GIS in R: Tutorial 3

Marie Auger-Méthé

1 SpatialPolygons

1.1 Components of SpatialPolygons

The last tutorial introduced SpatialLines and some of the basic elements of SpatialPolygons. In particular, it discussed in detail the hierarchy of the elements needed for both of these classes. As for SpatialLines, SpatialPolygons have a set of nested of classes. However, SpatialPolygons are more complex than SpatialLines and all levels of object classes are composed of more slots than their SpatialLines analogue.

The fundamental class of SpatialPolygons is the Polygon class. As you can see below, a Polygon object is similar to a Line object, but with a few extra slots.

```
library(sp)
getSlots("Polygon")

##     labpt      area      hole   ringDir    coords
## "numeric" "numeric" "logical" "integer"  "matrix"

getSlots("Line")

##   coords
## "matrix"
```

As for the Line class, the Polygon class is the fundamental class because it's the class that has a slot for the coordinates (coords). However, the Polygon class has four additional slots. Just like the bbox slot of Spatial* objects, some of the slots of Polygon objects are generated automatically: 1) the slot labpt is the label point and has the coordinates of the centroid of the polygon; 2) the slot area has the area of the polygon in the metric of the coordinates; and 3) the slot ringDir has the ring direction. However, the slot hole, which indicates whether the Polygon object is a hole (e.g. the a lake in a land polygon) is one slot that sometimes need to be manipulated. More on holes below.

The next level of the hierarchy of polygon object class, the Polygons class, also have more slots than the lines analogue, the Lines class.

```r
getSlots("Polygons")
```

```
##    Polygons   plotOrder       labpt          ID        area
##      "list"   "integer"   "numeric" "character"   "numeric"
```

```r
getSlots("Lines")
```

```
##       Lines          ID
##      "list" "character"
```

The two most important slots are those that are similar for both object class: 1) the list of `Polygon` or `Line`; and 2) the `ID` slot. As we learned in the previous tutorial, the `ID` slot can only contain one single element describing the associated list of `Line` or `Polygon`. As for the `Polygon` class, the slots `labpt` and `area` are generally generated automatically. The `labpt` is the centroid of the largest `Polygon` constituting a `Polygons` object and the `area` of a `Polygons` object is the sum of all `Polygon` objects it contains. Note that the `labpt` is useful when we want to label polygons with the character string saved in the `ID` slot. The slot `plotOrder` is the order in which the `Polygon` should be plotted. The plot order is usually generated automatically based on the size of the `Polygon` object, largest first. The `plotOrder` slot is also present in the next level of the hierarchy, i.e., in `SpatialPolygons` objects. In fact, this extra slot (`plotOrder`), is the only difference in the structure of `SpatialPolygons` and `SpatialLines`.

```r
getSlots("SpatialPolygons")
```

```
##    polygons   plotOrder        bbox proj4string
##      "list"   "integer"    "matrix"       "CRS"
```

```r
getSlots("SpatialLines")
```

```
##       lines        bbox proj4string
##      "list"    "matrix"       "CRS"
```

Just like a `SpatialLines` object needs a list of `Lines` object, the `SpatialPolygons` take a list of `Polygons`. Note that, just like `SpatialLines` objects, `SpatialPolygons` are the level at which the coordinate reference system (CRS) is set.

The final level of the hierarchy is the `SpatialPolygonsDataFrame`. As for the objects of class `SpatialLinesDataFrame`, this level incorporates attributes.

```r
getSlots("SpatialPolygonsDataFrame")

##          data     polygons    plotOrder         bbox  proj4string
## "data.frame"       "list"    "integer"     "matrix"        "CRS"

getSlots("SpatialLinesDataFrame")

##          data        lines         bbox  proj4string
## "data.frame"       "list"     "matrix"        "CRS"
```

Just like the `SpatialLinesDataFrame`, you can only assign data value to each `Polygons` object (you cannot assign data value to the `Polygon` objects that are lumped into a single `Polygons` object).

In brief, the `Polygon` object level is used to assign the coordinates, the `Polygons` object level is used to assign the ID, the `SpatialPolygons` object level is used to assign the CRS, and the `SpatialPolygonsDataFrame` object level is used to assign attributes to `Polygons` object.

