GIS in R: Tutorial 4

Marie Auger-Méthé

1 Manipulating layers

Now that we have a good understanding of `SpatialPoints`, `SpatialLines`, and `SpatialPolygons`, we can start extracting information from these objects and perform vector analyses. Often we are interested in manipulating multiple `Spatial` objects at the same time and `R` has many tools to help handling `Spatial` objects. We can think of each `Spatial` object in an analysis as a layer. Here, I'm going to present a few tools to manipulate multiples layers, most of which will be from the package `rgeos`. This package is an interface to Geometry Engine - Open Source (GEOS; see `http://trac.osgeo.org/geos/`). GEOS has a set of spatial functions, including functions to assess whether spatial objects intersect one another and functions that assess the distance between objects. In this tutorial, we will explore a few of the GEOS functions available through the `rgeos` package.

1.1 Setting up the layers

We are going to import four shapefiles. The four `Spatial` objects created when we import these shapefile are our original layers. Three of the layers are based on data downloaded from the Natural Earth website (`http://www.naturalearthdata.com/`). This is a great website with a set of good based layers. Here, I used two of the Natural Earth rivers datasets: 1) 10 m global rivers and lakes centerlines; and 2) 10m North American river supplement. In addition, I used the Natural Earth 10m states and province datasets. I modified these layers to clip them to the extent of our study area and I removed most all superfluous information from the layers. The resulting shapefiles are named: mainRivers, smallRivers, and borders. Our final layr is the salmon data from the Ocean Tracking Network (OTN) Kintama project data (more informationon the project website: `http://members.oceantrack.org/data/discovery/KNTM.htm`). The shapefile kntm 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`). This is based on the same dataset that we used in Tutorial 3.

```r
library(rgdal)
```

```
## Loading required package: sp
## 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)
```

```r
# OTN slamon point data
kntm <- readOGR(dsn = ".", layer = "kntm")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "kntm"
## with 3677 features and 6 fields
## Feature type: wkbPoint with 2 dimensions
```

```r
# Main rivers
mainRiv <- readOGR(dsn = ".", layer = "mainRivers")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "mainRivers"
## with 20 features and 1 fields
## Feature type: wkbLineString with 2 dimensions
```

```r
# Smaller rivers
smallRiv <- readOGR(dsn = ".", layer = "smallRivers")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "smallRivers"
## with 253 features and 1 fields
## Feature type: wkbLineString with 2 dimensions
```

```r
# Administrative boundaries and coast lines
bounds <- readOGR(dsn = ".", layer = "borders")
```

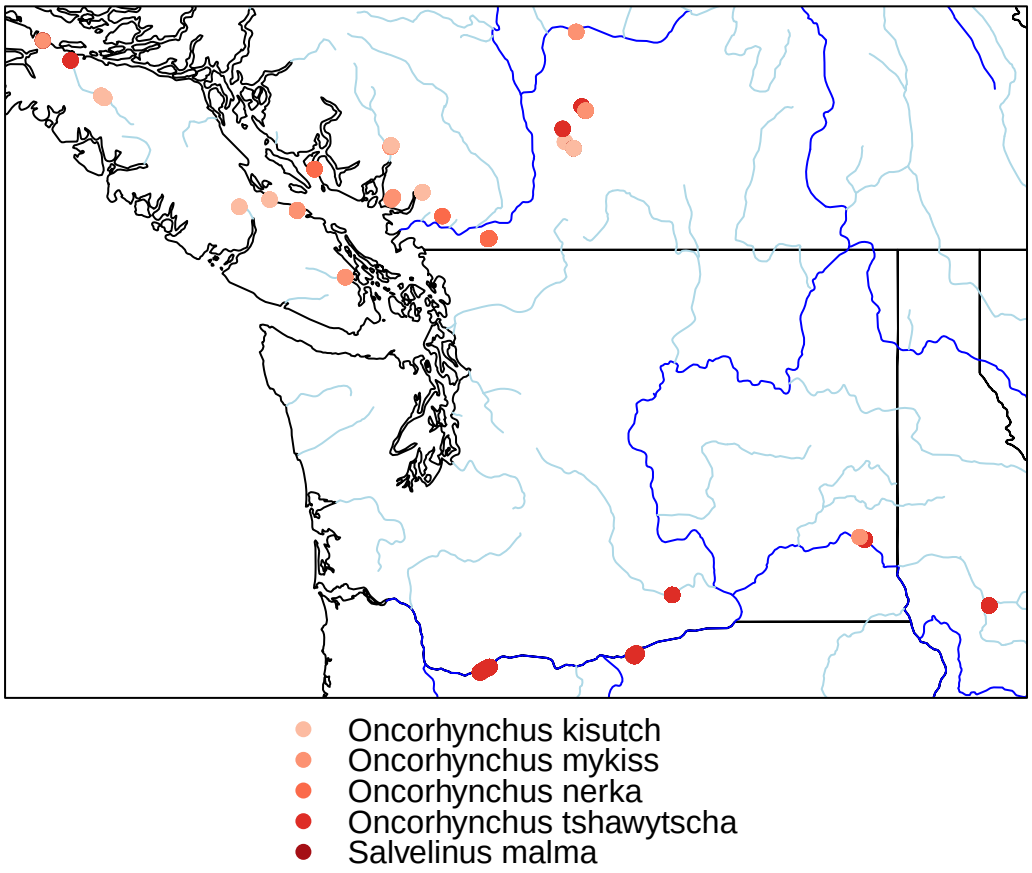
```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "borders"
## with 6 features and 2 fields
## Feature type: wkbPolygon with 2 dimensions
```

```
# Take a quick look at what attributes these objects
# contain
names(kntm)

## [1] "scntfcn" "dtcllct" "timefdy" "stock"   "lif_stg" "length"

names(mainRiv)

## [1] "name"

names(smallRiv)

## [1] "name"

names(bounds)

## [1] "name"  "admin"
```

