Spatial workshop 2: vector spatial data in R

Contents

Find lower-resolution global shapefiles in ohiprep/globalprep/spatial/downres. Filename contains info on what regions it covers (all, EEZ, Antarctica, FAO, land), resolution (low and medium), and CRS (lat-long GCS and Mollweide).

Look for med resolution, mol projection (for global equal area), and pull the .shp, .dbf, and .shx files. The medium resolution files are ~ 11 MB for global coverage, so not huge but also detailed enough for zooming in a bit.

Note that there is no .prj file - so no projection information will come along with these files. There are two sets of coordinate reference systems:

- mol: Mollweide: equal area projection, units in meters.
  - EPSG:54009
  - proj.4 string: `'+proj=moll +lon_0=0 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs'`
- gcs: lat and long in degrees
  - EPSG:4326 WGS 84
  - proj.4 string: `'+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0'`

Shapefiles and polygons in R

Reading shapefiles

Here are two different ways to read in spatial vector data, each with advantages and disadvantages:

```
library(rgdal)
```

```
## Loading required package: sp
## rgdal: version: 1.0-4, (SVN revision 548)
##  Geospatial Data Abstraction Library extensions to R successfully loaded
##  Loaded GDAL runtime: GDAL 1.11.2, released 2015/02/10
##  Path to GDAL shared files: /Library/Frameworks/R.framework/Versions/3.2/Resources/library/rgdal/gdal
##  Loaded PROJ.4 runtime: Rel. 4.9.1, 04 March 2015, [PJ_VERSION: 491]
##  Path to PROJ.4 shared files: /Library/Frameworks/R.framework/Versions/3.2/Resources/library/rgdal/pr
##  Linking to sp version: 1.1-1
```

```
dir_spatial   <- '~/github/spatial_analysis2_R/spatial_data'
layer_bc <- 'ohibc_rgn'

poly_bc_rgn <- readOGR(dsn = path.expand(dir_spatial), layer = layer_bc, stringsAsFactors = FALSE)
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "/Users/ohara/github/spatial_analysis2_R/spatial_data", layer: "ohibc_rgn"
## with 8 features
## It has 3 fields
```

```
  ### note the path.expand(): readOGR can be annoyingly finicky about file names;
  ### it is apparently not a fan of the '~'.  path.expand() gets rid of that.

### you can easily plot polygons, though complex polys take a while
plot(poly_bc_rgn, col = 'light blue', border = 'blue')
```

- `dsn` is the data source name. It can be a .gdb or a directory with shapefiles in it. It can be relative or absolute file path.
  - `dsn` is not a fan of `~` as a shortcut for home directory, e.g. `dsn = '~/github/ohiprep`.
  - To get around this, one option is to use `path.expand()`, e.g. `dsn = path.expand('~/github/ohiprep')`
- `layer` is the layer name.
  - If you're reading from a .gdb, you can tell it which layer to pull out of that .gdb
    - `ogrListLayers()` can help identify the layers within the .gdb if you're not sure.
  - If you're reading a .shp (and the associated .dbf etc), then `layer` should be the base name of the shapefile (without the .shp extension). E.g. if you want `rgn_all_mol_med_res.shp`, then `layer = 'rgn_all_mol_low_res'`.
- `p4s` allows you to input a proj.4 string to indicate a CRS. If there's already a .prj file associated with this layer, `readOGR()` will automatically read it in.

```
library(maptools)
```

```
## Checking rgeos availability: TRUE
```

```
shp_bc <- file.path(dir_spatial, layer_bc)
p4s_bc <- CRS('+proj=aea +lat_1=50 +lat_2=58.5 +lat_0=45 +lon_0=-126 +x_0=1000000 +y_0=0 +datum=NAD83 +
  ### that's the proj4string for BC Albers projection

poly_bc_rgn <- readShapePoly(fn = shp_bc, proj4string = p4s_bc)
```

- `fn` is the full filename, leaving off the .shp extension. This has to be a shapefile (not a .gdb).
- `proj4string` is an optional proj.4 CRS designation.
  - `readShapePoly()` does not read .prj files, so it will not know the projection unless you manually tell it.
  - This argument has to be a CRS object, so for example, if you want to set Mollweide you would use: `proj4string = CRS('+proj=moll +lon_0=0 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs')`

`readOGR()` and `readShapePoly()` (and its cousins `readShapeLines()`, `readShapePoints()`, etc) also have analogous `write` functions.

