Handling spatial vector data in R

Sytze de Bruin, Jan Verbesselt

November 1, 2013

1 Today's learning objectives

In today's lecture, we will explore the basics of handling spatial vector data in R. There are several R packages for this purpose but we will focus on using `sp`, `rgdal`, `rgeos` and some related packages. At the end of the lecture, you should be able to

- create point, line and polygon objects from scratch;
- explore the structure of `sp` classes for spatial vector data;
- plot spatial vector data;
- transform between datums and map projections;
- apply basic operations on vector data, such as buffering, intersection and area calculation;
- use a Date-Time class;
- write spatial vector data to a kml file;
- convert spatial data read from a plain text file into a spatial class for vector data.

2 Some packages for working with spatial vector data in R

The packages `sp` and `rgdal` are widely used throughout this course. Both packages not only provide functionallity for raster data but also for vector data. For example, rgdal includes bindings to parts of the OGR Simple Feature Library which provides access to a variety of vector file formats such as ESRI Shapefiles and kml. Similarly, rgeos is an interface to the powerful Geometry Engine Open Source (GEOS) library for all kind of operations on geometries (buffering, overlaying, area calculations, etc.). Thus, functionality that you commonly find in expensive GIS software is also available within R, using free but very powerful software libaries. The possiblities are huge; in this course we can only scratch the surface with some essentials which hopefully invite you to experiment further and use them in your research. Details can be found in the book Applied Spatial Data Analysis with R (Bivand et al., 2013) and several vignettes authored by Roger Bivand, Edzer Pebesma and Virgilio Gomez-Rubio. Owing to time constraints, this lecture cannot cover the related package `spacetime` with classes and methods for spatio-temporal data.

3 Creating and manipulating geometries

The package sp provides classes for spatial-only geometries, such as SpatialPoints (for points), and combinations of geometries and attribute data, such as a SpatialPointsDataFrame. The following data classes are available for spatial vector data (Pebesma & Bivand, 2005):

data type	class	attributes
points	SpatialPoints	No
points	SpatialPointsDataFrame	data.frame
line	Line	No
lines	Lines	No
lines	SpatialLines	No
lines	SpatialLinesDataFrame	data.frame
rings	Polygon	No
rings	Polygons	No
rings	SpatialPolygons	No
rings	SpatialPolygonsDataFrame	data.frame

We will go through a few examples of creating geometries from scratch to familiarize yourself with these classes.
First, start Google Earth on your computer and make a note of the longitude and latitude of two points in Wageningen that are relevant to you. Use a decimal degree notation with at least 4 digits after the decimal point. To change the settings in Google Earth click `Tools | Options` and change the Show lat/Long setting on the 3D View Tab.

Points

The example below shows how you can create spatial point objects from these coordinates. Type `?<function name>` (e.g. `?cbind`) for finding help on the functions used.

```
> # load sp package
> library(sp)
> library(rgdal)
> # coordinates of two points identiefied in Google Earth, for example
> pnt1_xy <- cbind(5.6660, 51.9872)   # enter your own coordinates
> pnt2_xy <- cbind(5.6643, 51.9668)   # enter your own coordinates
> # combine coordinates in single matrix
> coords <- rbind(pnt1_xy, pnt2_xy)
> # make spatial points object
> prj_string_WGS <- CRS("+proj=longlat +datum=WGS84")
> mypoints <- SpatialPoints(coords, proj4string=prj_string_WGS)

> # inspect object
> class(mypoints)
> str(mypoints)

> # create and display some attribute data and store in a data frame
> mydata <- data.frame(cbind(id = c(1,2),
+                  Name = c("my description 1",
+                           "my description 2")))
> # make spatial points data frame
> mypointsdf <- SpatialPointsDataFrame(
+   coords, data = mydata,
+   proj4string=prj_string_WGS)

> class(mypointsdf) # inspect and plot object
> names(mypointsdf)
> str(mypointsdf)
> spplot(mypointsdf, zcol="Name", col.regions = c("red", "blue"),
+        xlim = bbox(mypointsdf)[1, ]+c(-0.01,0.01),
+        ylim = bbox(mypointsdf)[2, ]+c(-0.01,0.01),
+        scales= list(draw = TRUE))
```

Notice the difference between the objects mypoints and mypointsdf.

