GIS in R: Tutorial 1

Marie Auger-Méthé

1 Good R practices

To be able to reproduce your analyses, it is important to keep track of your files, write down all coding steps, and preferably do version control. People differ in their method to make their analyses reproducible. In this first section, I will introduce my favourite procedures.

1.1 Keep all our files in one place with R Studio projects

Keeping your files organised is particularly important for spatial analyses because some spatial data files need to be kept together (e.g., a shapefile requires a minium of three files: .shp, .shx, and .dbf). While other GIS softwares (e.g., ArcCatalog) focuss on organisation and management of spatial files, R, being a programming language, was not created with this purpose in mind. However, it's really easy to keep track of your files using RStudio and I highly recommend creating a RStudio project for each tutorial, assignement or research project. For those that are unfamiliar with RStudio, you can find information on how to create a project on the RStudio webpage. You can then easily place a copy of your data files in the project folder and saved all of your scripts and results in this same folder. By default a RStudio project will have the main project folder as the working directory, facilitating the import of files into R.

Exercise 1

Create a RStudio project for this tutorial and place the two .csv files found in the GitHub Tutorial1 folder into project folder you just created.

1.2 Make reproducible examples with R scripts

Keeping track of your steps is both important to make your research reproducible and easily shareable, and just because it is nice not to have to start from scratch every time you want to do an analysis. The best way to keep track of your coding steps is to create a .R script that has all code lines written and saved. In RStudio, you can easily run the lines from the R script using the keyboard shortcut (Mac: command+enter, Windows: ctrl+enter, see RStudio website for other shortcuts). I write down every steps in the R script, including the file import function, all of my data handling and management, the analysis functions, and even the sanity checks that I do to make sure that I didn't insert a bug in the code. Very importantly, I comment my code. I tend to write a comment for almost every single line I write. This is extremely helpful, when you go back to an old analysis and is really good when you want to share your code with someone else. To put comments, you simply need to put # at the beggining of the line, e.g.,

```
# This is a comment and if you run the line nothing
# will happen
```

In this document my comments are in purpely-pink, while the R output, which also start with # are in black. Outputs start with ## on purpose, so when you copy and paste the code section into the output is not ran.

Exercise 2

Create an R script for this tutorial and write down all of the code lines. Make sure to add, in addition to my comments, your own comments to help you understand what you did. For those that are taking the class for credit, I want you to send me by e-mail for each tutorial a `R` script that has all of the code from the tutorial, including the code found in the document. I will grade you, based on your code.

1.3 Keep track of your changes with version control

Version control is a system to keep track of the changes you make in you files. Version control is particularly important when coding, as it allows you to go back to previous version of your code and debug more easily. They are many version control methods, the one I use is git, in particular I use GitHub and GitLab. In general you need to pay to have private remote repositories (place on a server that saves the files and changes), but public repositories on GitHub are free of charge. I recommend creating yourself a GitHub account and downloading

the GitHub app (GitHub app for Mac, GitHub for windows). All of the class info is on GitHub and this class is a good way to get you accustomed to git.

2 `Spatial` class

Vector data (i.e., Points, Lines, and Polygons) are handle in `R` using the foundation class `Spatial` associated with the package `sp`. To get a feel for how `Spatial` objects differ from other types of `R` objects, we will import a .csv table with the locations of bioprobes (this is a simplified version of the data available on the Ocean Tracking Network (OTN) website).

```
# Read the .csv file with OTN Sable Island bioprobe
# data
bioPr <- read.csv("sableSealBioP.csv")
```

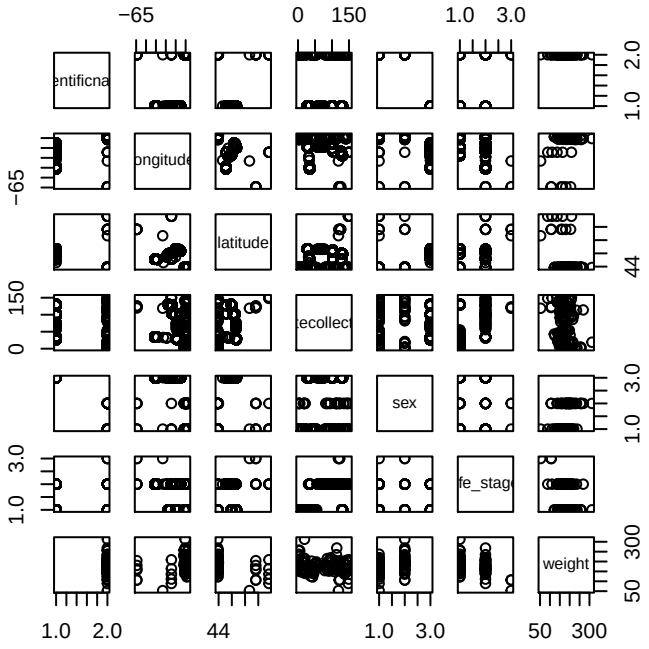
To get a sense for this object we will investigate the class, and some of the generic functions.

```
# What's the class?
class(bioPr)

## [1] "data.frame"

# Can we summarise the information within it?
summary(bioPr)

##             scientificname   longitude         latitude
##  Gadus morhua      :609    Min.   :-64.93   Min.   :43.93
##  Halichoerus grypus:118    1st Qu.:-61.74   1st Qu.:44.56
##                            Median :-60.90   Median :45.01
##                            Mean   :-61.17   Mean   :44.89
##                            3rd Qu.:-60.51   3rd Qu.:45.18
##                            Max.   :-59.74   Max.   :47.80
##
##              datecollected sex         life_stage      weight
##  2011-06-09T00:00:00: 62   F: 81            :255   Min.   : 53.0
##  2012-11-19T00:00:00: 57   M: 37   ADULT    :469   1st Qu.:152.0
##  2012-11-20T00:00:00: 57   U:609   SUB-ADULT:  3   Median :169.0
```

```
##  2013-05-16T00:00:00: 49                                  Mean   :174.7
##  2014-05-13T00:00:00: 45                                  3rd Qu.:196.0
##  2013-11-16T00:00:00: 42                                  Max.   :312.5
##  (Other)            :415                                  NA's   :610
```