When we do vector manipulation it is essential that the layers are in the same coordinate reference system (CRS). So the first thing we will do is check the CRS with the function `proj4sting` and by plotting the layers on top of one another to get a qualitative sense of whether the object by where they should be.

```
# What is the CRS of our main data
proj4string(kntm)

## [1] "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"

# Are the other layers in the same CRS?
identical(proj4string(kntm), proj4string(mainRiv))

## [1] TRUE

identical(proj4string(kntm), proj4string(smallRiv))

## [1] TRUE

identical(proj4string(kntm), proj4string(bounds))

## [1] TRUE

# Ok, we are good, they are all the same
# Let's take a quick look
```

```
library(RColorBrewer)  # For color palette (brewer.pal)
spplot(kntm, zcol = "scntfcn", col.regions = brewer.pal(6,
    "Reds")[-1], sp.layout = list(list("sp.polygons",
    bounds), list("sp.lines", mainRiv, col = "blue"),
    list("sp.lines", smallRiv, col = "lightblue")))
```



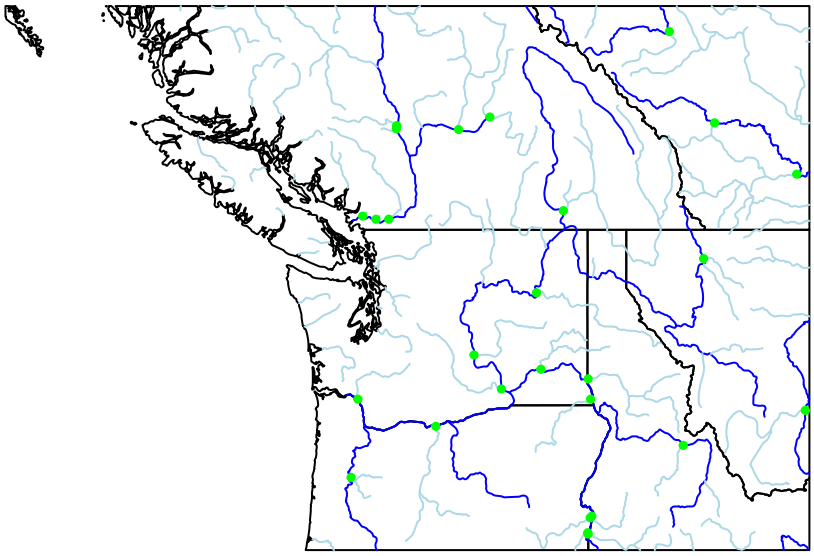
We can see in the plots that rivers are on land and many of them end up on the coast line, which is a sign that things match up. In addition, most salmon points are on rivers, which is also a sign that the CRS are consistents. Note that some salmon points are not on rivers, this could be because these rivers are too small to be incorporated in the Natural Earth datasets. Something that would need to be explored further if this was a real analysis.

1.2 Creating new layers

In our case we are not interested in dividing the rivers into big and small rivers. The separate layers were only made because the river data from Natural Earth came into two different shapefiles. Thus, the first thing we might like to do his to lump the rivers into one layer.

Before we create one layer with all rivers we want to verify whether rivers intersect and for this we can use the `rgeos` package tool `gInstersection`.

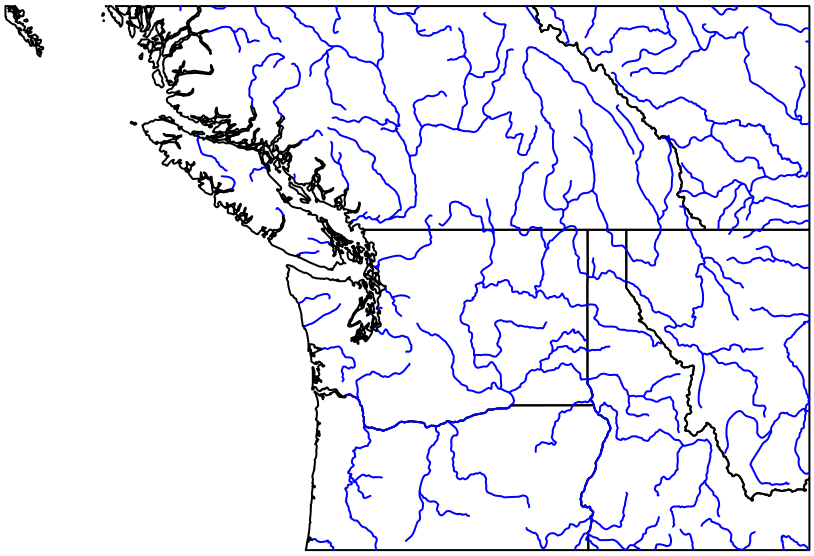
```
library(rgeos)

## rgeos version:  0.3-6, (SVN revision 450)
##  GEOS runtime version:  3.4.2-CAPI-1.8.2 r3921
##  Polygon checking:  TRUE

# Find area where mainRIv intersects with smallRiv
rivInt <- gIntersection(mainRiv, smallRiv)
plot(bounds)
plot(mainRiv, add = TRUE, col = "blue")
plot(smallRiv, add = TRUE, col = "lightblue")
plot(rivInt, add = TRUE, col = "green", pch = 19, cex = 0.5)
```



We can see that the rivers are intersecting at many points, although not at all points were the main rivers appear to cross smaller rivers. In cases were they look like the rivers crossed but no intersection was found could be the results of lines being close to one another but not exactly on top of each other.

Here we would like to create one layer with all rivers but where intersecting lines become one. We can use the `rgeos` `gUnion` function, which will join intersecting geometries.

```
rivAll <- gUnion(mainRiv, smallRiv)
# We now have all the rivers in one layer
plot(bounds)
plot(rivAll, add = TRUE, col = "blue")
```



```r
# What's the object created?
class(rivAll)

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

# Compared to the original datasets
class(mainRiv)

## [1] "SpatialLinesDataFrame"
## attr(,"package")
## [1] "sp"
```

```
class(smallRiv)
```

```
## [1] "SpatialLinesDataFrame"
## attr(,"package")
## [1] "sp"
```

```
# So we lost the sttributes associated with the
# mainRiv and smallRiv
# How many lines objects is there? (remember only
# lines object can have an ID and thus attributes)
length(rivAll)
```

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

```
# In contrast smallRiv and mainRiv had many more
# Lines objects
length(smallRiv)
```

```
## [1] 253
```

```
length(mainRiv)
```

```
## [1] 20
```

So `gUnion` creates one object with one `Lines` object. This may not always be optimal. However, in our case we are only interested in identifying areas with river water and `gUnion` is a perfect tool for this.