Lines

Now let us connect the two points by a straight line. First find information on the classes for lines that are available in `sp`. The goal is to create SpatialLinesDataFrame but we have to go through some other classes.

```
> # consult help on SpatialLines class
> simple_line <- Line(coords)
> lines_obj <- Lines(list(simple_line), "1")
> spatlines <- SpatialLines(list(lines_obj), proj4string=prj_string_WGS)
> line_data <- data.frame(Name = "straight line", row.names="1")
> mylinesdf <- SpatialLinesDataFrame(spatlines, line_data)

> class(mylinesdf)
> str(mylinesdf)
> spplot(mylinesdf, col.regions = "blue",
+        xlim = bbox(mypointsdf)[1, ]+c(-0.01,0.01),
+        ylim = bbox(mypointsdf)[2, ]+c(-0.01,0.01),
+        scales= list(draw = TRUE))
```

Try to understand the above code and its results by studying help.

Writing and reading spatial vector data using OGR

What now follows is a brief intermezzo before we continu with the classess for polygons. Let us first export the objects created thusfar to kml files that can be displayed in Google Earth. We will use OGR functionality for that purpose, which is available through the package `rgdal`.

```
> library(rgdal)
> # write to kml ; below we assume a subdirectory data within the current
> # working directory.
```

```
>
> writeOGR(mypointsdf, file.path("data","mypointsGE.kml"),
+          "mypointsGE", driver="KML", overwrite_layer=TRUE)
> writeOGR(mylinesdf, file.path("data","mylinesGE.kml"),
+          "mylinesGE", driver="KML", overwrite_layer=TRUE)
```

Check (in Google Earth) whether the attribute data were written to the kml output.

The function readOGR allows reading OGR compatible data into a suitable Spatial vector object. Similar to writeOGR, the function requires entries for the arguments dsn (data source name) and layer (layer name). The interpretation of these enties vary by driver. Please study details in the help file.
Digitize a path (e.g. a bicycle route) between the two points of interest you selected earlier in Google Earth. This can be achieved using the Add Path functionality of Google Earth. Save the path in the data folder within the working directory under the name route.kml. We will read this file into a spatial lines object and add it to the already existing SpatialLinesDataFrame object.

```
> myroute <- readOGR(file.path("data","route.kml"), "route.kml")

> # put both in single data frame
> myroute@proj4string <- prj_string_WGS
> names(myroute)

[1] "Name"        "Description"

> myroute$Description <- NULL # delete Description
> mylinesdf <- rbind(mylinesdf, myroute)
```

Try to understand the above code and results. Feel free to display the data and export to Google Earth.

Transformation of coordinate system

Transformations between coordinate systems are crucial to many GIS applications. The Keyhole Markup Language (kml) used by Google Earth uses latitude and longitude in a polar WGS84 coordinate system. However, in some of the examples below we will use metric distances. One way to deal with this is by transforming the data to a planar coordinate system. In R this can be achieved via bindings to the PROJ.4 - Cartographic Projections Library (http://trac.osgeo.org/proj/), which are available in rgdal. We will transform our spatial data to the Dutch grid (Rijksdriehoekstelsel), often referred to as RD. Please note that:

- some widely spread definitions of the Dutch grid (EPSG: 28992) are incomplete;
- the transformation used below is approximate since it does not account for time dependent differences between WGGS84 and ETRS89 and it does not apply a correction grid for modelling errors in the original measurements of RD. Details can be found at http://nl.wikipedia.org/wiki/Rijksdriehoekscoordinaten.

```
> # define CRS object for RD projection
> prj_string_RD <- CRS("+proj=sterea +lat_0=52.15616055555555 +lon_0=5.38763888888889
+ +k=0.9999079 +x_0=155000 +y_0=463000 +ellps=bessel +towgs84=565.2369,50.0087,465.658,
+ -0.406857330322398,0.350732676542563,-1.8703473836068,4.0812 +units=m +no_defs")
> # perform the coordinate transformation from WGS84 to RD
> mylinesRD <- spTransform(mylinesdf, prj_string_RD)
```

Now that the geometries are projected to a planar coordinate system the length can be computed using a function from the package rgeos.

```
> # use rgeos for computing the length of lines
> library(rgeos)
> mylinesdf$length <- gLength(mylinesRD, byid=T)
```

Feel free to export the updated lines to Google Earth or to inspect the contents of the `data` slot of the object `mylinesdf` (i.e. type `mylinesdf@data`).