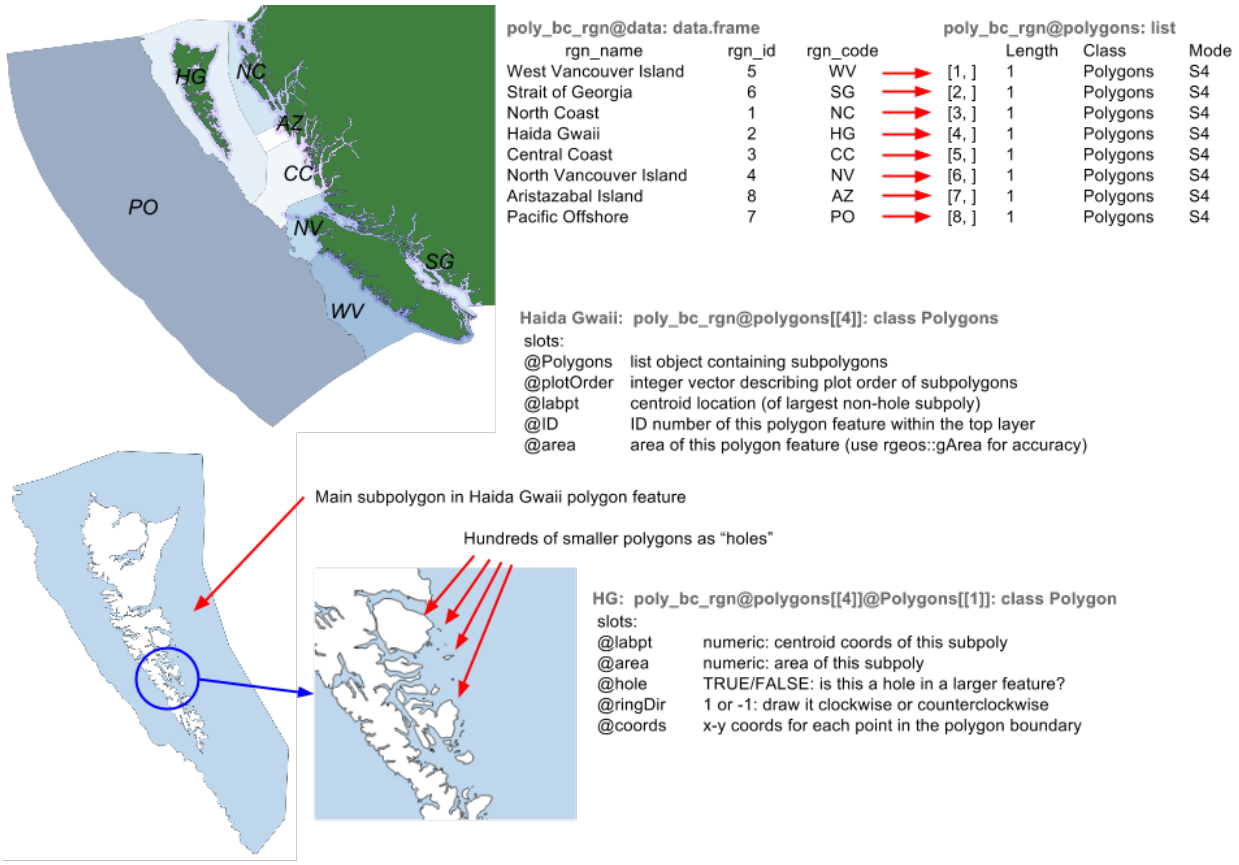
`readShapePoly()` seems to be significantly faster for big shapefiles, but doesn't work on .gdb and doesn't automatically read in CRS info, which `readOGR()` can do. I prefer to stick with `readOGR()` where possible (especially for smaller files) to use projection/CRS information where it is available.

One note, learned the annoying painful way, is that if you wish to read layers from a geodatabase (.gdb), you need to have GDAL version 1.11 or higher installed on your computer (outside of R). In that version, they included the driver for the OpenFileGDB file format. As of this writing, Neptune's GDAL version is only 1.10, though hopefully it will be updated soon, when the OS etc is updated.