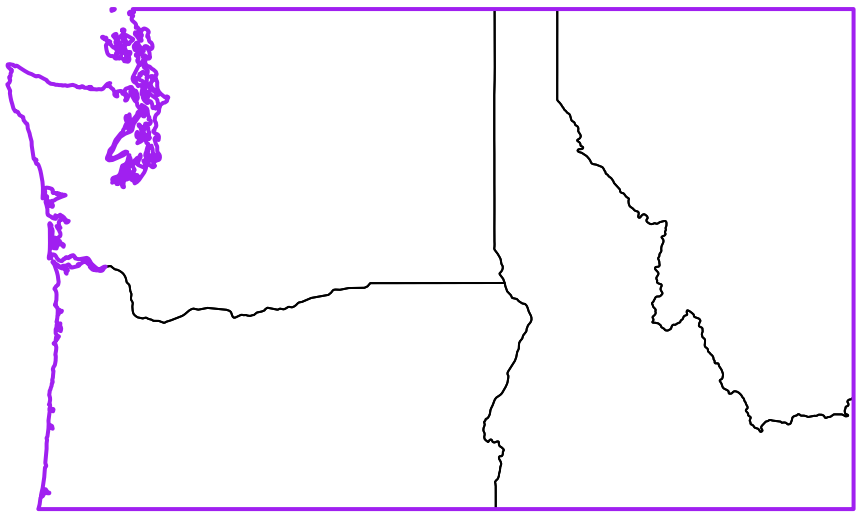
Now let's say we would like to focuss only on the salmons from United States. We would like to create a new layer that discard all inormation from the Canadian salmons. As a first step, we we another union tool `gUnaryUnion` from `rgeos` to create a new layer with a polygon that merge all American polygons into one.

```
# We want all polygons from the bounds layers that
# are of the admin 'United States of America'. For
# this we can simply subset the bounds layer using
# row index and the admin attributes.
usa <- bounds[bounds$admin == "United States of America",
    ]
# Nwe we only have the America polygons
plot(usa)
# We still have all the data associated with these
```

```r
# polygons
summary(usa)
```

```
## Object of class SpatialPolygonsDataFrame
## Coordinates:
##          min        max
## x -124.73461 -111.36826
## y   43.53241   48.99262
## Is projected: FALSE
## proj4string :
## [+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0]
## Data attributes:
##                name                          admin
##  Alberta         :0   Canada                   :0
##  British Columbia:0   United States of America:4
##  Idaho           :1
##  Montana         :1
##  Oregon          :1
##  Washington      :1
```

```r
# We want to create one big polygon that represent
# the borders of this multipolygon. For this we can
# use gUnaryUnion
usaB <- gUnaryUnion(usa)
# We can see now that we only have the surrounding
# border
plot(usaB, add = TRUE, border = "purple", lwd = 2)
```

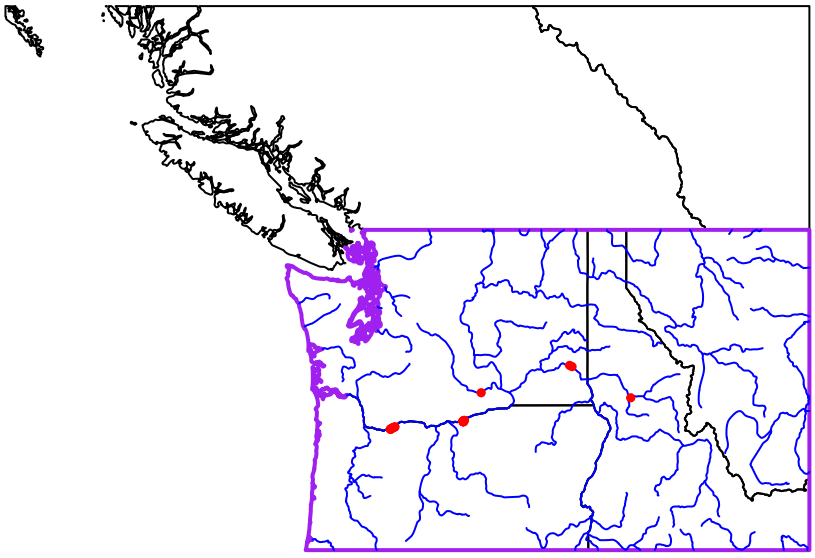


```
# But what is the new object?
class(usaB)

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

length(usaB)

## [1] 1

# Just one SpatialPolygons, with no data
```

We can then use this new polygon (usaB) to clip the layers to be only of the extent of

this new polygon using `gIntersection`.

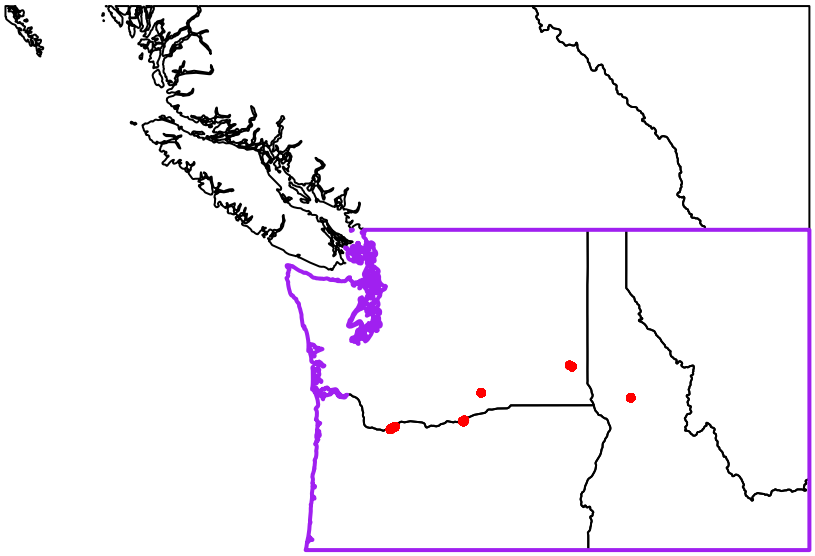
```
# Get the rivers in the usa
usaRiv <- gIntersection(rivAll, usaB)
# Get the salmons in the usa
usaKntm <- gIntersection(kntm, usaB)
# let's plot these new layers
plot(bounds)
plot(usaB, add = TRUE, border = "purple", lwd = 2)
plot(usaRiv, col = "blue", add = TRUE)
plot(usaKntm, col = "red", add = TRUE, pch = 19, cex = 0.5)
```



```
# What's in new layers
class(usaRiv)

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

class(usaKntm)

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

# They have no data
```