One of the generic function is the `plot` function.

```
plot(bioPr)
```



Since we know that the data includes the locations of bioprobes, it would be much better to be able to plot it spatially. To do this, we will trasform this `data.frame` into a `Spatial` object. In particular, since our data is point data, we will transform the `data.frame` into a `SpatialPointsDataFrame`. The simplest way to do so, is by assigning values to the `coords` slot, which is the component that contains the coordinate values.

```
# Load package
library(sp)
# Create new data.frame with the exact same value as
# bioPr
bioPrSP <- bioPr
# Transform this new object into a
# SpatialPointsDataFrame using the columns that
# contain the latitude and longitude values
colnames(bioPrSP)

## [1] "scientificname" "longitude"      "latitude"       "datecollected"
## [5] "sex"            "life_stage"     "weight"

coordinates(bioPrSP) <- ~longitude + latitude
```

Now let's compare the results from the same generic functions.

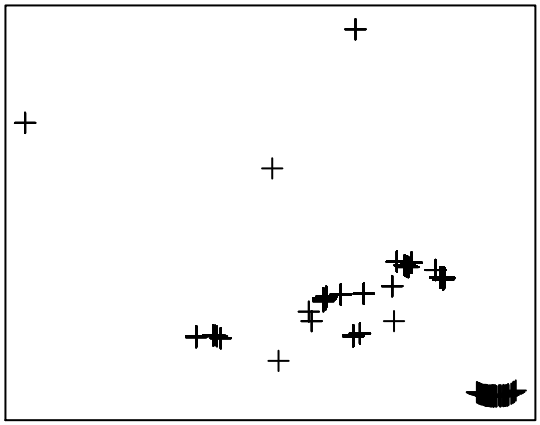
```
class(bioPrSP)

## [1] "SpatialPointsDataFrame"
## attr(,"package")
## [1] "sp"

summary(bioPrSP)

## Object of class SpatialPointsDataFrame
## Coordinates:
##                 min       max
## longitude -64.92774 -59.74377
## latitude   43.92560  47.80300
## Is projected: NA
## proj4string : [NA]
## Number of points: 727
## Data attributes:
##             scientificname             datecollected sex
##  Gadus morhua      :609    2011-06-09T00:00:00: 62   F: 81
```

```
##  Halichoerus grypus:118    2012-11-19T00:00:00: 57   M: 37
##                            2012-11-20T00:00:00: 57   U:609
##                            2013-05-16T00:00:00: 49
##                            2014-05-13T00:00:00: 45
##                            2013-11-16T00:00:00: 42
##                            (Other)            :415
##      life_stage      weight
##           :255   Min.   : 53.0
##  ADULT    :469   1st Qu.:152.0
##  SUB-ADULT:  3   Median :169.0
##                  Mean   :174.7
##                  3rd Qu.:196.0
##                  Max.   :312.5
##                  NA's   :610
```

We see now the bounding box (`bbox`) with the min and max values of the coordinates. We know that the data is not projected and that the `proj4string` slot has a missing value, more on this below. We also see that we have 727 points, which correspond to the 727 rows of the original `data.frame`. Finally we see a summary of the attributes of the points, which is the same summary as the one for the orginal data frame, when you exclude the columns with the coordinates.

But more interestingly, let's look at the `plot` function.

```
plot(bioPrSP)
# With an added box around the figure
box()
```



Here instead of plotting the data values from all the columns the plot shows the location of each bioprobe (each row of the `SpatialPoints` object) in space.

3 Projection

While this is great, this plot is not particularly informative. In part, because we don't have any reference points that tells us where the locations are. In Canada? In New Zealand?

To be able to put reference points, the object need to have a coordinate reference system (CRS). The CRS is set using the `proj4string` slot of the `Spatial` object. When we ran `summary(bioPrSP)` above, we could see both the Is projected is NA and the proj4string is NA, indicating that the data doesn't have a projection. This is not surprising since we have not specified the projection. We can confirm this with the `proj4string` and `is.projected` function.

