct Segments

You probably noticed that when you are splitting transects the buffered segments have overlapping areas. Under certain circumstances you may want to avoid such areas. This is possible with the functions `dshm_check_segments` and `dshm_correct_segments`. You need to use `dshm_check_segments` to create an object containing a map of all intersections bwtween buffered segments:

```
intersections<-dshm_check_segments(data = segments_2km_par)
```

The code you just used displays the message `27 overlapping features found. Go to 'dshm_correct_segments'`, and creates a map of all intersections (available typing `intersections$inter.IDs`) as well as a list of all overlapping areas (i.e. intersection polygons, available typing `intersections$inter.Polys`). You can plot all overlapping polygons with the following code:

```
raster::plot(segments_2km_par)
raster::plot(intersections$inter.Polys, col = "red", add = TRUE)
```

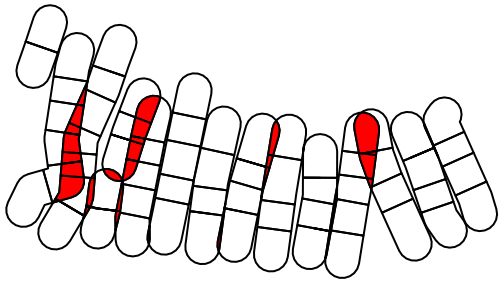

Figure 5: Overlapping features (red areas) between buffered segments.

This shows in red all overlapping areas between segments. Note that some areas are so small that are impossible to notice on the selected scale. You can now correct the segments:

```
cor_seg <- dshm_correct_segments(data = segments_2km_par,
           intersections = intersections$inter.IDs)
```

You have corrected the segments. To be 100% sure that the function `dshm_correct_segments` did a good job you can always re-check the corrected segments by typing `dshm_check_segments(data = cor_seg)`. If

there are no overlapping areas left you should get the message `No overlapping features. Segments are OK`. We can now plot the corrected segments:

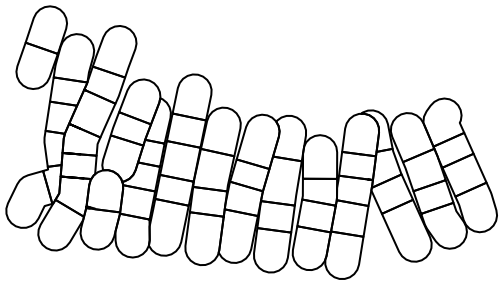

Figure 6: Corrected segments.

Area values are updated according to the correction.

Covariate Statistics & Land Correction

Now we have our segements with no overlapping areas and with information about:

- Transect origin
- Segment ID
- Segment length
- Segment area

However, to model animal distribution spatially we would also like to:

- Correct segments for land. This is particularly useful for marine animals that live in coastal waters. In fact, there is high chance that the segment buffer might include land or small islands. In order to be precise we have to take out such areas that are never 'used' by marine species.
- Associate covariate values with each segment. This is compulsory if we want to model (and predict) animal distribution spatially.

These two steps can be easily done with the function `dshm_finalize_segments`. This function corrects the buffered segments for land and calculates covariate statistics for each segment given covariate raster images. You can access three raster images for `depth`, distance to river (`DR`) and distance to coast (`DC`) at 50 m resolution. You can visualize such rasters together with land and corrected segmnets by typing:

```
dev.new()
par(mfrow=c(2,2))
raster::plot(depth_crop,main="Depth (m)",axes=FALSE)
raster::plot(land_crop,col="grey",add=TRUE)
raster::plot(cor_seg,col=rgb(0,0,0,0),lwd=3,add=TRUE)
raster::plot(DC_crop/1000,main="Distance to coast (km)",axes=FALSE)
raster::plot(land_crop,col="grey",add=TRUE)
raster::plot(cor_seg,col=rgb(0,0,0,0),lwd=3,add=TRUE)
raster::plot(DR_crop/1000,main="Distance to river (km)",axes=FALSE)
raster::plot(land_crop,col="grey",add=TRUE)
raster::plot(cor_seg,col=rgb(0,0,0,0),lwd=3,add=TRUE)
```