1.2 Creating `SpatialPolygons`

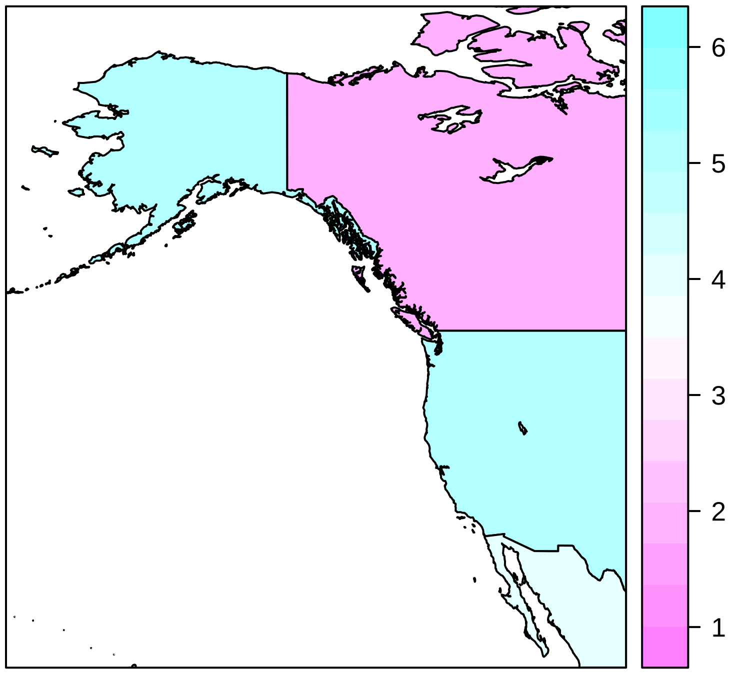
It's rare that you need to create `SpatialPolygons` from their raw coordinates. `SpatialPolygons` data are often imported from other sources or created based on similar object classes (e.g. maps of the world).

1.2.1 Importing shapefiles

First, we will import a shapefile with a `SpatialPolygonsDataFrame`. Shapefiles are files used by ArcGIS (see `http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#/What_is_a_shapefile/005600000002000000/` for more information). Shapefiles require a set of different files: 1) the .shp file, which keeps track of the geometry (for `SpatialPolygons` similar of the `Polygon` level), 2) a .shx file, which keeps track of the index of feature geometry (for `SpatialPolygons` similar to the `Polygons` level), and 3) .dbf file that stores the attributes of the features (for `SpatialPolygons` similar to the `data.frame` in the `SpatialPolygonsDataFrame`). Other files can be assosciated with the shapefiles. For example, the .prj file keep tracks of the CRS (for `SpatialPolygons` similar to the CRS in the `SpatialPolygons`). All of these file should have the same name and only differ in their extension and they need to be saved in the same folder.

Here, we will load a `SpatialPolygonsDataFrame` with a map of the west coast of north america. To import a shapefile in `R`, you need the package `rgdal`.

```r
# Load rgdal package which has the function readOGR
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)

wC <- readOGR(dsn = ".", layer = "WestCoast")

## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "WestCoast"
## with 6 features and 1 fields
## Feature type: wkbPolygon with 2 dimensions

# This already a SpatialPolygonsDataFrame
class(wC)

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

# It already has a CRS because it had a WestCoast.prj
# file
proj4string(wC)

## [1] "+proj=longlat +ellps=WGS84 +no_defs"

# lets' plot it
spplot(wC)
```



Note that other file types can be loaded with `readOGR` including google earth files (.kml).

1.2.2 `SpatialPolygons` from `SpatialPoints`

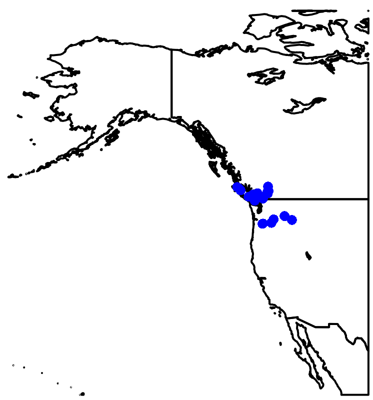
While creating `SpatialPolygons` from coordinates is rarely needed, there is one exception. I often need to create a `SpatialPolygons` object to represent my study area on a bigger map. Something I'll show you how to do using the Ocean Tracking Network (OTN) Kintama project data (more informationon the project website: `http://members.oceantrack.org/data/discovery/KNTM.htm`). The file kintama.csv is a modified version from the file animal.csv found on the OTN public data website (`http://members.oceantrack.org/data/discovery/bypublic.htm#K`). First we will create a `SpatialPointsDataFrame`.

```
# Import data
kntm <- read.csv("kintama.csv")
head(kntm)
```

```
##              scientificname longitude latitude       datecollected
## 1     Oncorhynchus kisutch -126.6469 50.23139 2004-06-14T22:30:00
## 2       Oncorhynchus nerka -121.9797 49.07988 2007-05-16T22:00:00
```

```
## 3       Oncorhynchus nerka -121.9797 49.07988 2007-05-16T22:00:00
## 4         Salvelinus malma -127.3509 50.67511 2004-06-05T05:00:00
## 5 Oncorhynchus tshawytscha -120.2138 45.72360 2010-05-02T16:34:00
## 6       Oncorhynchus nerka -121.9797 49.07988 2007-05-16T22:00:00
##   timeofday          stock life_stage length
## 1  22.50000 Nimpkish River             0.144
## 2  22.00000    Cultus Lake             0.181
## 3  22.00000    Cultus Lake             0.191
## 4   5.00000    Keogh River   JUVENILE  0.188
## 5  16.56667        Unknown   JUVENILE  0.149
## 6  22.00000    Cultus Lake             0.202
```