So `gIntersection` can clip the data to it's appropriate size but the returned object don't have data anymore, i.e., they are not `Spatial` objects with data frames. This is not surprising for the rivers since the rivAll object was already simply a `SpatialLines`, but waht if we wanted to keep the salmon attribute data? One option is to use spatial indexing. This is particularly useful for `SpatialPointsDataFrame`.

```
# We can subset Spatial objects with another Spatial
# object.  For example you can take only the data
# points that are in the usaB
usaKntm2 <- kntm[usaB, ]
# Just like before this is going to have only the
# American points
plot(bounds)
plot(usaKntm2, add = TRUE, col = "red", pch = 19, cex = 0.5)
plot(usaB, add = TRUE, border = "purple", lwd = 2)
```

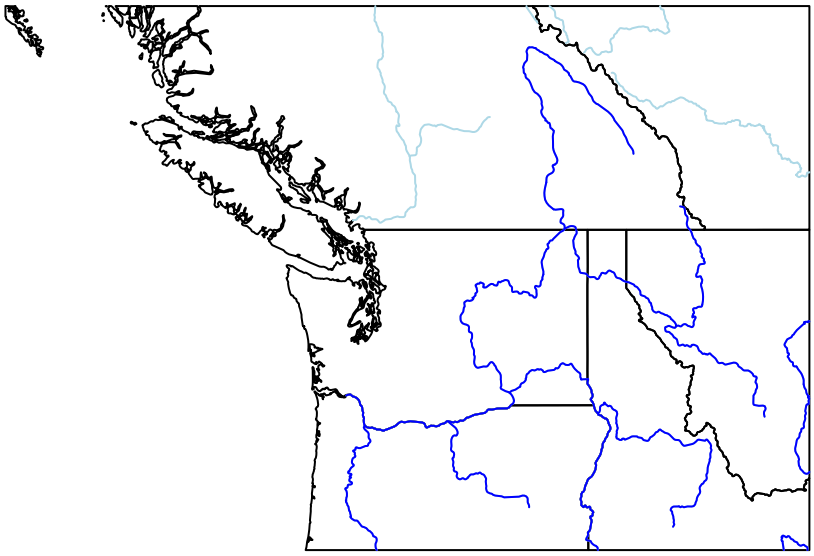


```
# However, unlike the usaKntm object made above this
# will have data
class(usaKntm)

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

class(usaKntm2)

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

```r
# It keeps all the columns from the kntm that we are
# subsetting
identical(names(usaKntm2), names(kntm))

## [1] TRUE
```

While we can also do a spatial subseting with polygons and lines, it will not clip the `Lines` and `Polygons` to the border. What it does is take all lines that goes inside the usaB but also keeps the parts that are outside. Here I'm using the mainRiv to give you a good examples.

```r
usaRiv2 <- mainRiv[usaB, ]
plot(bounds)
# Original mainRiv
plot(mainRiv, add = TRUE, col = "lightblue")
# Subsetted usaRiv2
plot(usaRiv2, add = TRUE, col = "blue")
```



```
# But it keeps the data
class(usaRiv2)

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

identical(names(usaRiv2), names(mainRiv))

## [1] TRUE
```

So you can see that all rivers that never enters the usaB polygons are excluded. Any rivers that enters usaB is kept, including the sections of these rivers that are outside usaB. The advantage here is that all of the attibutes associated with the rivers that intersects with

usaB are kept.

I often want to summarise the data from one layer based on whether they are in the features of another layer. For example, we might want to know how many species of salmon have been captured in each of the regions of our study area. To do this, we can use the function `aggregate`. The function `aggregate` will split the data into subset and apply a function to this subset. You can use simple function such as mean directly or you can create a small function that you would like to apply to this subset. In our case, we want to count the number of different species in each `Polygons` found in the bounds object.

In our case, we will first create a function that will get the unique values of the subset, here the unique scientific names found in the subset, and count how many unique values this subset had by getting the length of the vectored return by the function `unique`.

```
# Get the number of unique value in a dataset
nSp <- function(x) {
    length(unique(x))
}
# Just for fun let's apply this function to a simple
# vector
simEg <- c(1, 1, 1, 4, 6, 4)
simEg

## [1] 1 1 1 4 6 4

# So although there are 6 elements to this vector ,
# it only has 3 unique values: 1,4,6.
# Let's see if our function returns 3
nSp(simEg)