Understanding Spatial Polygons in R

Once the shapefile is read in, it becomes a SpatialPolygonsDataFrame object (from the `sp` package). SpatialPolygonsDataFrame store shape info and attributes in different slots, with a structure like this:

- `SpatialPolygonsDataFrame` (top level structure)
  - `@data`: a dataframe that holds the attribute table; each row correlates with a polygon in the `polygons` slot.
  - `@polygons`: a list of Polygons-class objects, each with their own bits and pieces. Each item in the list is a complete polygon feature, made up of sub-polygons; and each item in the list corresponds with a row in the dataframe in the `data` slot.
  - `@plotOrder`: a vector of integers showing which order to plot the `polygons` list
  - `@bbox`: info on the bounding coordinates (x & y min & max).
  - `@proj4string`: a CRS object that contains projection or coordinate system info



Each of these slots can be accessed by a `@` symbol (rather than a `$` symbol). If your SpatialPolygonsDataFrame is called `x`, then you can access the attribute table by calling `x@data` and treat it exactly like a regular data frame (though be wary of operations that might change the order or length of the data frame - confusion and chaos will reign!)

```
### view the overall structure
summary(poly_bc_rgn)
```

```
## Object of class SpatialPolygonsDataFrame
## Coordinates:
##        min     max
## x 159640.2 1237659
## y 173874.5 1222687
## Is projected: TRUE
## proj4string :
## [+proj=aea +lat_1=50 +lat_2=58.5 +lat_0=45 +lon_0=-126 +x_0=1000000
## +y_0=0 +datum=NAD83 +units=m +no_defs +ellps=GRS80 +towgs84=0,0,0]
## Data attributes:
##                    rgn_name     rgn_id        rgn_code
##  Aristazabal Island     :1   Min.   :1.00   AZ     :1
##  Central Coast          :1   1st Qu.:2.75   CC     :1
##  Haida Gwaii            :1   Median :4.50   HG     :1
##  North Coast            :1   Mean   :4.50   NC     :1
##  North Vancouver Island:1   3rd Qu.:6.25   NV     :1
##  Pacific Offshore       :1   Max.   :8.00   PO     :1
##  (Other)                :2                  (Other):2
```

```
### check out the attribute table in the data slot
head(poly_bc_rgn@data)
```

```
##                 rgn_name rgn_id rgn_code
## 0  West Vancouver Island      5       WV
## 1      Strait of Georgia      6       SG
## 2            North Coast      1       NC
## 3            Haida Gwaii      2       HG
## 4          Central Coast      3       CC
## 5 North Vancouver Island      4       NV
```

```
### check out the basics of the polygon features
summary(poly_bc_rgn@polygons)  ### summary of the polygon features
```

```
##      Length Class    Mode
## [1,] 1      Polygons S4
## [2,] 1      Polygons S4
## [3,] 1      Polygons S4
## [4,] 1      Polygons S4
## [5,] 1      Polygons S4
## [6,] 1      Polygons S4
## [7,] 1      Polygons S4
## [8,] 1      Polygons S4
```

```
### inspect one of the smaller sub-polys that make up the first polygon
### feature; this one draws a small hole in the main polygon
poly_bc_rgn@polygons[[1]]@Polygons[[2]]
```

```
## An object of class "Polygon"
## Slot "labpt":
## [1] 958590.8 510484.6
##
## Slot "area":
## [1] 37382.12
##
## Slot "hole":
## [1] TRUE
##
## Slot "ringDir":
## [1] -1
##
## Slot "coords":
##           [,1]     [,2]
##  [1,] 958598.4 510391.4
##  [2,] 958680.8 510448.1
##  [3,] 958708.1 510505.0
##  [4,] 958693.1 510552.6
##  [5,] 958693.0 510599.7
##  [6,] 958665.0 510602.7
##  [7,] 958610.9 510624.9
##  [8,] 958604.0 510602.3
##  [9,] 958565.5 510567.8
```

```
## [10,] 958486.9 510454.9
## [11,] 958450.4 510422.4
## [12,] 958445.0 510410.6
## [13,] 958469.5 510373.5
## [14,] 958525.5 510371.3
## [15,] 958598.4 510391.4
```

```
### check out the CRS info
poly_bc_rgn@proj4string
```

```
## CRS arguments:
##  +proj=aea +lat_1=50 +lat_2=58.5 +lat_0=45 +lon_0=-126 +x_0=1000000
## +y_0=0 +datum=NAD83 +units=m +no_defs +ellps=GRS80 +towgs84=0,0,0
```

You can also manipulate these slots individually, for example filtering out rows of the dataframe in the `data` slot, but you will also need to treat the other slots accordingly.