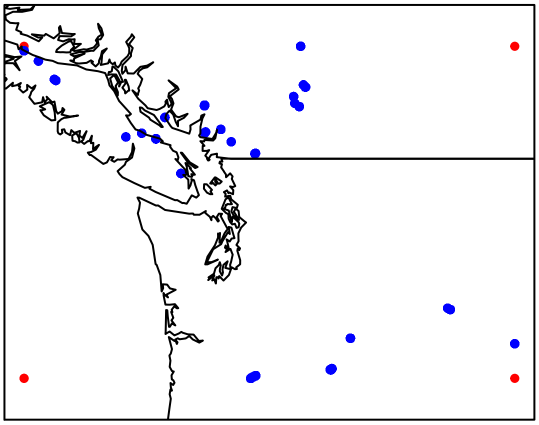
```r
# Create a SpatialPointsDataFrame object
coordinates(kntm) <- ~longitude + latitude
proj4string(kntm) <- CRS("+proj=longlat +datum=WGS84")
# Plot the point on the map of west coast
plot(wC)
plot(kntm, col = "blue", pch = 19, add = TRUE, cex = 0.5)
```



While we can see where the points are, a nicer way to show the study area would be to create a `SpatialPolygons` that represent the study area (i.e., a rectangle on the extent of the study area). For this, we can use the bounding box (`bbox`) from our `SpatialPoints`. First we need to get the coordinates of the bounding box and place them in the order that we would trace the polygon in.

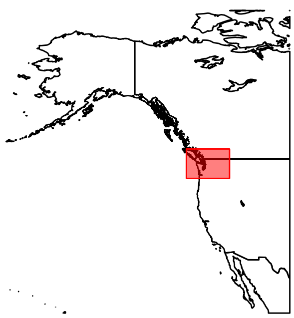
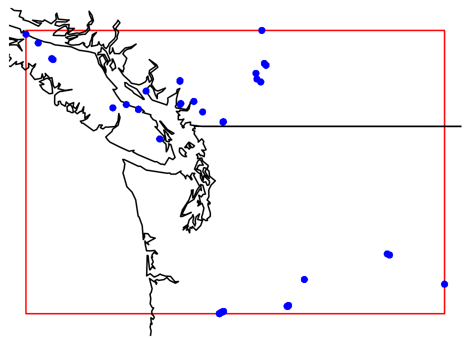
```
# Get the bounding box of the SpatialPoints object
bbox(kntm)

##                   min        max
## longitude -127.35088 -115.93472
## latitude    45.59333   50.74564

# We know that the SpatialPolygons should have a
# points for all combination of these points. You can
# use expand.grid to do this
studyA <- expand.grid(long = bbox(kntm)["longitude",
    ], lat = bbox(kntm)["latitude", ])
# See it has all the combination of longitude and
```

```r
# latitude
studyA

##        long      lat
## 1 -127.3509 45.59333
## 2 -115.9347 45.59333
## 3 -127.3509 50.74564
## 4 -115.9347 50.74564

# However, that's not enough. The coordinates should
# be in order in which you would trace the polygon
# (the ring direction). In most cases it should be
# clockwise. So here, 1: c(min(lon), min(lat)); 2:
# c(min(lon), max(lat)); 3: c(max(lon)), max(lat));
# 4: c(max(lon), min(lat))
studyA <- studyA[c(1, 3, 4, 2), ]
# See we change the order
studyA

##        long      lat
## 1 -127.3509 45.59333
## 3 -127.3509 50.74564
## 4 -115.9347 50.74564
## 2 -115.9347 45.59333

# The last trick, is that a polygon needs to be a
# closed line so the first and last locations need to
# be the same.  Here we repeat the first location at
# the end and bind it with rbind
studyA <- rbind(studyA, studyA[1, ])
# We can make these coordinates a SpatialPoints
# object
coordinates(studyA) <- ~long + lat
proj4string(studyA) <- proj4string(kntm)
# Let's plot these
plot(studyA, col = "red", pch = 19, cex = 0.5)
plot(kntm, col = "blue", pch = 19, add = TRUE, cex = 0.5)
plot(wC, add = TRUE)
box()
```



Now we can create a SpatialPolygons from the SpatialPoints. We are using the same hierarchy as for the SpatialLines.

```
studyASP <- SpatialPolygons(list(Polygons(list(Polygon(studyA)),
    ID = "Study site")), proj4string = CRS(proj4string(studyA)))
# Let's plot it
layout(matrix(1:2, nrow = 1))
# First panel: zoomed out view
plot(wC)
# I'm using rgb to make stranparant colour
plot(studyASP, border = "red", col = rgb(1, 0, 0, 0.5),
    add = TRUE)
# Second panel: zoomed in view
plot(studyASP, border = "red")
plot(wC, add = TRUE)
plot(kntm, col = "blue", pch = 19, add = TRUE, cex = 0.5)
```

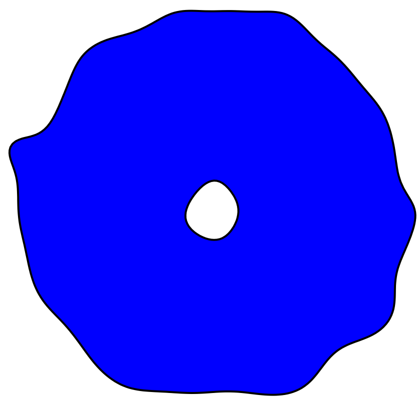
 

1.3 Holes in polygons

R is not a true GIS and does not represent polygons by their topology. The package sp does not check whether the lines cross or polygons have errors. Unlike other GIS software, the sp functions associated with polygons do not check whether the features are simple. This is has repercussion on how holes in SpatialPolygons are handled (e.g., the great bear lake in the Northwest Territories, Canada, see first figure of this tutorial). The package sp marks whether a Polygon as hole by using the hole slot and the ring direction (clockwise for nonholes, represented with 1, and anti-clokwise for holes, represeted with -1.).