## [1] 3
```

Now that we have the function we want to apply to the subsets, we can use aggregate. The kntm object will be subsetted by the `Polygons` from bounds with which they intersect.

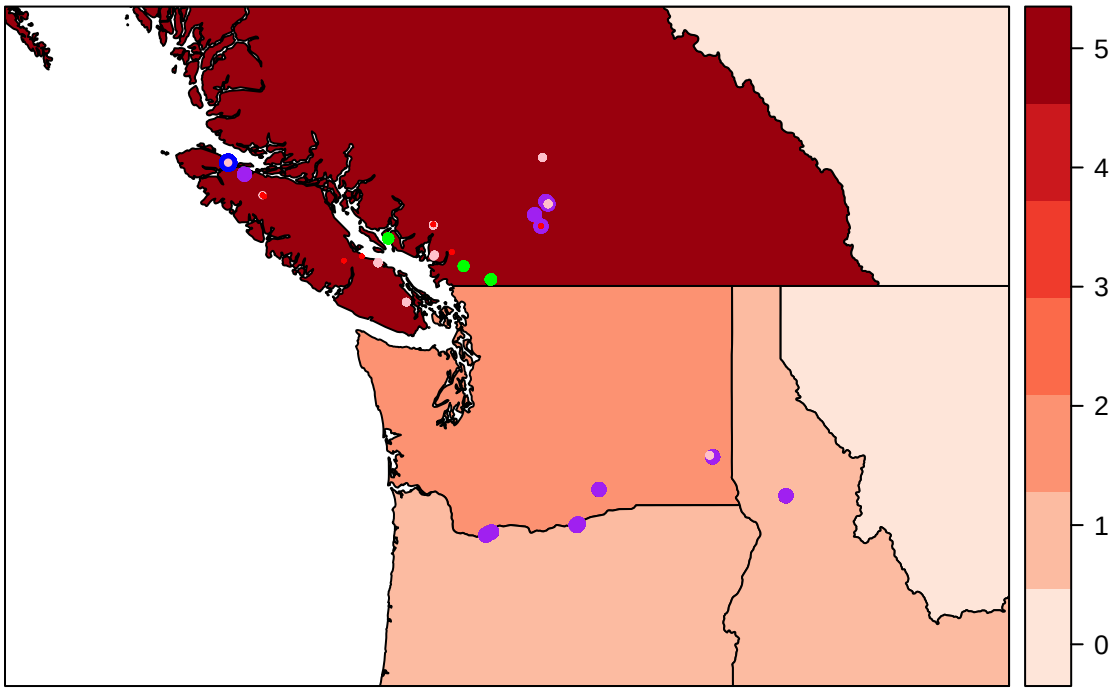
```
# Use aggregate to apply the function nSp to each
# subset
regionsSp <- aggregate(kntm[c("scntfcn")], by = bounds,
    FUN = nSp)
summary(regionsSp)

## Object of class SpatialPolygonsDataFrame
```

```
## Coordinates:
##          min        max
## x -131.91734 -111.36826
## y   43.53241   52.80656
## Is projected: FALSE
## proj4string :
## [+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0]
## Data attributes:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##    1.00    1.00    1.50    2.25    2.75    5.00       2
```

```
# Note that we get NA instead of 0 We can replace
# these values using is.na function
regionsSp$scntfcn[is.na(regionsSp$scntfcn)] <- 0
names(regionsSp)
```

```
## [1] "scntfcn"
```

```
# Setting colors for salmon species
salCol <- c("red", "pink", "green", "purple", "blue")
salSiz <- 1:5/5
spplot(regionsSp, zcol = "scntfcn", col.regions = brewer.pal(7,
    "Reds"), cuts = 6, sp.layout = list("sp.points",
    kntm, pch = 19, col = salCol[kntm$scntfcn], cex = salSiz[kntm$scntfcn]))
```



We can see here that our British Columbia dataset has the most salmon diversity. You can confirm that our code worked by counting the number of different coloured point in each regions and see if the number correspond to the color gradient of the polygon of the region.

`aggregate` is a very useful function and here we are going to use it to calculate the number of detections taken at each receivers. The first step we will do here is to make a new layer that has the location of each receivers. For this we will use the function `unique` on the coordinates of the kntm object.

```
# Get the coordinates of the receivers using the
# function unique on the coordinates of kntm
recLoc <- data.frame(unique(coordinates(kntm)))
# Make a SpatialPoints object using these coordinates
coordinates(recLoc) <- ~coords.x1 + coords.x2
```

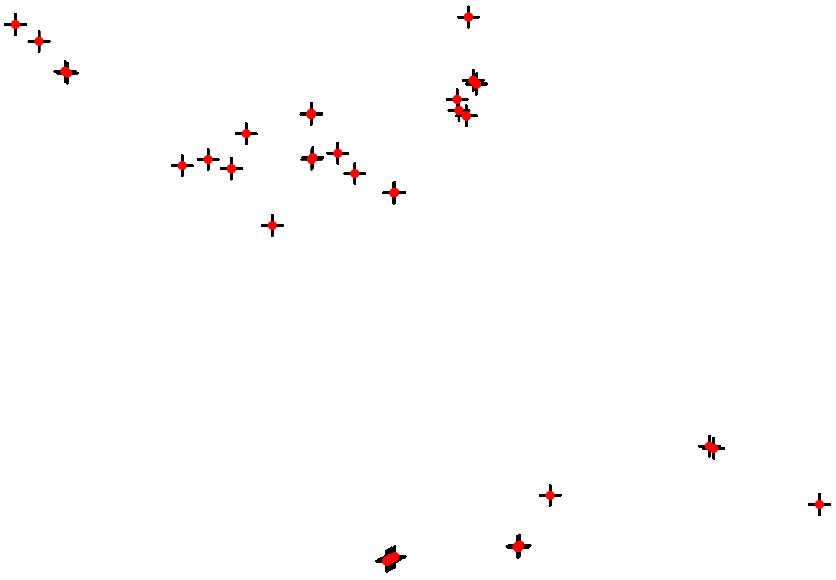
```r
proj4string(recLoc) <- CRS(proj4string(kntm))
# We can see that there are only a few receiver
# locations compared to the number of detections
length(recLoc)
```