Adding data to polygons

If you'd like to add a column of data to the SpatialPolygonsDataFrame, for example harvest tonnes or a region ID value, you should be able to simply access the `data` slot and treat it like a data frame.

```
### create a data frame of harvest per region.  Note not all regions are represented here!
harv_data <- data.frame(rgn_id   = c(  1,  2,  3,  5,  8),
                        h_tonnes = c(105, 89, 74, 21, 11))

### use left_join to join data to attributes table without changing order
poly_bc_rgn@data <- poly_bc_rgn@data %>%
  left_join(harv_data, by = 'rgn_id')

### note unassigned regions have NA for harvest tonnage
poly_bc_rgn@data
```

```
##                 rgn_name rgn_id rgn_code h_tonnes
## 1  West Vancouver Island      5       WV       21
## 2      Strait of Georgia      6       SG       NA
## 3            North Coast      1       NC      105
## 4            Haida Gwaii      2       HG       89
## 5          Central Coast      3       CC       74
## 6 North Vancouver Island      4       NV       NA
## 7     Aristazabal Island      8       AZ       11
## 8       Pacific Offshore      7       PO       NA
```

Selecting or filtering polygons based on attributes

Use indexing to select just rows where there is harvest data. This eliminates the polygons as well as the attribute table rows, and resets the plot order. Probably safer than filtering on the data slot directly...

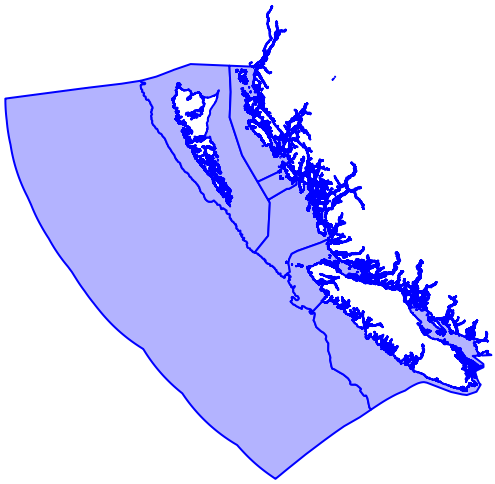
```
### Select only regions with non-NA harvest values
poly_bc_harvest <- poly_bc_rgn[!is.na(poly_bc_rgn@data$h_tonnes), ]

poly_bc_harvest@data
```

```
##                rgn_name rgn_id rgn_code h_tonnes
## 1 West Vancouver Island      5       WV       21
## 3           North Coast      1       NC      105
## 4           Haida Gwaii      2       HG       89
## 5         Central Coast      3       CC       74
## 7    Aristazabal Island      8       AZ       11
```

Fixing projections

Mismatched coordinate reference systems are just as much a pain in R as they are in ArcGIS or QGIS. But they're easy to work with once you know how to inspect them and re-project them.

```
poly_bc_mpa <- readShapePoly(file.path(dir_spatial, 'mpa_bc'))

plot(poly_bc_rgn, col = rgb(.7, .7, 1), border = rgb(0, 0, 1))
plot(poly_bc_mpa, col = rgb(1, .5, .5, .5), border = rgb(1, 0, 0, .5), add = TRUE)
```



```
  ### fun fact: the 'rgb()' function allows you to specify color by
  ### Red/Green/Blue proportions, but also allows a fourth argument
  ### for 'alpha' (opacity) to create semi-transparent layers

### Note that the MPA polys don't show up on the map!  That's because of a
### CRS error...
poly_bc_mpa@proj4string ### NA!  But it's actually in lat-long coord system.
```

```
## CRS arguments: NA
```