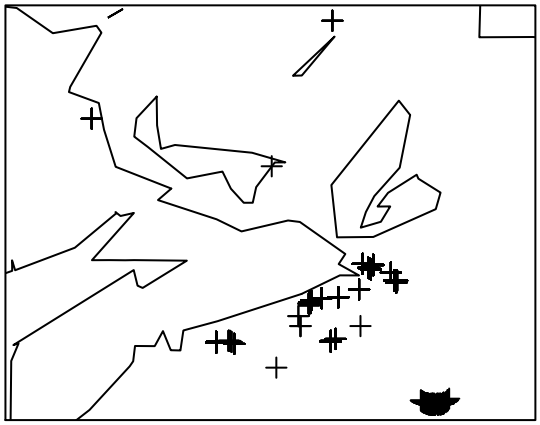
```
# Identifies the object's coordinate reference
# system. This CRS can be a simply a geographic CRS
# or a projected CRS. If not CRS was assigned it will
# have a NA value.
proj4string(bioPrSP)

## [1] NA

# Check whether the object is using a projected CRS.
# NA means the object has no CRS at all.
is.projected(bioPrSP)

## [1] NA
```

As we discussed in the lecture, a geographic CRS represent the location of a point a globe (i.e., a model of the earth), while a projected CRS maps the Earth's on a plane. CRS are essential part of GIS and had been the focus of cartography. One of the most common geographical CRS is WGS84, it is the base of GPS data and I'm assuming that the latitude and longitude data from OTN use this conventional system. To set it, you can use again `proj4string` function.

```
proj4string(bioPrSP) <- CRS("+proj=longlat +datum=WGS84")
# Now bioPrSP has a geographic CRS
proj4string(bioPrSP)

## [1] "+proj=longlat +datum=WGS84"

# But is not projected
is.projected(bioPrSP)

## [1] FALSE
```

Because WGS84 is such a convential system, the many maps avaialble with R packages are generally in this same system and you can use this map to add references to your data (in fact because they are both WGS84, we didn't even need to set the CRS of the `bioPrSP` before hand).

```
# One way to get a sense of where we are is to plot
# the world on top
library(maps)
plot(bioPrSP)
# This is a coarse map
map("world", resolution = 0, add = TRUE)
box()
```



One of the important feature that makes `R` a good language for spatial data, is that there are packages and functions that allow to interpret CRSs and transforms between them. In particular, the `sp` packages depends on the GDAL and PROJ.4 libraries, which allow to interpret different CRSs. These libraries can be loaded at once using the package `rgdal`.

```
library(rgdal)


## rgdal:  version:  0.8-16, (SVN revision 498)
## Geospatial Data Abstraction Library extensions to R successfully loaded
```

```
## Loaded GDAL runtime:  GDAL 1.11.0, released 2014/04/16
## Path to GDAL shared files:  /usr/local/Cellar/gdal/1.11.0/share/gdal
## Loaded PROJ.4 runtime:  Rel.  4.8.0, 6 March 2012, [PJ_VERSION: 480]
## Path to PROJ.4 shared files:  (autodetected)
```

We'll see how to change projects using world maps from the package `maps`. But before I can show how to transform CRS, I need to make the map into a `Spatial` object using the package `maptools`.

```
library(maptools)  # Need maptools for pruneMap and map2SpatialPolygons

## Checking rgeos availability:  TRUE

# This is not a an Spatial object from the package sp
# like bioPrSP.  We remove the limit to limit
# problems on the boundary of the earth when we
# change projection
worldMap <- map("world", resolution = 0, fill = TRUE,
    plot = FALSE, xlim = c(-179, 179), ylim = c(-89,
        89))
class(worldMap)  #Instead it's a object class map

## [1] "map"

# But you can transform it into a spatial object
# using
worldMap <- pruneMap(worldMap, xlim = c(-179, 179))
IDs <- sapply(strsplit(worldMap$names, ":"), function(x) x[1])
worldSP <- map2SpatialPolygons(worldMap, IDs = IDs, proj4string = CRS("+proj=longlat +el
```

Now that we have the map of the world as a `Spatial` object, we can use the `spTransform` to transform accross different CRSs.

```
# Create 4 panels that will be used to display the
# different projections
layout(matrix(1:4, nrow = 2))
```

```r
# Plot the original map (WGS84-Plate Carree)
plot(worldSP)
title(main = "WGS84 - PC")
box()
# Tranform to NAD83 geographic CRS (Plate Carree
# projection), using the EPSG code
worldSPnad83 <- spTransform(worldSP, CRS("+init=epsg:4269"))
is.projected(worldSPnad83)  # Again NAD83 is not projected
```

```
## [1] FALSE
```