Here we will investigate a SpatialPolygonsDataFrame I've created that has a hole in it.

```
# Read the simulated polygons
HR <- readOGR(dsn = ".", layer = "hrSP")

## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "hrSP"
## with 1 features and 1 fields
## Feature type: wkbPolygon with 2 dimensions

# Plot the SpatialPolygonsDataFram
plot(HR, col = "blue")
```



```
# We want to look at each Polygon object. To do this
# we need to get the Polygons out of the
# SpatialPolygonsDataFrame.
HR <- slot(HR, "polygons")[[1]]
# We then need the list of Polygon objects out of the
# Polygons
hr <- slot(HR, "Polygons")
# Now we can see that there are 2 Polygon objects
length(hr)

## [1] 2

# The first Polygon is not a hole
slot(hr[[1]], "hole")

## [1] FALSE

# So its ring direction is clockwise: 1
slot(hr[[1]], "ringDir")
```

```
## [1] 1

# We can also look at it's area
slot(hr[[1]], "area")

## [1] 197.625

# Now we can look at the secon Polygon. Is it a hole
slot(hr[[2]], "hole")

## [1] TRUE

# So its ring should be anticlokwise: -1
slot(hr[[2]], "ringDir")

## [1] -1

# What's its area, much smaller
slot(hr[[2]], "area")

## [1] 3.840118
```

2 Visualisation

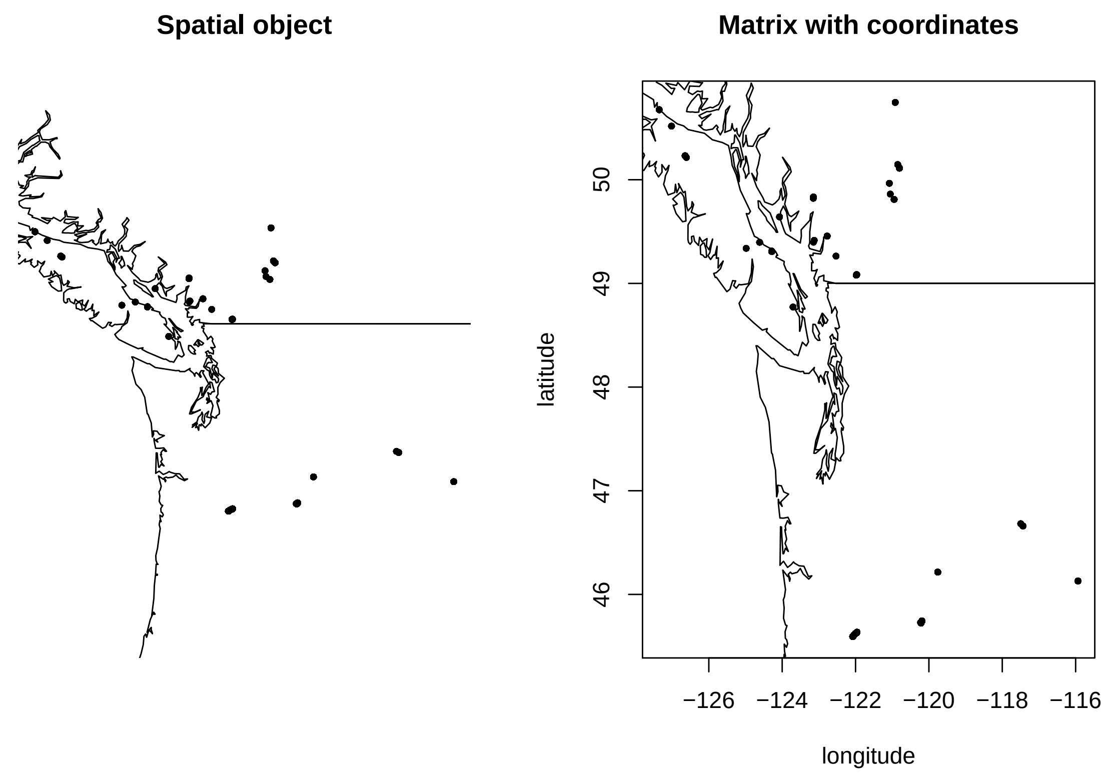
Traditional graphics, done with the function `plot`, are made incrementally. Graphic elements are added with a set of different functions. The `sp` package provide a traditional `plot` function.

Trellis graphics (e.g., those associated with the package `lattice`) allow to plot high-dimensional data by providing conditioning plots, sets of plots with shared axis. The `sp` package also provide a `spplot` function that use the trellis system from the `lattice` package.