```
## [1] 95
```

```r
length(kntm)
```

```
## [1] 3677
```

```r
# Let's plot the recLoc on top of the kntm to make
# that we have all of the receivers and that they are
# in the good locations
plot(kntm)
plot(recLoc, add = TRUE, pch = 19, col = "red", cex = 0.5)
```



Now that have the receiver locations, we can use aggregate to count the number of detections at each receiver. Here our new function is simply getting the length of the subset and thus the number of detections.

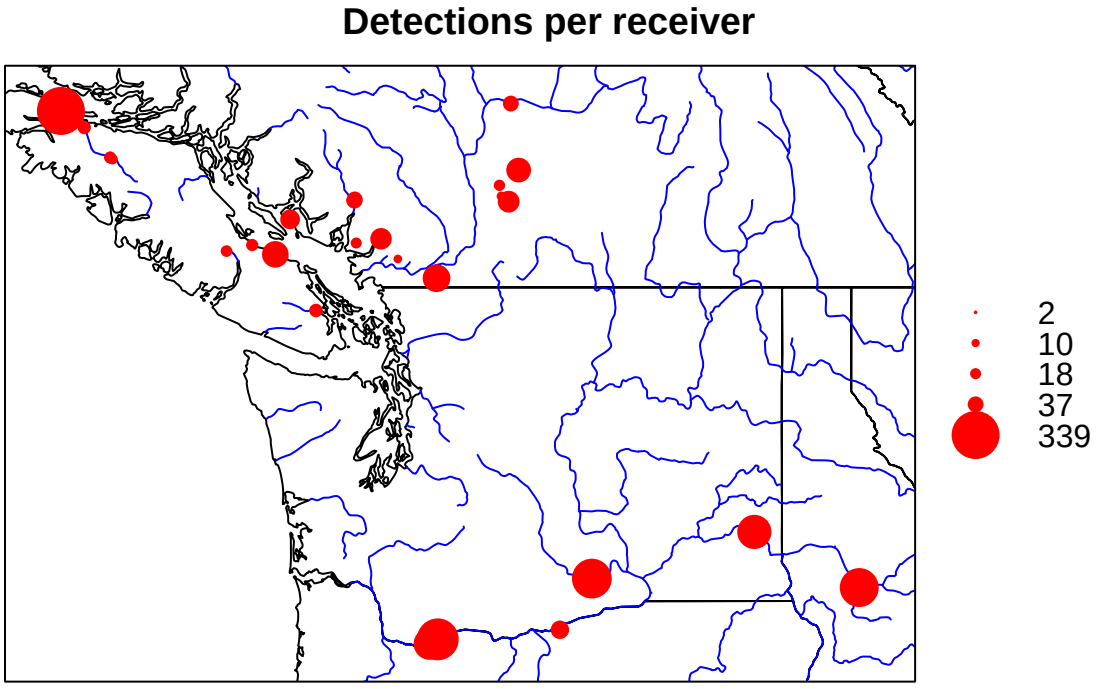
```
# Function to get the number of detections in the
# subset
nDe <- function(x) {
    length(x)
}
# Get the number of detections per receiver locations
decPerRec <- aggregate(kntm[c("scntfcn")], by = recLoc,
    FUN = nDe)
summary(decPerRec)
```

```
## Object of class SpatialPointsDataFrame
## Coordinates:
##                  min        max
## coords.x1 -127.35088 -115.93472
## coords.x2   45.59333   50.74564
## Is projected: FALSE
## proj4string :
## [+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0]
## Number of points: 95
## Data attributes:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    2.00   10.00   18.00   38.71   37.00  339.00
```

```
names(decPerRec)
```

```
## [1] "scntfcn"
```

```
# Let's plot it with bubble
bubble(decPerRec, zcol = "scntfcn", col = "red", sp.layout = list(list("sp.polygons",
    bounds), list("sp.lines", rivAll, col = "blue")),
    main = "Detections per receiver")
```



1.3 Measuring object features

In general, GEOS only handle planar geometries, and it's better to use projected CRS. This is especially true when one wants to look at area and distance. In fact, if you try to get the area using `gArea` on data with a geographical CRS, you'll get a warning,

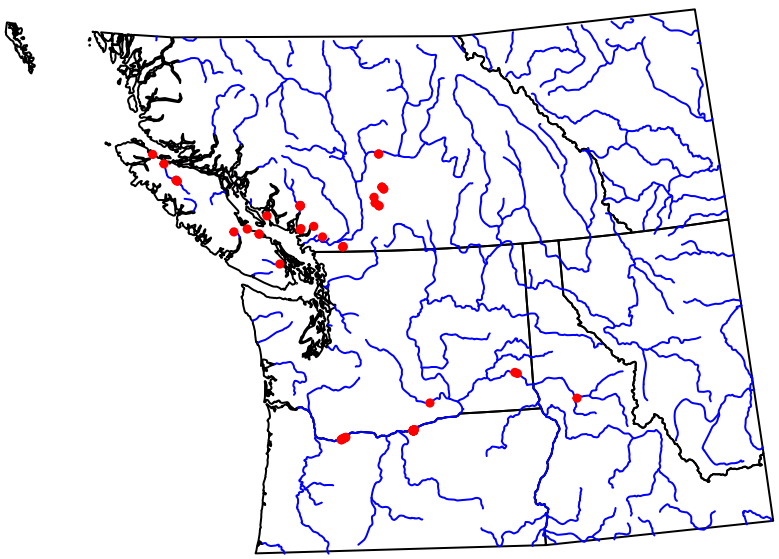
```
gArea(bounds)
```

```
## Warning in RGEOSMiscFunc(spgeom, byid, "rgeos_area"):  Spatial object is not
projected; GEOS expects planar coordinates
```

```
## [1] 129.5048
```

So let's transfrom our lwayers to a projected CRS. Let's use the Universal Tranverse Mercator (UTM) zone 10. Zone 10 covers most of our kntm data. Using UTM is not perfect in this case, since some points are in zone 11, but it should do the trick for now.

```
# Use spTransform to creat new layers with UTM
# projection
boundsUTM <- spTransform(bounds, CRS("+proj=utm +zone=10 +datum=WGS84"))
decUTM <- spTransform(decPerRec, CRS("+proj=utm +zone=10 +datum=WGS84"))
riversUTM <- spTransform(rivAll, CRS("+proj=utm +zone=10 +datum=WGS84"))
# Lets' look at it
plot(boundsUTM)
plot(riversUTM, add = TRUE, col = "blue")
plot(decUTM, pch = 19, cex = 0.5, col = "red", add = TRUE)
```



Now we should be able to get the area with `gArea`.