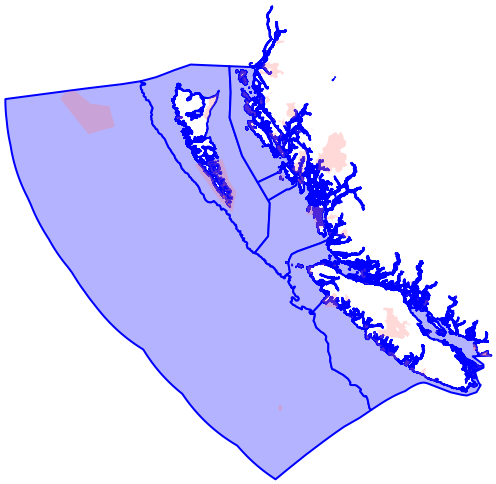
```
poly_bc_mpa@proj4string <- CRS('+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0')

### Now that it knows the correct CRS we can reproject to BC Albers:
poly_bc_mpa <- spTransform(poly_bc_mpa, p4s_bc)
poly_bc_mpa@proj4string
```

```
## CRS arguments:
##  +proj=aea +lat_1=50 +lat_2=58.5 +lat_0=45 +lon_0=-126 +x_0=1000000
## +y_0=0 +datum=NAD83 +units=m +no_defs +ellps=GRS80 +towgs84=0,0,0
```

```
poly_bc_rgn@proj4string  ### OK, now they match!
```

```
## CRS arguments:
##  +proj=aea +lat_1=50 +lat_2=58.5 +lat_0=45 +lon_0=-126 +x_0=1000000
## +y_0=0 +datum=NAD83 +units=m +no_defs +ellps=GRS80 +towgs84=0,0,0
```

```
plot(poly_bc_rgn, col = rgb(.7, .7, 1), border = rgb(0, 0, 1))
plot(poly_bc_mpa, col = rgb(1, .5, .5, .3), border = NA, add = TRUE)
```



```
  ### The semi-transparency not only looks way awesome, but also
  ### allows us to see whether there are any overlapping polygons!
```

Note that transforming might introduce geometry errors in a previously error-free Spatial object... for notes on checking and fixing this, see the appendix.

Rasters and polygons

Extracting raster data into regions defined by a SpatialPolygons* object is pretty easy, using the `extract()` function from the `raster` package. In next few code chunks we'll determine the extent of marine protected areas within each analysis region in BC's EEZ.

While it is possible to use vector geoprocessing to figure this out, overlapping polygons could be double-counted, so I will instead rasterize the MPA polygons, prioritizing the earliest date of protection. First step - establish a base raster.

Creating a raster from scratch

```
library(raster)
```

```
##
## Attaching package: 'raster'
##
## The following object is masked from 'package:tidyr':
```

```
##
##     extract
##
## The following object is masked from 'package:dplyr':
##
##     select
```

```
### use poly_bc_rgn to set raster extents and then round to nearest km
ext <- extent(poly_bc_rgn)
ext@xmin <- round(ext@xmin - 500, -3); ext@ymin <- round(ext@ymin - 500, -3)
ext@xmax <- round(ext@xmax + 500, -3); ext@ymax <- round(ext@ymax + 500, -3)

reso <- 1000 ### BC Albers uses meters as units, set a 1-km grid
xcol <- (ext@xmax - ext@xmin)/reso; yrow <- (ext@ymax - ext@ymin)/reso

rast_base <- raster(ext, yrow, xcol, crs = p4s_bc)

rast_base ### inspect it: resolution and extents are nice and clean
```

```
## class       : RasterLayer
## dimensions  : 1050, 1079, 1132950  (nrow, ncol, ncell)
## resolution  : 1000, 1000  (x, y)
## extent      : 159000, 1238000, 173000, 1223000  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=aea +lat_1=50 +lat_2=58.5 +lat_0=45 +lon_0=-126 +x_0=1000000 +y_0=0 +datum=NAD83
```

Rasterize polygons

Next step: rasterize the MPA polygons. See later in the doc for an alternative to raster::rasterize() that is way faster, avoids some odd problems, but has a fairly important limitation.

```
rast_bc_mpa <- rasterize(poly_bc_mpa, rast_base, field = 'STATUS_YR', fun = 'min')
```

The gloriousness of raster::extract()

Now the big step: raster::extract()

Note that `extract()` is also a function in the `tidyr` package and elsewhere; so I generally force the issue by calling `raster::extract()` just to make sure.

The basic arguments are simply the raster with the interesting data, and the polygon features that define the regions of interest. There are other arguments, but one set of arguments that may be interesting is `weights` and/or `normalizeWeights`.