2.1 Traditional plots

One advantage of plotting a `Spatial*` object is the szie of the axis will be automatically handled adequatly. For object with a projected CRS, the units in the x and y coordinates will be of equal length. For object with a geographical CRS, a sensible aspect ratio will be used. Note that although the default aspect ratio of `Spatial*` objects is generally adequate, you still adjust it using the `asp` argument (see `?asp`). The default aspect ratio chosen for `Spatial*` differ from the default chosen when plotting non spatial objects.

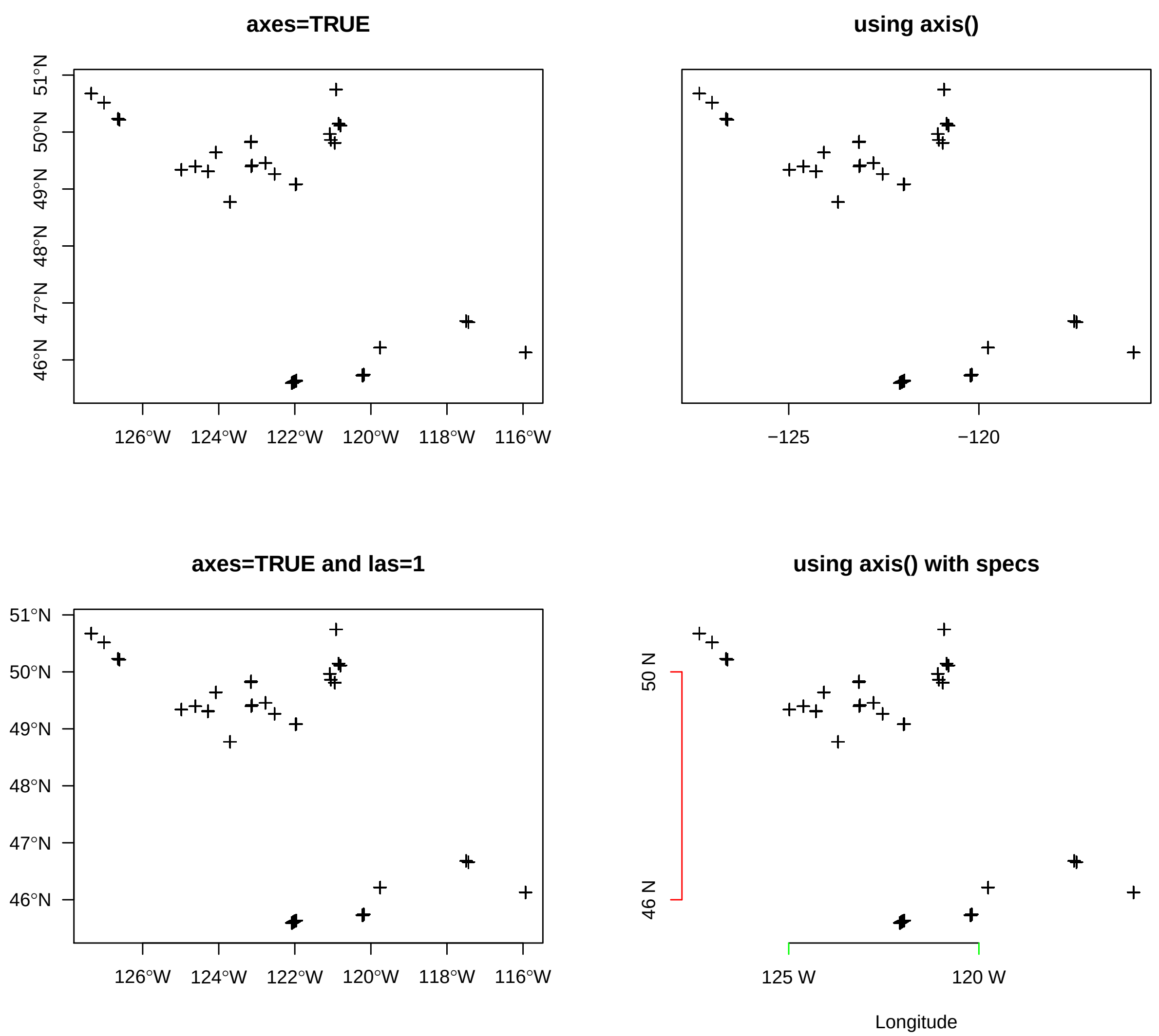
```r
# Make 2 panels
layout(matrix(1:2, nrow = 1))
# Plot Spatial* object
plot(kntm, pch = 19, cex = 0.5)
plot(wC, add = TRUE)
title(main = "Spatial object")
# Plot normal object, use coordinates
plot(coordinates(kntm), pch = 19, cex = 0.5)
plot(wC, add = TRUE)
title(main = "Matrix with coordinates")
```



You can notice a few differences. The plot with the matrix of coordinates looks distorded, but not the plot of the `Spatial*` object. Another difference is that the `Spatial*` object doesn't plot the axis automatically, but the plot of the matrix does.