```
# This will give us the overall area of our
# SpatialPolygonsDataFrame object
gArea(boundsUTM)
```

```
## [1] 1.069774e+12
```

```
# If we want the area of each polygons object we can
# use
gArea(boundsUTM, byid = TRUE)
```

```
##            0            1            2            3            4
## 136999135734 349059512106 132764787897 130000524013 174312017741
##            5
## 146638463570
```

```
# These are in the units of the projection, so in our
# case m^2. SO if you want it in km^2 divide by 1 000
# 000 We can add these in a new column of the
# boundsUTM
boundsUTM$size <- gArea(boundsUTM, byid = TRUE)/1e+06
# The only provincs/state we have complete is
# Washington, the others are clipped at the edge, but
# we can use Washington to check wehther the area
# estimate is correct
boundsUTM$size[boundsUTM$name == "Washington"]
```

```
## [1] 174312
```

```
# According to wikiedia it's 184,827 km2 So we area a
# bit off, not too sure why
```

One thing that is often useful is to be able to put a buffer around a feature of interest. For example, here we would like to estimate how much river water there is close to each receiver locations. For this we can use `gBuffer`. This is a completely artificial example. I made the buffer really big just so we can visualise them easily.

```
# Let's put a 10 km buffer around the rivers
riv10k <- gBuffer(riversUTM, width = 10000)
# Let's put a 25 km buffer arround the receivers. In
```

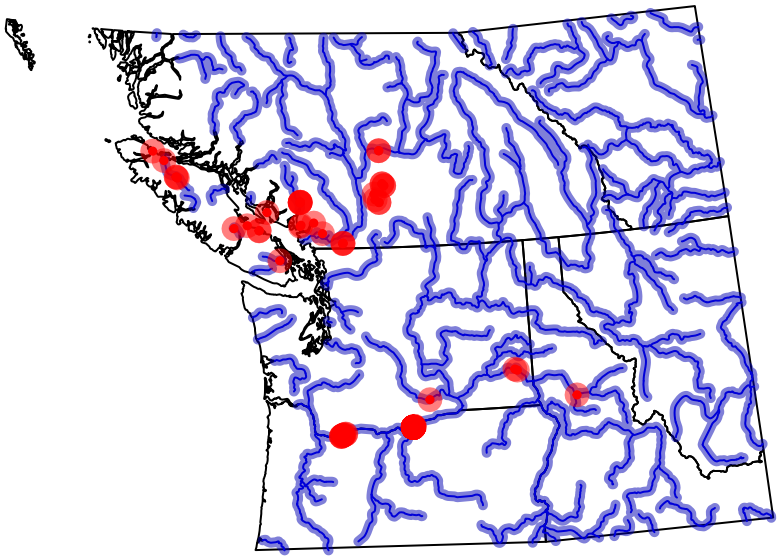
```r
# this case we want to keep each points indepdent and
# keep the data and so we use byid=TRUE
dec25k <- gBuffer(decUTM, width = 25000, byid = TRUE)
# We now have a SpatialPolygons and
# SpatialPolygonsDataFrame
class(riv10k)
```

```
## [1] "SpatialPolygons"
## attr(,"package")
## [1] "sp"
```

```r
summary(dec25k)
```

```
## Object of class SpatialPolygonsDataFrame
## Coordinates:
##         min     max
## x  167631.4 1070745
## y 5024278.7 5648609
## Is projected: TRUE
## proj4string :
## [+proj=utm +zone=10 +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0]
## Data attributes:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    2.00   10.00   18.00   38.71   37.00  339.00
```

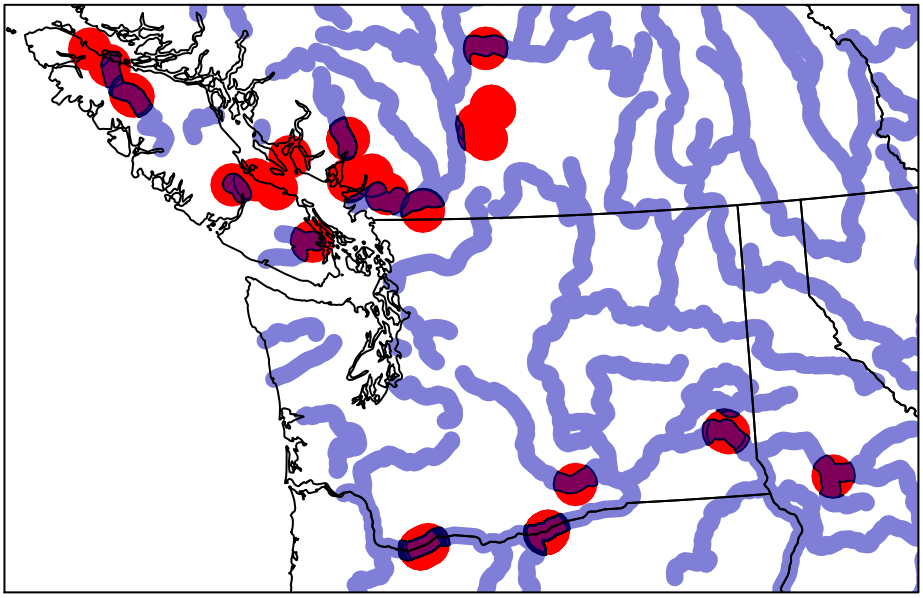
```r
plot(boundsUTM)
plot(riversUTM, col = "blue", add = TRUE)
plot(riv10k, col = rgb(0, 0, 0.7, 0.5), border = NA,
    add = TRUE)
plot(decUTM, col = "red", add = TRUE, pch = 19, cex = 0.5)
plot(dec25k, col = rgb(1, 0, 0, 0.5), border = NA, add = TRUE)
```



Now we would like to see how much water edge (represented by the 10k buffer) is in the vacinity of each receiver (represented by the 25k buffer). Here, again we will make use of `gIntersection` again.