- `weights` (logical): if `weights = TRUE`, provides info on how much of a cell was covered by the polygon in question.
- `normalizeWeights` (logical): if `normalizeWeights = TRUE` then the reported weights will be normalized for each polygon, so the total for each polygon adds up to 1.00. If you just want the fraction of cell coverage for each cell, set `normalizeWeights = FALSE`.

It's a bit processor-intensive so be patient...

```
### for this, let's see how much area within 3 nautical miles of the coast
### is designated as a protected area.
poly_bc_3nm <- readShapePoly(fn = file.path(dir_spatial, 'ohibc_offshore_3nm'), proj4string = p4s_bc)

mpa_by_rgn <- raster::extract(rast_bc_mpa, poly_bc_3nm)

# mpa_by_rgn_weights <- raster::extract(rast_bc_mpa, poly_bc_3nm, weights = TRUE, normalizeWeights = FA
```

Turning raster::extract output into something useful

`raster::extract()` returns a list of vectors; each list item is a vector of all the cell values contained within one of the polygon features (`poly_bc_3nm@polygons`). At this point, we need to assign a non-generic name to our list items so we can keep track. The following line names the list items according to the `rgn_id` column in the attribute table contained in `poly_bc_3nm@data` (this works since the list is in order of items in the `polygons` slot, which is the same order of attributes in the `data` slot).

```
names(mpa_by_rgn) <- poly_bc_3nm@data$rgn_id

# names(mpa_by_rgn_weights) <- poly_bc_3nm@data$rgn_id
```

To get the data out of annoying list form and into convenient data frame form, this next code chunk does the following:

- create an empty data frame (`mpa_rgn_df`) that we will build up as we unlist each list item.
- loop over each of the list items (named after the `rgn_id`s in the previous step)
  - create a temporary data frame that stores the `rgn_id` (from loop index) and the entire list of cell values for that particular `rgn_id`
  - tack that temp data frame onto the bottom of the existing `data_rgn_df` (first time through, tacks onto an empty data frame)

Voilà! a data frame of cell values by region.

```
### For the dataframe without cell weights, each list is just a
### vector of values, so we can simply assign that to a column in
### the data frame.
mpa_rgn_df <- data.frame()
for (rgn_id in names(mpa_by_rgn)) {
  temp_df <- data.frame(rgn_id, year = unlist(mpa_by_rgn[[rgn_id]]))
  mpa_rgn_df <- rbind(mpa_rgn_df, temp_df)
}

### for the dataframe with cell weights, each list item is a
### matrix containing value & weight, so we can unlist it into a temp
### matrix and then assign each variable separately.
# mpa_rgn_weights_df <- data.frame()
# for (rgn_id in names(mpa_by_rgn_weights)) {
#   temp_matrix <- unlist(mpa_by_rgn_weights[[rgn_id]])
#   temp_df <- data.frame(rgn_id, year = temp_matrix$value, wt = temp_matrix$weight)
#   mpa_rgn_df <- rbind(mpa_rgn_weights_df, temp_df)
# }
```

Further examples

To see much of this stuff put into action, check out OHIBC LSP script and its associated functions.

For a more complex version that maintains the spatial distribution of the data, check out OHI global 2015 SPP_ICO goal script and its associated functions. In these scripts, a raster of cell IDs is extracted against polygons for many different species; the resulting lists tell which cell IDs contain which species.

This downres_polygons.R script was used to simplify the high resolution global polygon shapefiles (308 MB of polygon fun) into lower resolution files (.7 MB and 11 MB versions) for quicker plotting. While the script itself may not be all that useful to you, there are plenty of examples of digging into and manipulating the details of SpatialPolygonsShapeFile objects. The script first examines each polygon feature, and strips out sub-polygons whose area falls below a certain threshold; then it simplifies the geometry of the remaining polygons using a Douglas-Peuker algorithm.

Appendix

Alt method of rasterizing polygons

Most of what you need to know about rasters is covered in Jamie's spatial data workshop. One added point: an alternative method of rasterizing vector data, when `raster::rasterize()` is causing problems.