To add axis on the a plot of `Spatial*` object, you can use the argument `axes`. Some of the `par` arguments used to change the appearance of the plot can also be used directly in the `plot` function, e.g., make the y-axis labels horizontal rather than vertical using the `las` argument. You can use the function `axis` to add an axis with a specific format. Note that you can add axis title and main graphic title using the function `title`.

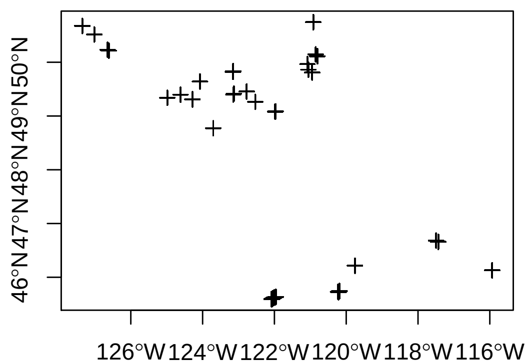
```
layout(matrix(1:4, nrow = 2))
# Plot a spatial object with axes
plot(kntm, axes = TRUE)
# Add plot title
title("axes=TRUE")
# Make the y-axis label horizontal
plot(kntm, axes = TRUE, las = 1)
# Add title
title("axes=TRUE and las=1")
# Make plot and add x-axis with axis
plot(kntm)
# 1 specify that it's the x-axis
axis(1, at = c(-125, -120))
# add box
box()
# Add title
title("using axis()")
# Make a plot and add x-axis with specific label
plot(kntm)
# x-axis with specified label and color for ticks
axis(1, at = c(-125, -120), labels = c("125 W", "120 W"),
    col.ticks = "green")
# y-axis color for both line and ticks
axis(2, at = c(46, 50), labels = c("46 N", "50 N"), col = "red")
# add title and x-axis title
title("using axis() with specs", xlab = "Longitude")
```

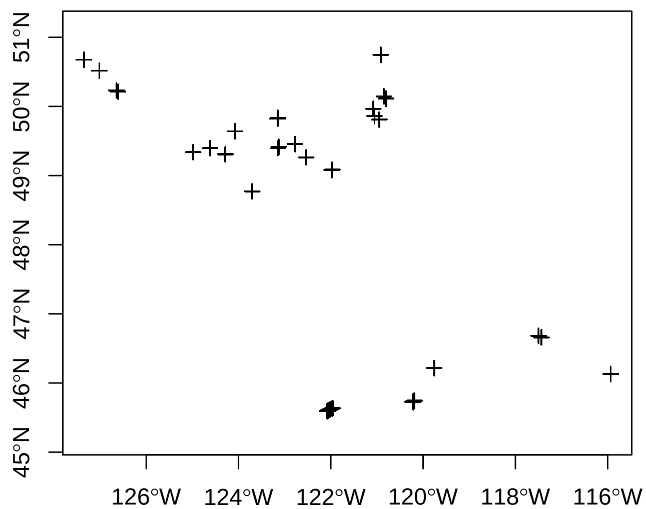


Margin size and other ploting parameters can be adjusted for all graphics using the function `par`. For example, the argument `mar` can be used to set the margin size in units of height of a line of text. The order of the argument `mar` is bottom, left, top, and right margins.

```
layout(matrix(1:2, nrow = 1))
# plot with original margin
plot(kntm, axes = TRUE)
```

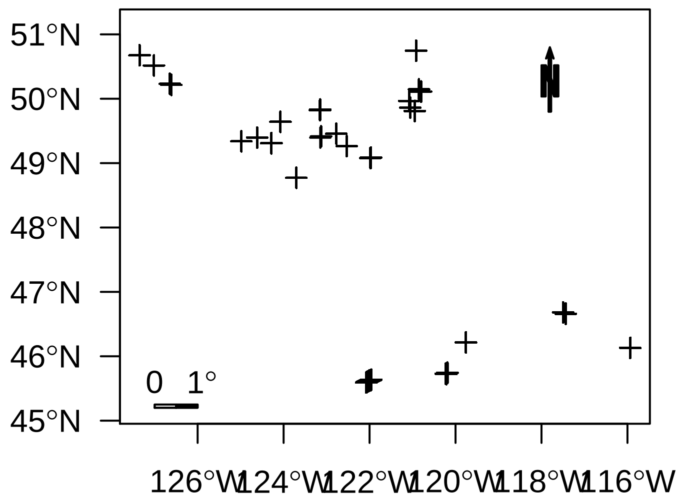
```r
par(mar = c(3, 1, 1, 1))
plot(kntm, axes = TRUE)
```





Often when we plot spatial data, we want to add features such as the North arrow and/or a scale bar. This can be done using the function `SpatialPolygonsRescale`.

```r
# Plot the SpatialPoints
plot(kntm, axes = TRUE, las = 1)
# Add North arrow
SpatialPolygonsRescale(layout.north.arrow(), offset = c(-118,
    49.8), plot.grid = FALSE)
# Add scale bar
SpatialPolygonsRescale(layout.scale.bar(), offset = c(-127,
    45.2), fill = c("transparent", "black"), plot.grid = FALSE)
# add the value for scale
text(x = -126.9, y = 45.6, "0 ")
text(x = -125.9, y = 45.6, expression(1 * degree))
```