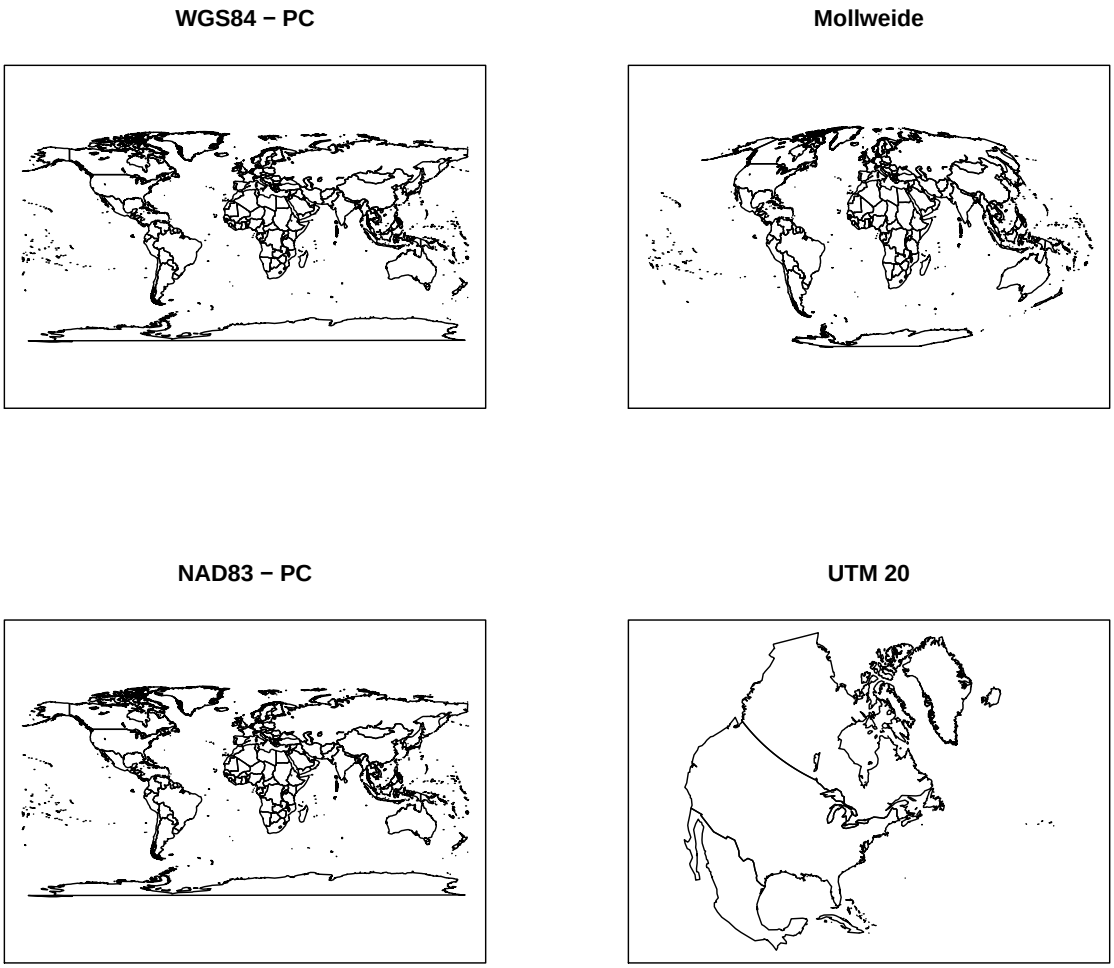
```r
plot(worldSPnad83)
title(main = "NAD83 - PC")
box()
# Using Projected CRSs Projecting to Mollweide
# projection
worldSPmoll <- spTransform(worldSP, CRS("+proj=moll"))
is.projected(worldSPmoll)
```

```
## [1] TRUE
```

```r
plot(worldSPmoll)
title(main = "Mollweide")
box()
# Universal Transverse Mercator (UTM) zone 20 for
# Nova Scotia Need to crop map around Nova Scotia
# Remove the limit
worldMapNS <- map("world", resolution = 0, fill = TRUE,
    plot = FALSE, xlim = c(-100, -20), ylim = c(20, 89))
# But you can transform it into a spatial object
# using
IDs <- sapply(strsplit(worldMapNS$names, ":"), function(x) x[1])
worldNSSP <- map2SpatialPolygons(worldMapNS, IDs = IDs,
    proj4string = CRS("+proj=longlat +ellps=WGS84"))
worldSPutm20 <- spTransform(worldNSSP, CRS("+proj=utm +zone=20 +datum=WGS84"))
is.projected(worldSPutm20)
```

```
## [1] TRUE
```

```
plot(worldSPutm20)
title(main = "UTM 20")
box()
```



Note that WGS84 and NAD83 are very similar. The Mollweide projection is an equal-area projection. The Universal Transverser Mercator (UTM) is a conformal porjection (preserve

angle/shape). In this case I used the UTM zone appropriate for Nova Scotia (20), and it means that spatial data close to the center of the zone are not particularly distorted, but those that are far will be.

It can be hard to find the best projection and even when you know the projection you want to use, it's sometime hard to specify it in R. One way to search for a CRS is using the European Petroleum Survey Group (EPSG) list or through the PROJ.4 info.

```
# First make a data frame of EPSG project codes
EPSG <- make_EPSG()
# Search this data frame using key words, e.g. Nova
# Scotia
EPSG[grep("Nova Scotia", EPSG$note), 1:2]

##     code                              note
## 749 2294 # ATS77 / MTM Nova Scotia zone 4
## 750 2295 # ATS77 / MTM Nova Scotia zone 5

# Canada
EPSG[grep("Canada", EPSG$note), 1:2]

##      code                                        note
## 1801 3347       # NAD83 / Statistics Canada Lambert
## 1802 3348 # NAD83(CSRS) / Statistics Canada Lambert
## 2027 3573         # WGS 84 / North Pole LAEA Canada
## 2328 3978            # NAD83 / Canada Atlas Lambert
## 2329 3979      # NAD83(CSRS) / Canada Atlas Lambert

# NSIDC - National Snow and Ice Data Center
EPSG[grep("NSIDC", EPSG$note), 1:2]

##      code                                                  note
## 1862 3408                            # NSIDC EASE-Grid North
## 1863 3409                            # NSIDC EASE-Grid South
## 1864 3410                           # NSIDC EASE-Grid Global
## 1865 3411          # NSIDC Sea Ice Polar Stereographic North
## 1866 3412          # NSIDC Sea Ice Polar Stereographic South
## 1867 3413 # WGS 84 / NSIDC Sea Ice Polar Stereographic North
```