Note that one major issue with `gdal_rasterize()` (aside from the arcane argument list) is that in the event of overlapping polygons, there's no place for a decision rule as to which polygon value gets assigned to the cell. If you know there are no overlaps, then this is a very fast and seemingly accurate option.

```
library(gdalUtils)
rast_3nm_gdal <- gdal_rasterize(
    src_datasource = file.path(dir_spatial, 'ohibc_offshore_3nm.shp'),
      # src_datasource needs to be an OGR readable format (e.g. .shp)
      # NOTE: doesn't need source to already be in memory!
    dst_filename = file.path(dir_spatial, 'rast_3nm_gdal.tif'),
      # destination for output
    a = 'rgn_id',
      # the attribute in the shapefile to be assigned to the cell values
    te = c(ext@xmin, ext@ymin, ext@xmax, ext@ymax),
      # extents for output raster
    tr = c(1000, 1000),
      # resolution for x and y; for my projection, 500 m resolution
    tap = TRUE,
      # target aligned pixels - align coords of extent of output to values of -tr
    a_nodata = NA,
      # nodata value for raster; otherwise they will be filled in with zeroes
    output_Raster = TRUE,
      # return output as a RasterBrick?
    verbose = TRUE)
  ### unused but interesting arguments:
    # l,     # layer; maybe needed if src_datasource is .gdb, with mult layers
    # of,    # output format; default = GTiff, so I left it stay as default
    # a_srs  # override projection
```

More fun with shapefiles and polygons

The `rgeos` package has some helpful functions for manipulating SpatialPolygons - including ways to check invalid geometries and typical GIS tools for checking and manipulating geometries, such as `gUnion()`, `gDifference()`, `gIntersection()`, `gBuffer()`, etc.

Finding and fixing invalid polygon geometries in R

Sometimes shape files have errors - common ones are too few points and self-intersections. `rgeos::gIsValid()` can tell you whether the SpatialPolygons* object is valid or not, but it can't fix the problems directly.

- An internet search might show you a few ways to repair invalid geometries, but some (like setting a buffer of zero width) have unintended consequences - e.g. a zero-width buffer just deletes offending polygons entirely. Not cool!
- The `cleangeo` package has tools for identifying problems, and has a tool to repair geometries.
  - It's not on CRAN but you can install from github, as shown in the code.

```
library(rgeos)
gIsValid(poly_bc_rgn, byid = TRUE)
  ### checks valid geometries; in this case, two polygon features
  ### have invalid geometries... how to fix 'em? cleangeo!

library(cleangeo) ### devtools::install_github('eblondel/cleangeo')
poly_bc_rgn_clean <- clgeo_Clean(poly_bc_rgn, print.log = TRUE)
gIsValid(poly_bc_rgn_clean, byid = TRUE) ### all good, yay!

### cleangeo provides an alternate way of checking geometries if
### gIsValid isn't good enough for you
report  <- clgeo_CollectionReport(poly_bc_rgn)
View(report)
issues  <- report %>% filter(valid == FALSE)
issues ### two polygons with problems
```

Plotting polygons with ggplot2

`ggplot` works with data frames; so if you want to plot a polygon, you need to first turn it into a data frame. `ggplot2::fortify()` takes care of this. The `region` argument lets you assign a variable within the attribute table to become a region identifier.

Note: make sure you have all your CRS and projection stuff figured out first!

```
library(ggplot2)
library(RColorBrewer)

### Fortify the existing poly_bc_rgn SpatialPolygonsDataFrame into a
### data frame that ggplot can work with.
poly_rgn_df <- fortify(poly_bc_rgn, region = 'rgn_id') %>%
  rename(rgn_id = id) %>%
  mutate(rgn_id = as.integer(rgn_id))

### Note: Even though we already have a map with harvest data attached,
### I'm not sure that there's a good way to keep all the attributes
```