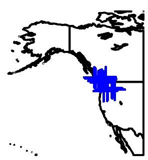


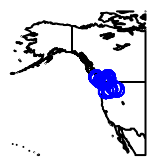

```
# You can use the locator to choose the location,
# (uncomment the code below)
# SpatialPolygonsRescale(layout.north.arrow(),offset=locator(1),
# plot.grid=FALSE)
# SpatialPolygonsRescale(layout.scale.bar(), offset =
# locator(1), fill = c('transparent', 'black'),
# plot.grid = FALSE)
# text(locator(1), '0')
# text(locator(1), expression(1*degree))
```

Multiple `Spatial*` objects can be added to a plot using the argument `add=TRUE`. This can be thought as the equivalent of adding layers on top of the orginal layer. Note that the function `points` can also be used to add SpatialPoints objects or `lines` to add `SpatialLines`.

```
layout(matrix(1:2, nrow = 1))
plot(wC)
plot(kntm, add = TRUE, col = "blue")
```

```
# An alternative, the default points are different
plot(wC)
points(kntm, col = "blue")
```



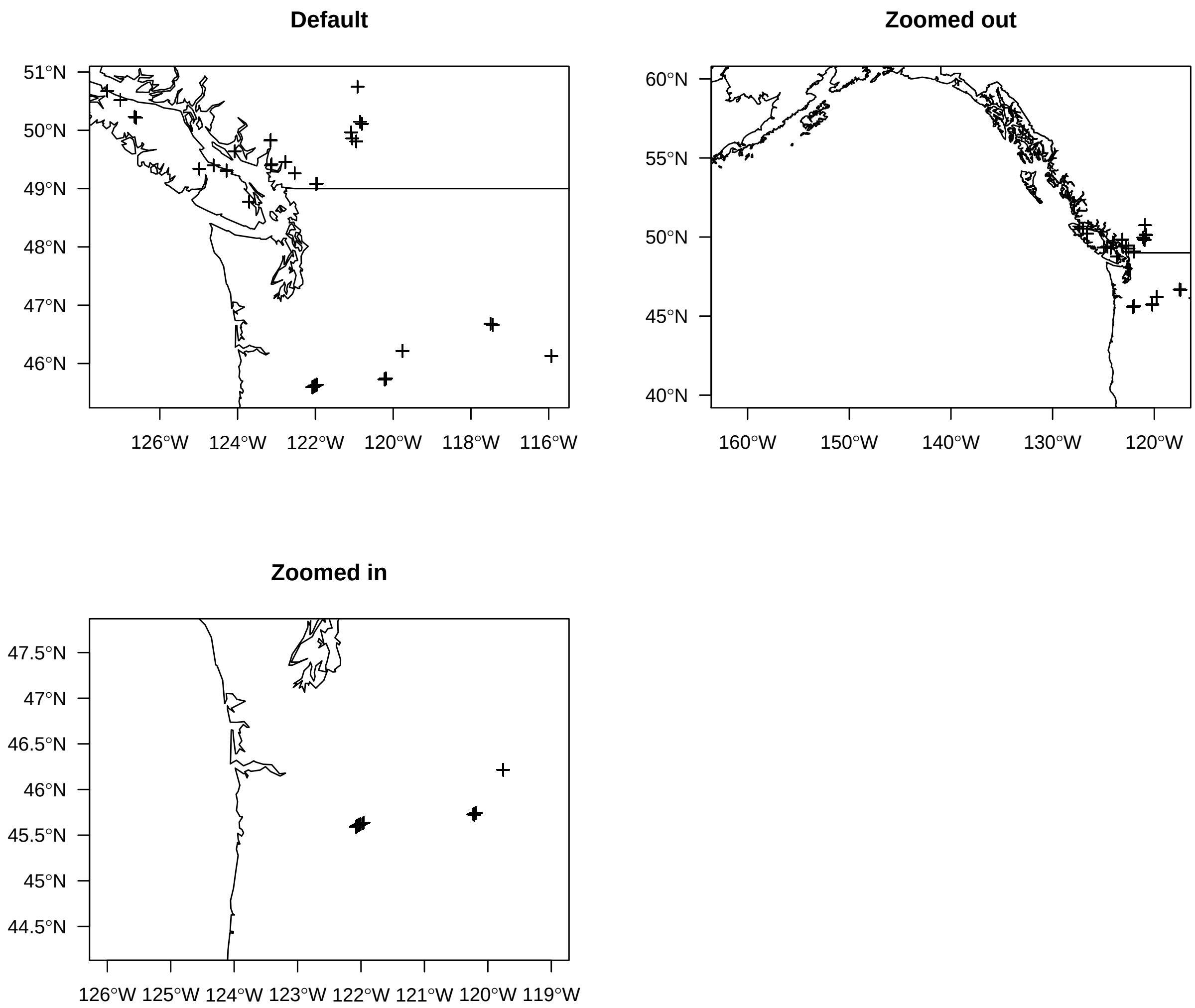


By default, the final plot will have the geographical extent of the first layer plotted. More precisely, the default geographical area that is plotted in a plot of a `Spatial*` object is the geographical extent of data extended on each side by a margin of a minimum of 4% the extent.