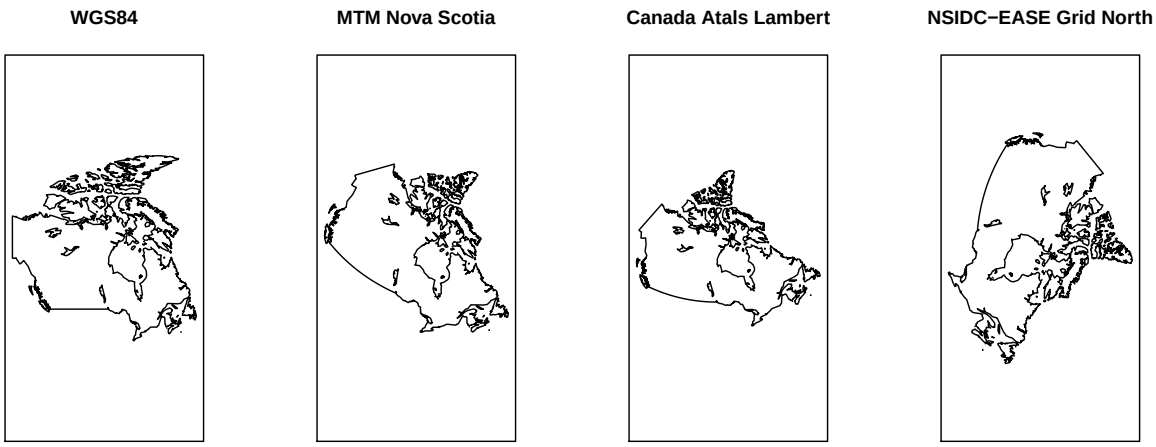
```
## 2324 3973                    # WGS 84 / NSIDC EASE-Grid North
## 2325 3974                    # WGS 84 / NSIDC EASE-Grid South
## 2326 3975                   # WGS 84 / NSIDC EASE-Grid Global
## 2327 3976 # WGS 84 / NSIDC Sea Ice Polar Stereographic South

# You can then use the epsg code in CRS directly or
# use CRS to get proj4string code
CRS("+init=epsg:3408")

## CRS arguments:
##  +init=epsg:3408 +proj=laea +lat_0=90 +lon_0=0 +x_0=0 +y_0=0
## +a=6371228 +b=6371228 +units=m +no_defs

CRS("+init=epsg:3978")

## CRS arguments:
##  +init=epsg:3978 +proj=lcc +lat_1=49 +lat_2=77 +lat_0=49 +lon_0=-95
## +x_0=0 +y_0=0 +datum=NAD83 +units=m +no_defs +ellps=GRS80
## +towgs84=0,0,0

CRS("+init=epsg:2295")

## CRS arguments:
##  +init=epsg:2295 +proj=tmerc +lat_0=0 +lon_0=-64.5 +k=0.9999
## +x_0=5500000 +y_0=0 +a=6378135 +b=6356750.304921594 +units=m
## +no_defs

# You can get further information using projInfo
projI <- projInfo("proj")
projI[projI$name == "laea", ]

##    name                  description
## 52 laea Lambert Azimuthal Equal Area

projI[projI$name == "lcc", ]
```

```
##    name             description
## 60  lcc Lambert Conformal Conic

projI[projI$name == "tmerc", ]

##      name         description
## 113 tmerc Transverse Mercator

# You can also use projInfo to find a projection
projI[grep("Interrupted", projI$description), ]

##    name                  description
## 45  igh Interrupted Goode Homolosine
```

We can use the information we found to transform the CRS.

```
# Canada in different projections
layout(matrix(1:4, nrow = 1))
canadaMap <- map("world", "canada", resolution = 0, fill = TRUE,
    plot = FALSE)
IDs <- sapply(strsplit(canadaMap$names, ":"), function(x) x[1])
canadaSP <- map2SpatialPolygons(canadaMap, IDs = IDs,
    proj4string = CRS("+proj=longlat +ellps=WGS84"))
# Plate carree - WGS84
plot(canadaSP)
title(main = "WGS84")
box()
# MTM Nova Scotia zone 5
canadaSPmtm5 <- spTransform(canadaSP, CRS("+init=epsg:2295"))
plot(canadaSPmtm5)
title(main = "MTM Nova Scotia")
box()
# Canada Atals Lambert
canadaSPal <- spTransform(canadaSP, CRS("+init=epsg:3978"))
plot(canadaSPal)
title(main = "Canada Atals Lambert")
box()
```

```
# NSIDC EASE-Grid North
canadaSPnsidc <- spTransform(canadaSP, CRS("+init=epsg:3408"))
plot(canadaSPnsidc)
title(main = "NSIDC-EASE Grid North")
box()
```



The most noticable difference here is the difference in the shape of Canada accross the WGS4 CRS and the other 3, which are projected. One other very noticable difference is that the center of the CRS for the maps differ a lot (that's why Canada's North is not always at the top and center of the map).