```
rivInRec <- gIntersection(riv10k, dec25k, byid = TRUE)
# Now we will have new polygons for each
# intersection, but that means that if some dec25 did
# not intersect with the river buffer we will have
# fewer number of polygons
length(rivInRec)

## [1] 86

length(dec25k)
```

```
## [1] 95

# This is the case
plot(dec25k, col = "red", border = NA)
plot(rivInRec, add = TRUE)
plot(riv10k, col = rgb(0, 0, 0.7, 0.5), border = NA,
    add = TRUE)
plot(boundsUTM, add = TRUE)
box()
```



So now we have a set of polygons that represent the area where the riv10k intersected with dec25k. As we can see above, the new `SpatialPolygons` has less `Polygons` than the dec25k buffer polygons and that's because `gIntersection` only creates a new polygon for area where riv10k and dec25k intersects and not for the dec25k buffer polygons that don't

intersect wth riv10k.

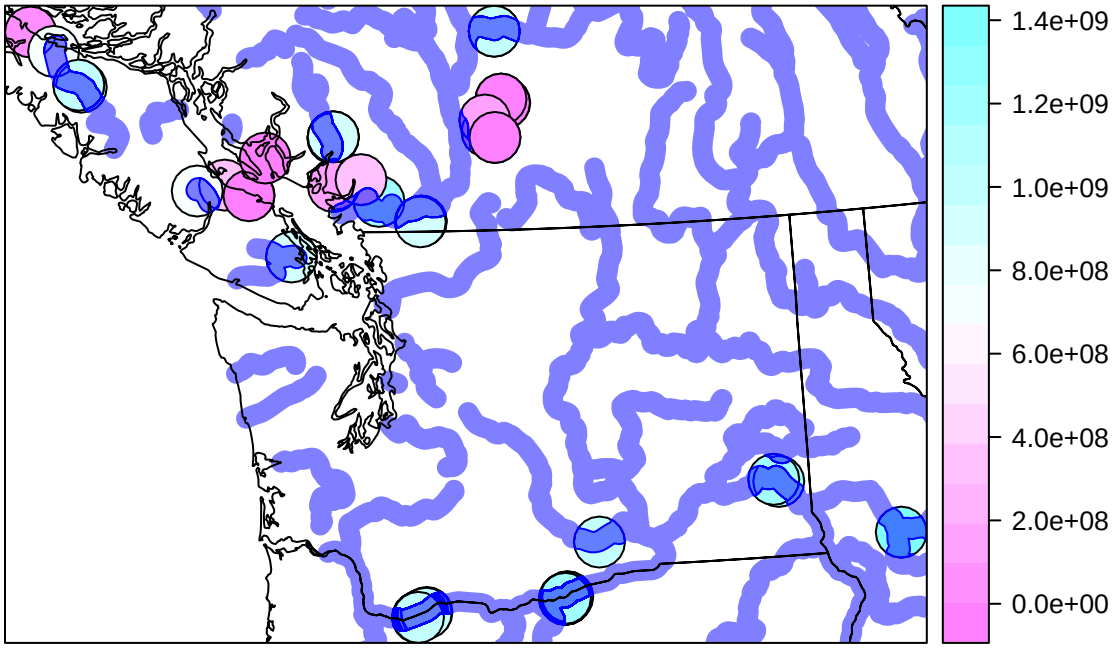
In our case we would like to associated each dec25k buffer `Polygons` object to the area of water river in them. For this, we first need to find the appropriate row names to link the rivInRec object that resulted from the `gInstersection` of riv10k with dec25k. The row name of an object created with `gIntersection` will be a combination of the name of the two objects is intersects. We can use the function `strsplit` to divide the row name into its two components.

```
# First let's look at the name of the riv10k, dec25k,
# and rivInRec
head(row.names(riv10k))

## [1] "buffer"

head(row.names(dec25k))

## [1] "1" "2" "3" "4" "5" "6"

head(row.names(rivInRec))

## [1] "buffer 1" "buffer 2" "buffer 3" "buffer 4" "buffer 5" "buffer 6"

# You can see that rivInRec is a combination of the
# row names of riv10k and dec25k
# Now we want to use strsplit to divide the name of
# rivRec and get the name of dec25k
divName <- strsplit(row.names(rivInRec), " ")
# It gives a list wth each element having 2 elements
head(divName, 3)

## [[1]]
## [1] "buffer" "1"
##
## [[2]]
## [1] "buffer" "2"
##
## [[3]]
## [1] "buffer" "3"

length(divName[[1]])

## [1] 2
```

```
# We only want the second element, because that's the
# name associated with dec25k. We will the second
# element using sapply (which is really similar to
# lapply we learned in previous Tutorial)
divName <- sapply(divName, "[", 2)
# Ok we only have the names associated with dec25k
head(divName)

## [1] "1" "2" "3" "4" "5" "6"
```

Now we can use the function `gArea` on each `Polygons` object of the rivInRec layers and associated these with the appropriate dec25k `Polygons` using the names we have just found above.

```
# Now we can create a new column in dec25k that will
# contain the amount of river edge we have in the
# buffer of the receiver and put 0.
dec25k$H2O <- 0
# We can calculate the area of each Polygons in
# RivInRec using gArea(byid=TRUE). We use the name we
# created above to associated the rivInRec Polygons
# to the of dec25k
dec25k$H2O[match(divName, row.names(dec25k))] <- gArea(rivInRec,
    byid = TRUE)
# Looks qualitatively ok
spplot(dec25k, zcol = "H2O", sp.layout = list(list("sp.polygons",
    rivInRec, col = "blue"), list("sp.polygons", riv10k,
    fill = rgb(0, 0, 1, 0.5), col = NA), list("sp.polygons",
    boundsUTM)))
```



Exercise 1

The package `rgeos` has a multitude of tools for spatial analyses. In this tutorial I have only presented a small subsets of the tools tha are available to you. For this exercise, I want you to explore the function `gDistance`.

1. Get the information on this function by typing `?gDistance` in your console.
2. Apply `gDistance` to the decUTM object. Write in a comment in the code what the results mean.
3. Apply `gDistance(byid=TRUE)` to the decUTM object. Write in a comment your interpretation of the results.
4. Transform the kntm layer into a new layer with an UTM zone 10 projection. Using this new layer calculate the minimum distance between detection of *Oncorhynchus nerka* and `Salvelinus malma`.