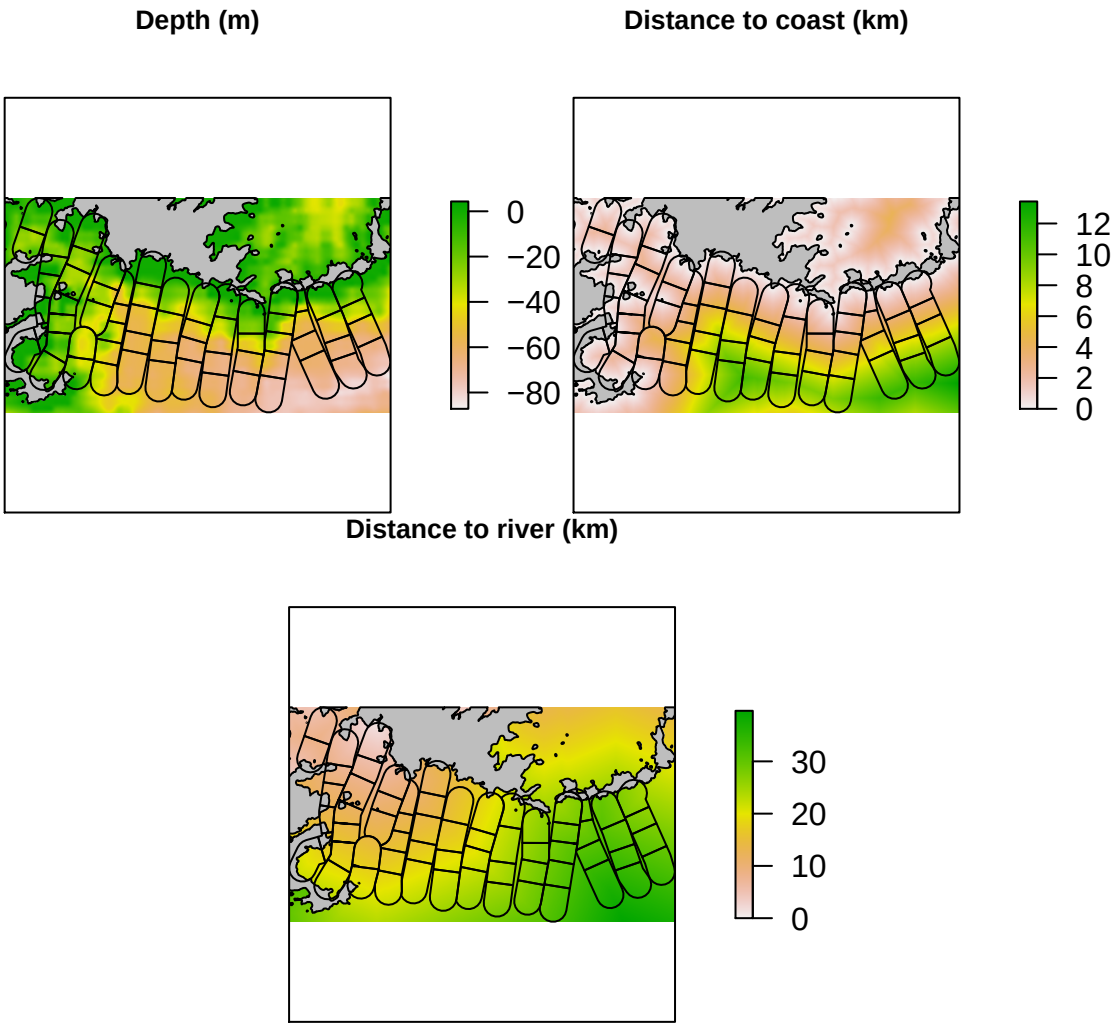

Figure 7: Corrected segments superimposed to covariate rasters for depth, distance to coast and distance to river.