When you have locations that are outside CRS, e.g., if there is a typo in your data or if the data comes from highly error-prone system (e.g. light curve location system), you will not be able to assign the CRS.

```
# Create a duplicate data.frame
bioPr2 <- bioPr
# We will put a non-sensical coordinate in the data
# frame
bioPr2$longitude[1] <- -300  # -300 degree longitude makes no sense
head(bioPr2)

##   scientificname longitude latitude       datecollected sex life_stage
```

```
## 1   Gadus morhua  -300.000   44.953 2014-05-13T00:00:00   U
## 2   Gadus morhua   -61.752   44.953 2014-05-13T00:00:00   U
## 3   Gadus morhua   -61.778   44.945 2014-05-15T00:00:00   U
## 4   Gadus morhua   -61.745   44.978 2014-05-14T00:00:00   U
## 5   Gadus morhua   -61.745   44.978 2014-05-14T00:00:00   U
## 6   Gadus morhua   -61.745   44.978 2014-05-14T00:00:00   U
##   weight
## 1     NA
## 2     NA
## 3     NA
## 4     NA
## 5     NA
## 6     NA

# Now let's try to create the spatial object again,
# using the same steps
bioPrSP2 <- bioPr2
coordinates(bioPrSP2) <- ~longitude + latitude
proj4string(bioPrSP2) <- CRS("+proj=longlat +datum=WGS84")

## Error in `proj4string<-`(`*tmp*`, value = <S4 object of class
structure("CRS", package = "sp")>):  Geographical CRS given to
non-conformant data:  -300
```

We get an error because the longitude value -300 doesn't exist in the WGS84 CRS. As a result, the object remains without a coordinate system.

```
proj4string(bioPrSP2)

## [1] NA
```

If you remove the row with the non-conformat data, you'll be able to assign the coordinate system.

```
# Get the index for the non-conformant coordinates
ncLonLat <- which(coordinates(bioPrSP2) == -300, arr.ind = TRUE)
# We know it should be in row 1, col 1, since we've
```

```
# put it there
ncLonLat


##      row col
## [1,]   1   1


# Remove that row
bioPrSP2 <- bioPrSP2[-ncLonLat[, 1], ]
# Now assign the coordinate system
proj4string(bioPrSP2) <- CRS("+proj=longlat +datum=WGS84")
```

The coordinates in the .csv files were in decimal degrees. However, you might get values that are in degrees, minutes, and seconds. You can convert these using `char2dms` and `as`.

```
# Halifax Lat Lon in Decimal, Minute, and Second:
Hchar <- c("44d38'55.9716\"N", "63d34'31.1232\"W")
# Our string has the default spearator for d, m, s,
# but these could be specified, see ?char2dms
Hdms <- char2dms(Hchar)
Hdd <- as(Hdms, "numeric")
Hdd


## [1]  44.64888 -63.57531
```

4 Attributes: subsetting

What if we wanted to create a object that only include a subset of the location. In this example we would like to only have the Atlantic cod (*Gadus morhua*) locations.

```
# Only take the points that have 'Gadus morhua' in
# the column scientificname
codPrSP <- bioPrSP[bioPrSP$scientificname == "Gadus morhua",
    ]
# We now have 0 Halichoerus grypus, but we can see
# that the subset will keep the coordinate system,
```

```
# but will update the bounding box
summary(codPrSP)
```

```
## Object of class SpatialPointsDataFrame
## Coordinates:
##              min      max
## longitude -63.12 -60.5000
## latitude   44.30  45.3475
## Is projected: FALSE
## proj4string : [+proj=longlat +datum=WGS84]
## Number of points: 609
## Data attributes:
##             scientificname             datecollected sex
##  Gadus morhua      :609    2011-06-09T00:00:00: 62   F:  0
##  Halichoerus grypus:  0    2012-11-19T00:00:00: 57   M:  0
##                            2012-11-20T00:00:00: 57   U:609
##                            2013-05-16T00:00:00: 49
##                            2014-05-13T00:00:00: 45
##                            2013-11-16T00:00:00: 42
##                            (Other)            :297
##      life_stage      weight
##           :200   Min.   : NA
##  ADULT    :409   1st Qu.: NA
##  SUB-ADULT:  0   Median : NA
##                  Mean   :NaN
##                  3rd Qu.: NA
##                  Max.   : NA
##                  NA's   :609
```

```
# For example compare
bbox(bioPrSP)
```

```
##                 min       max
## longitude -64.92774 -59.74377
## latitude   43.92560  47.80300
```