Polygons

We now continue with `sp` classes for polygon objects. The idea is to illustrate the classes; the data are meaningless. Let us create overlapping circles around the points you defined earlier.

```
> # make circles around points, with radius equal to distance between points
> mypointsRD <- spTransform(mypointsdf, prj_string_RD)
> pnt1_rd <- coordinates(mypointsRD)[1,]
> pnt2_rd <- coordinates(mypointsRD)[2,]
> # define a series of angles
> ang <- pi*0:200/100
> circle1x <- pnt1_rd[1] + cos(ang) * mylinesdf$length[1]
> circle1y <- pnt1_rd[2] + sin(ang) * mylinesdf$length[1]
> circle2x <- pnt2_rd[1] + cos(ang) * mylinesdf$length[1]
> circle2y <- pnt2_rd[2] + sin(ang) * mylinesdf$length[1]
> # Iterate through some steps to create SpatialPolygonsDataFrame object
> circle1 <- Polygons(list(Polygon(cbind(circle1x, circle1y))),"1")
> circle2 <- Polygons(list(Polygon(cbind(circle2x, circle2y))),"2")
> spcircles <- SpatialPolygons(list(circle1, circle2), proj4string=prj_string_RD)
> circledat <- data.frame(mypointsRD@data, row.names=c("1", "2"))
> circlesdf <- SpatialPolygonsDataFrame(spcircles, circledat)
```

Similar results can be obtained using the function gBuffer of the package rgeos, as demonstrated below. Notice the use of two overlay functions from the package `rgeos`.

```
> buffpoint <- gBuffer(mypointsRD[1,], width=mylinesdf$length[1], quadsegs=25)
> mydiff <- gDifference(circlesdf[1,], buffpoint)
> gArea(mydiff)
```

```
[1] 8005.344
```

```
> myintersection <- gIntersection(circlesdf[1,], buffpoint)
> gArea(myintersection)
```

```
[1] 16215548
```

```
> print(paste("The difference in area =", round(100 * gArea(mydiff) /
+                                              gArea(myintersection),3), "%"))
```

```
[1] "The difference in area = 0.049 %"
```

Here is an example of a plot of the results which employs a few more advanced options of `spplot`.

```
> spplot(circlesdf, zcol="Name", col.regions=c("gray60", "gray40"),
+        sp.layout=list(list("sp.points", mypointsRD, col="red", pch=19, cex=1.5),
+                       list("sp.lines", mylinesRD, lwd=1.5)))
```

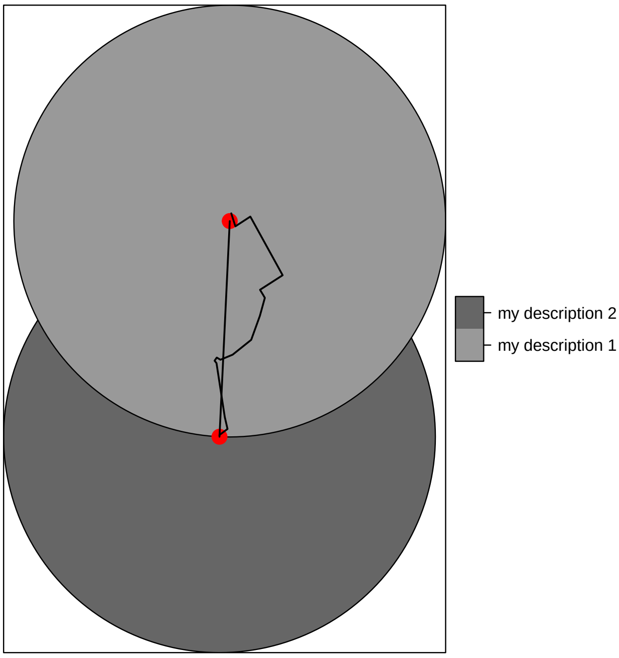

Figure 1: Basic points, lines and polygons plotted with `spplot`.

4 Real world example

We will now study a data set acquired by a potato harvester equiped with a yield monitoring system (YieldMasterPro of Probotiq / Van den Borne). The data come in a comma delimited text file and correspond to GPS positions (WGS84) and yield data at point locations along with other attributes. The typical working width is 3 meters, i.e. the potatos are harvested over such width. The harvester has a load capacity of approximately 25 metric tons after which the bunker is emptied.

Our goal is to map where a particular load was harvested (tracking and tracing) and also to

map the average yield (ton/hectare) over these harvest blocks . A complicating factor is that trajectories consist of multiple swaths that can be spatially separated whilst the path between the swaths does not contribute to the load. Furthermore, some parts of the field (typically headlands) are harvested first and later used to facilitate manoeuvring on the field. Of course these areas should not be counted multiple times for computing yield per hectare.
We will read the data and convert these in a spatial data frame and next use a custom built function (`CreateHarvestTracks`) for creating tracks. One of *your* tasks is to study the function `CreateHarvestTracks` and explain it. Next, you are asked to write a brief script in which:

- sequences of tracks are converted into harvest blocks;
- average productivity per block is computed;
- the blocks and their attribute data are exported to Google Earth.