As you notice, the coast is highly convoluted and many buffered segments have overlapping areas with land. You can finalize your segments by typing:

```
cor_seg_final <- dshm_finalize_segments(segment.data = cor_seg,
                 land.data = land_crop,
                 covariates = list(DR=DR_crop,DC=DC_crop,depth=depth_crop),
                 fun=mean,
                 parallel=TRUE,
                 ncores=7)
```

As the function `dshm_split_transects`, also the function `dshm_finalize_segments` can be parallelized. The code above took 11 seconds while the non-parallelized version 40 seconds. In order to make the function work you have to specify the `segment.data` (i.e. the `SpatialPolygonsDataFrame` from the previous step), `land.data` (i.e. land as `SpatialPolygonsDataFrame`, this can be ignored if there is no land), the `covariates` as a list of `RasterLayer` files, and the `fun` (i.e. the function to be used to calculate the covariate statistics within each segment). The final segments should look like this:

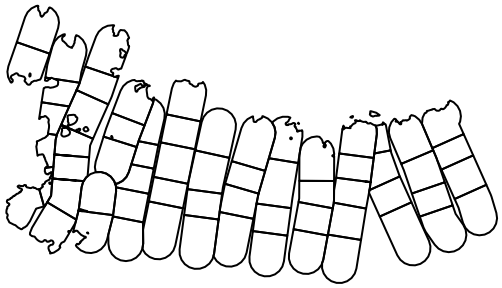

Figure 8: Finalized segments.

With associated covariate values:

Transect.Label	Sample.Label	length	area	DR	DC	depth
1049102	1	2.024548	9.537729	16109.034	3330.079	-35.97474
1049102	2	2.053058	8.741833	19426.773	1081.578	-21.58158
1049103	2	2.071532	6.206064	10596.591	1451.752	-34.09187
1049103	1	2.265854	8.668225	8564.912	1177.653	-17.04233
1049103	3	2.047469	9.289726	12939.528	2163.381	-34.83399
1049104	2	2.071706	3.835170	13415.268	3427.887	-48.10600

Where DC and DR are mean values for distances (in meters) to coast and river, respectively. If we inspect carefully the data we notice that there may be some segments with positive `depth`. Since this is impossible, we have to take out those segments with the following code:

```
cor_seg_final<-cor_seg_final[cor_seg_final$depth<0,]
```

One segment was deleted in the process:

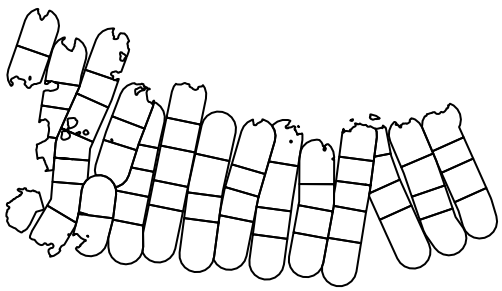

Figure 9: Finalized segments contrained to depth<0.

If you have information about strata, you can add Region.Label (i.e. ID for the stratum where the segment is located) as follows:

```
seg_region <- sp::over(cor_seg_final,strata)$ID
cor_seg_final@data <- data.frame(Region.Label = seg_region, cor_seg_final@data)
```

This will add a column for stratum ID to the segment data:

Region.Label	Transect.Label	Sample.Label	length	area	DR	DC	depth
8	1049102	1	2.024548	9.537729	16109.034	3330.079	-35.97474
8	1049102	2	2.053058	8.741833	19426.773	1081.578	-21.58158
8	1049103	2	2.071532	6.206064	10596.591	1451.752	-34.09187
8	1049103	1	2.265854	8.668225	8564.912	1177.653	-17.04233
8	1049103	3	2.047469	9.289726	12939.528	2163.381	-34.83399
7	1049104	2	2.071706	3.835170	13415.268	3427.887	-48.10600

Save Segments as Shapefile

You can easily save the segments as a shapefile (i.e. .shp) so that you can import them in other GIS software such as ArcGIS or QGIS. As a first step you have to abbreviate all column descripions:

```
names(cor_seg_final)<-c("RL","TL","SL","L","A","DR","DC","D")
```

Then you can run the following code:

```
rgdal::writeOGR(obj=cor_seg_final,dsn="/Users/User/Desktop/data",
layer="cor_seg_final",driver="ESRI Shapefile",overwrite_layer=TRUE)
```

You have to specify your own `dsn`, i.e. the directory where the file will be saved. You can then upload the saved file into a GIS software or to reload it into R by typing:

```
rgdal::readOGR("cor_seg_final.shp")
```

Note that you have to specify the directory where the file was saved.