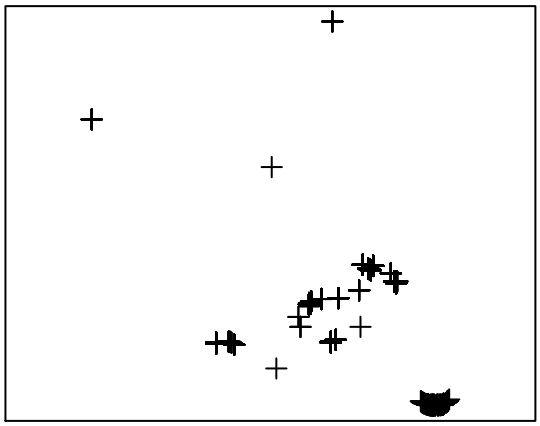
```
bbox(codPrSP)
```

```
##               min      max
## longitude -63.12 -60.5000
## latitude   44.30  45.3475
```

The object class we have been using so far, `SpatialPointsDataFrame`, is actually a more complex level class for point data. I've presented first, because it's the class that I find the more useful. Most of my data consist of points with some kind of attributes. However, there may be cases where the data you have is only coordinates, with no further attributes associated with them. In this case you would use a `SpatialPoints` class. For example, let's say we only have the locations of the bioprobes, with no information on the species, sex, age, etc..

```
# We are making a bit of an artifical example here by
# first getting the coordinates of the bioPrSP
bioPrCoord <- coordinates(bioPrSP)
# Now let assume that this new object is the only
# information we have
summary(bioPrCoord)

##    longitude         latitude
##  Min.   :-64.93   Min.   :43.93
##  1st Qu.:-61.74   1st Qu.:44.56
##  Median :-60.90   Median :45.01
##  Mean   :-61.17   Mean   :44.89
##  3rd Qu.:-60.51   3rd Qu.:45.18
##  Max.   :-59.74   Max.   :47.80

# We can create a SpatialPoints object as follow
bioPrCoordSP <- SpatialPoints(bioPrCoord, proj4string = CRS("+proj=longlat +ellps=WGS84"
# This now a SpatialPoints object, without any
# attribute
summary(bioPrCoordSP)

## Object of class SpatialPoints
## Coordinates:
##                 min       max
## longitude -64.92774 -59.74377
```

```
## latitude   43.92560  47.80300
## Is projected: FALSE
## proj4string : [+proj=longlat +ellps=WGS84]
## Number of points: 727

# That can be plotted just like before
plot(bioPrCoordSP)
box()
```



It can happen that you may have the coordinated information coming from a different file than the file in which you have information. In which case, you can put the two together using the `match.ID` option of `SpatialPointsDataFrame`. The idea here is similar to matching rows of database based on their unique key. So here the unique key is the name of the row. So for example let's say we want to match the coordinates of the bioprobes found in the matrix `bioPrCoord` back to the data.frame `bioPr`.

```r
# First indentify whether the objects have row name
str(row.names(bioPrCoord))
```

```
##  NULL
```

```r
str(row.names(bioPr))
```

```
##  chr [1:727] "1" "2" "3" "4" "5" "6" "7" "8" "9" ...
```

```r
# While bioPr has number values from 1-727,
# bioPrCoord doesn't.  We know that bioPrCoord should
# be in the same order as bioPr, becasue we've
# created bioPrCoord from bioPr. So I think we could
# use directly
bioPrSP3 <- SpatialPointsDataFrame(bioPrCoord, bioPr,
    proj4string = CRS("+proj=longlat +ellps=WGS84"),
    match.ID = TRUE)
# Note that match.ID=TRUE is the default value, so
# you don't need to specify it.  We can see that the
# rows are adequately associated with the
# coordinates. Just compare the longitude and
# latitude columns to the coordinates
bioPrSP3[1:5, ]
```

```
##         coordinates scientificname longitude latitude       datecollected
## 1 (-61.752, 44.953)   Gadus morhua   -61.752   44.953 2014-05-13T00:00:00
## 2 (-61.752, 44.953)   Gadus morhua   -61.752   44.953 2014-05-13T00:00:00
## 3 (-61.778, 44.945)   Gadus morhua   -61.778   44.945 2014-05-15T00:00:00
## 4 (-61.745, 44.978)   Gadus morhua   -61.745   44.978 2014-05-14T00:00:00
## 5 (-61.745, 44.978)   Gadus morhua   -61.745   44.978 2014-05-14T00:00:00
##   sex life_stage weight
## 1   U                NA
## 2   U                NA
## 3   U                NA
## 4   U                NA
## 5   U                NA
```

```r
# Now just for fun let's mix up the row on
# bioPrCoord, in this case if we want the row.names
# to be assigned.
row.names(bioPrCoord) <- 1:727
# Then we can sample at random and the row name will
# be the same
bioPrCoordM <- bioPrCoord[sample(1:727, 727), ]
# We see now that, compare to the original, the row
# are mixed randomly. But, the original row names
# remain associated with their original row
head(bioPrCoordM)
```

```
##     longitude latitude
## 596 -60.87983  45.2915
## 247 -63.12000  44.5600
## 257 -60.51000  45.1700
## 410 -60.59000  45.2600
## 273 -63.12000  44.5600
## 245 -63.12000  44.5600
```

```r
head(bioPrCoord)
```

```
##   longitude latitude
## 1   -61.752   44.953
## 2   -61.752   44.953
## 3   -61.778   44.945
## 4   -61.745   44.978
## 5   -61.745   44.978
## 6   -61.745   44.978
```

```r
# Now we can create a spatial object that will use
# the row names, even if mixed randomly, to match the
# data.frame of attributes to the coordinates
bioPrSP4 <- SpatialPointsDataFrame(bioPrCoordM, bioPr,
    proj4string = CRS("+proj=longlat +ellps=WGS84"),
    match.ID = TRUE)
# Now the order is mixed up but the coordinates still
```

```
# correspond to the good row of the attributes.
bioPrSP4[1:5, ]


##              coordinates scientificname longitude latitude
## 596 (-60.87983, 45.2915)   Gadus morhua -60.87983  45.2915
## 247      (-63.12, 44.56)   Gadus morhua -63.12000  44.5600
## 257      (-60.51, 45.17)   Gadus morhua -60.51000  45.1700
## 410      (-60.59, 45.26)   Gadus morhua -60.59000  45.2600
## 273      (-63.12, 44.56)   Gadus morhua -63.12000  44.5600
##           datecollected sex life_stage weight
## 596 2010-11-30T06:00:00   U                NA
## 247 2011-06-09T00:00:00   U      ADULT     NA
## 257 2012-11-19T00:00:00   U      ADULT     NA
## 410 2013-05-15T00:00:00   U      ADULT     NA
## 273 2011-06-10T00:00:00   U      ADULT     NA
```