Read and preprocess the yield data

In the previous section you created spatial point data from scratch. Of course `sp` has functionality for converting tabular data with coordinates into spatial objects. Below, the `sp` function `coordinates` is used for that purpose. Notice also the arguments passed to `spplot` for plotting the data.

```
> # download and read data
> download.file("http://rasta.r-forge.r-project.org/kroonven.csv", "kroonven.csv")
> borne_data = read.table("kroonven.csv", sep = ",", header = TRUE)

> names(borne_data)

 [1] "year"             "month"            "day"              "hr"
 [5] "min"              "sec"              "lon.degr."        "lat.degr."
 [9] "alt.m."           "qual"             "sats"             "x.m."
[13] "y.m."             "speed.km.h."      "load.kg."         "tare.kg."
[17] "conv.factor"      "beltspd.m.s."     "workwidth.m."     "yield.ton.ha."
[21] "totalyield.ton."  "totalarea.ha."    "worktime.s."      "loadnr"
[25] "loadweight.ton."  "loadbelt.m."      "enginespd.rpm."   "demandtorque.."
[29] "actualtorque.."   "fuelrate.l.h."    "totfuel.l."       "tractrear.bar."

> # make spatial
> coordinates(borne_data) <- c("lon.degr.","lat.degr.")
> borne_data@proj4string <- prj_string_WGS
> # transform to planar coordinate system
> all_rd <- spTransform(borne_data, prj_string_RD)
> dimnames(all_rd@coords)[[2]] <- c("x", "y")
> # plot the point data
> spplot(all_rd, zcol="yield.ton.ha.", colorkey=T, zlim=c(0,100),
+         col.regions=c(bpy.colors(25), rep("yellow", 75)), pch=19,
+         cex=0.25, main="Recorded yields ton/ha")
```

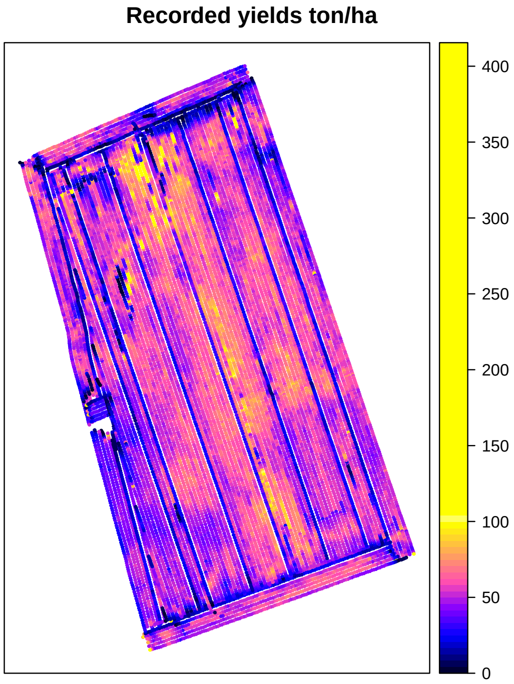

Figure 2: Yields per point. Note that the legend bar accomodates unrealistically high yield measurements.

Date and time classes

The date and time information in the yield data file is spread over 6 attributes. The base package of R includes two basic classes of date/times: POSIXct and POSIXlt. The former represents the (signed) number of seconds since the beginning of 1970 (in the UTC timezone) as a numeric vector. On the other hand, class POSIXlt is a named list of vectors representing seconds, minutes, hours, day of the month, month and year. The function as.POSIXct can be used for converting an object such as a character string to an object of class POSIXct.

```
> # add datetime attribute using POSIX class
> all_rd$datetime <- as.POSIXct(paste(paste(all_rd$year, all_rd$month, all_rd$day,
+                               sep="-"), paste(all_rd$hr, all_rd$min, all_rd$sec,
+                               sep=":")), tz="Europe/Andorra")
```

Try to understand the above code and its results by studying help.

Computing tracks and swaths

The trajectory of the harvester can be represented by a series of continuous tracks, which correspond to different loads (identified by load number and date) and working widths, while points within a track are separated by a maximum distance. The function CreateHarvestTracks (rasta) creates such tracks from a *data frame* with *temporally ordered* input records. Note that the previous sentence refers to a data frame, NOT a spatial data frame.

```
> # coerce spatial data frame to data frame
> all_rd <- as.data.frame(all_rd)
> # make sure points are temporally ordered
> all_rd <- all_rd[order(all_rd$datetime),]

> # load rasta library
> library(rasta)

> # call function from rasta to create lines; will take a minute or so
> sp_lines_df <- CreateHarvestTracks(all_rd, prj_string_RD)

> # inspect results
> names(sp_lines_df)

[1] "ID"    "loads" "width" "datim"

> spplot(sp_lines_df, zcol="ID", lwd=1.5, col.regions =
+          bpy.colors(nlevels(sp_lines_df$ID)))
```