This said, you can partly control how much you are zoomed in or out by using the argument `xlim` and `ylim`.

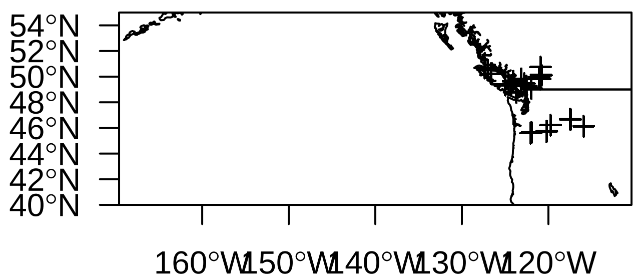
```
layout(matrix(1:4, nrow = 2))
# Original - default
plot(kntm, axes = TRUE, las = 1)
plot(wC, add = TRUE)
title("Default")
# Zoom in
plot(kntm, axes = TRUE, xlim = c(-126, -119), ylim = c(45,
```

```
    47), las = 1)
plot(wC, add = TRUE)
title("Zoomed in")
# Zoom out
plot(kntm, axes = TRUE, xlim = c(-160, -120), ylim = c(40,
    60), las = 1)
plot(wC, add = TRUE)
title("Zoomed out")
```



As you might noticed this is not perfect and that's because the geographical margin are partly fixed by the plot size. You can change the plot size using the `par` function. This is still not a perfect solution, but it brings you closer to the xlim and ylim.

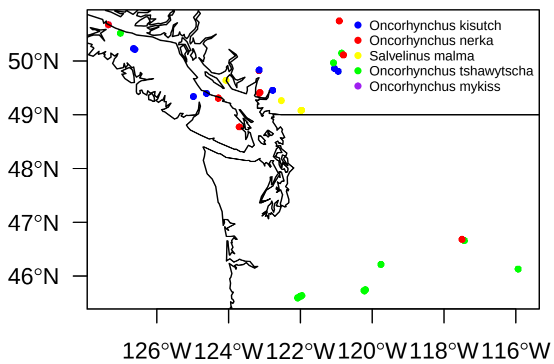
```
# To set own zoom Size of plot in inches
pin <- c(1, 1)
# The x-axis limits
xx <- c(-160, -120)
# The y-axis limits
yy <- c(40, 55)
# ratio between x and y axes
ratio <- diff(xx)/diff(yy)
# Set plot size based in ratio, xaxs='i' means to not
# add the 4% margin
par(pin = c(ratio * pin[2], pin[2]), xaxs = "i", yaxs = "i")
# Plot the SpatialPoints
plot(kntm, axes = TRUE, xlim = xx, ylim = yy, las = 1)
plot(wC, add = TRUE)
```

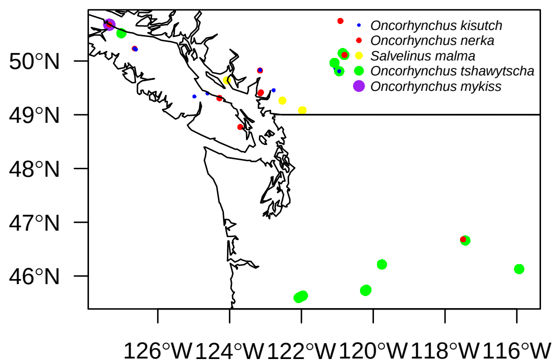


Often what is of interest is to plot the attributes, for example the salmon species of the Kintama projet.

```
layout(matrix(1:2, nrow = 1))
# Look at salmon species
salmonSp <- unique(kntm$scientificname)
# These are of class factor which is great, as the
# numbers will be related to the name
class(salmonSp)

## [1] "factor"

# how many species
length(salmonSp)

## [1] 5

# So choose 5 colours
salCol <- c("blue", "red", "yellow", "green", "purple")
```

```r
# Plot with a color for each species
plot(kntm, axes = TRUE, las = 1, pch = 19, col = salCol[kntm$scientificname],
    cex = 0.5)
# we don't see O. mykiss because it's under other
# dots
plot(wC, add = TRUE)
# Add lengend
legend("topright", legend = salmonSp, col = salCol, pch = 19,
    bty = "n", cex = 0.55)
# Change size and color
salSiz <- 1:5/5
plot(kntm, axes = TRUE, las = 1, pch = 19, cex = salSiz[kntm$scientificname],
    col = salCol[kntm$scientificname])
plot(wC, add = TRUE)
# Add lengend, change text font
legend("topright", legend = salmonSp, col = salCol, pch = 19,
    bty = "n", cex = 0.55, pt.cex = salSiz, text.font = 3)
```





While using the traditional `plot` function can help you do nice plots, it's often requires tedious choices for arguments. If you want to plot `Spatial*` object with attributes quickly, you can use the function `spplot` and trellis plots. If we have time more on this next class.