Note that if the row names don't match up, you won't be able to create a `SpatialPointsDataFrame`

```
# Creating a duplicate of the coordinate matix
bioPrCoordN <- bioPrCoord
# Giving new names, which don't match with one of the
# 727 orginal names, to the first 5 rows
row.names(bioPrCoordN)[1:5] <- 1001:1005
bioPrSP5 <- SpatialPointsDataFrame(bioPrCoordN, bioPr,
    proj4string = CRS("+proj=longlat +ellps=WGS84"),
    match.ID = TRUE)


## Error in SpatialPointsDataFrame(bioPrCoordN, bioPr, proj4string =
CRS("+proj=longlat +ellps=WGS84"), :  row.names of data and coords do not
match
```

Note that they are other ways in which you can construct a `SpatialPointsDataFrame`, including by linking a `data.frame` to a `SpatialPoints` object.

```
# Here we don't need to specify the coordinate system
# and projection, because the SpatialPoints object
```

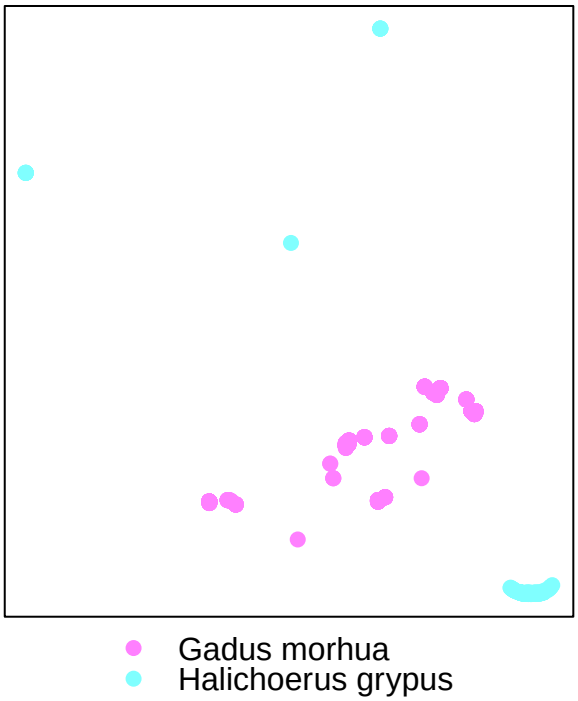
```
# already has this information.
bioPrSP6 <- SpatialPointsDataFrame(bioPrCoordSP, bioPr)
# It should be the same has linking the matrix of
# coordinate
identical(bioPrSP3, bioPrSP6)

## [1] TRUE
```

A final way which is similar to the one presented in my orginial example, is using the `coordinates` function.

```
# Make a duplicate of the data.frame
bioPrSP7 <- bioPr
coordinates(bioPrSP7) <- bioPrCoord
class(bioPrSP7)

## [1] "SpatialPointsDataFrame"
## attr(,"package")
## [1] "sp"

# To add the projection, you can just use proj4string
proj4string(bioPrSP7) <- CRS("+proj=longlat +ellps=WGS84")
```

Ok, so I've presented multiple ways to associated attributes to locations and create a `SpatialPointsDataFrame`. But why should we bother? Well, one of the great thing about attributes is that they can be used to create symbols in plots.

```
spplot(bioPrSP, zcol = "scientificname")
```



But more on this on the next class!

Exercise 3

Create a spatial object with the data found in nbCod.csv, which is a slightly altered version from the Animal file available on the Shippagan, NB: Code tagging project from OTN, see project website. The object should be assigned the appropriate CRS.

Exercise 4

Transform the CRS of the `Spatial` object you've created in the previous exercise an Universal Transverse Mercator projection (use the appropriate zone). Plot this transformed object with a map of the world overlaid on top of it.