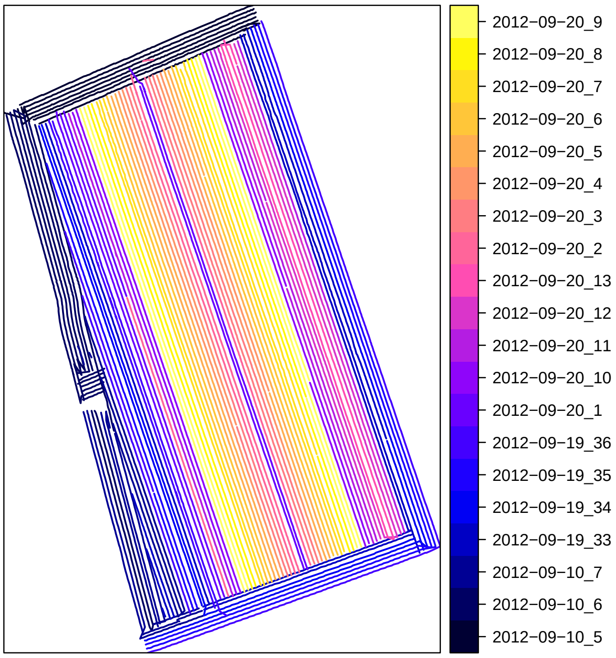

Figure 3: Tracks of the harvester identified by date and load number. Notice the occurrence of overlapping tracks.

Assignment 1:
Inspect the help and code of of the function `CreateHarvestTracks` (for the latter, type the name of the function on the R prompt) and find out what it does and how it works. Describe the function briefly in your own words.

The code below uses buffering to create swaths and to fill narrow spaces between adjacent swaths. Next, the functions difftime and gDifference are used to delete line segments that overlap swaths that were already harvested.

```
> # Buffer lines to make swaths
> sp_polys <- gBuffer(sp_lines_df,byid=T, width=0.5*sp_lines_df$width,
+                     capStyle="FLAT")
> # fill small holes by swelling and shrinking
> sp_polys <- gBuffer(sp_polys, byid=T,id=rownames(sp_polys), width = 2.0)
> sp_polys <- gBuffer(sp_polys, byid=T,id=rownames(sp_polys), width = -2.0)
> sp_polys_df <- SpatialPolygonsDataFrame(sp_polys, sp_lines_df@data)
> # Remove line segments that are within already harvested swaths using
> # gDifference
> tmp_lines <- sp_lines_df@lines # just a list with geometries
> for (i in 2:length(sp_lines_df)){
+   tmline <- sp_lines_df[i,]$datim
+   for (j in 1:(i-1)){
+     tmpoly <- sp_polys_df[j,]$datim
+     if (difftime(tmline, tmpoly, units = "secs") > 0){
+       tmp_line <- SpatialLines(tmp_lines[i], prj_string_RD)
+       if (gIntersects(tmp_line, sp_polys_df[j,])){
+         # compute difference
+         tmp_lines[[i]] <- gDifference(tmp_line, sp_polys_df[j,])@lines[[1]]
+         tmp_lines[[i]]@ID <- sp_lines_df[i,]@lines[[1]]@ID
+       }
+     }
+   }
+ }
> tmp_lines <- SpatialLines(tmp_lines, prj_string_RD)
> cln_lines_df <- SpatialLinesDataFrame(tmp_lines, sp_lines_df@data)
```

5 Now do it yourself

Create a *documented* script with code for the following tasks:

1. buffer the cleaned lines to create harvest blocks, i.e. blocks where a particular load came from;

2. fill small holes *within* these blocks by swelling and shrinking;

3. compute for each block the yield per hectare and add this attribute to the spatial polygons data frame;

4. make a map showing the the yield per hectare of each block, using spplot;

5. export the spatial polygons data frame to display polygon boundaries in Google Earth using `writeOGR`;

6. only if time permits: use functions from the package `plotKML` to display a map of the spatial polygons data frame in Google Earth.

Mail your R code for tasks 1-5 along with your response to assignment 1 to Sytze de Bruin.

References

Bivand, R. S., Pebesma, E. J., & Rubio, V. G. (2013). Applied spatial data analysis with R, .

Pebesma, E. J., & Bivand, R. S. (2005). Classes and methods for spatial data in R. *R News*, *5*, 9–13.