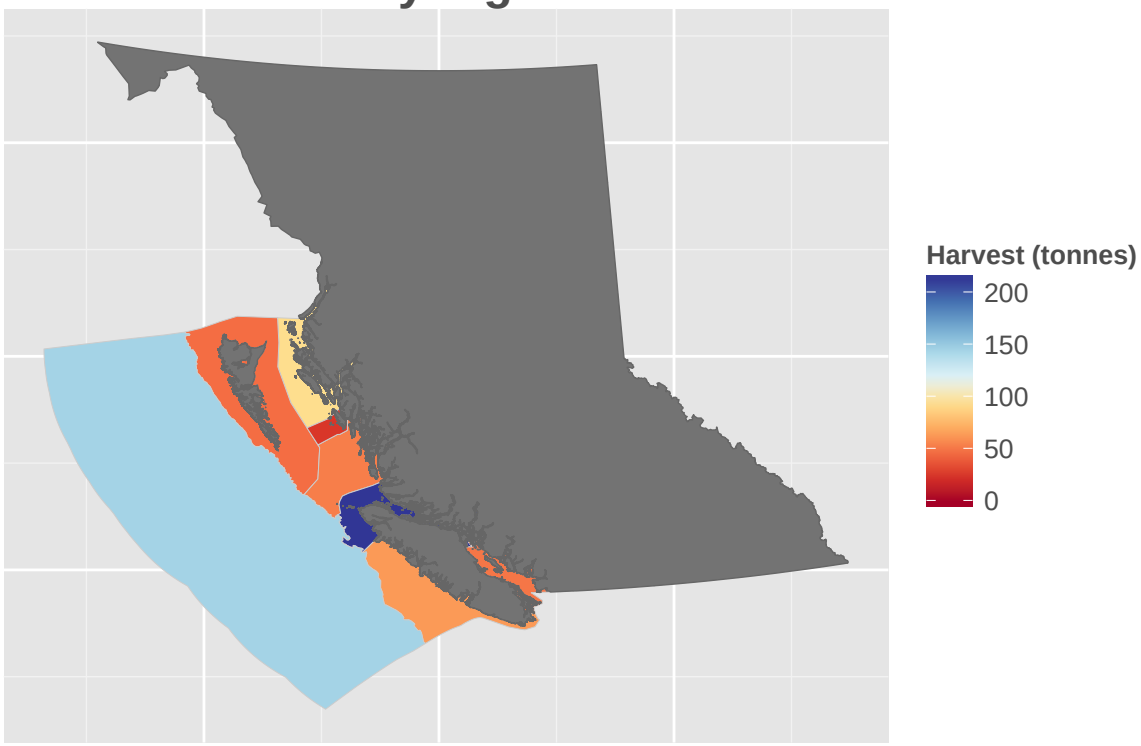
```
### so I'm just left-joining some harvest data to the fortified data frame.
n_rgn <- length(poly_bc_rgn@data$rgn_id)
harv_data2 <- data.frame(rgn_id   = c(1:n_rgn),
                         h_tonnes = abs(rnorm(n = n_rgn, mean = 100, sd = 80)))

poly_rgn_df <- poly_rgn_df %>%
  left_join(harv_data2, by = 'rgn_id')

head(poly_rgn_df)
```

```
##       long      lat order  hole piece group rgn_id h_tonnes
## 1 817328.2 914944.0     1 FALSE     1   1.1      1 93.06008
## 2 816414.7 913763.4     2 FALSE     1   1.1      1 93.06008
## 3 816384.4 913725.2     3 FALSE     1   1.1      1 93.06008
## 4 816376.0 913730.4     4 FALSE     1   1.1      1 93.06008
## 5 816340.1 913734.0     5 FALSE     1   1.1      1 93.06008
## 6 816315.0 913753.1     6 FALSE     1   1.1      1 93.06008
```

```
scale_lim <- c(0, max(poly_rgn_df$h_tonnes))

### To plot land forms, let's load a land shapefile too.
poly_land    <- readShapePoly(fn = file.path(dir_spatial, 'ohibc_land'), proj4string = p4s_bc)
poly_land_df <- fortify(poly_land)
```

```
## Regions defined for each Polygons
```

```
df_plot <- ggplot(data = poly_rgn_df, aes(x = long, y = lat, group = group, fill = h_tonnes)) +
  theme(axis.ticks = element_blank(), axis.text = element_blank(),
        text = element_text(family = 'Helvetica', color = 'gray30', size = 12),
        plot.title = element_text(size = rel(1.5), hjust = 0, face = 'bold'),
        legend.position = 'right') +
  scale_fill_gradientn(colours = brewer.pal(10, 'RdYlBu'), space = 'Lab',
                       na.value = 'gray80', limits = scale_lim) +
  geom_polygon(color = 'gray80', size = 0.1) +
  geom_polygon(data = poly_land_df, color = 'gray40', fill = 'gray45', size = 0.25) +
    ### df_plot order: EEZ score polygons, then land polygons (dark grey).
  labs(title = 'OHI BC harvest by region',
       fill  = 'Harvest (tonnes)',
       x = NULL, y = NULL)


df_plot
```




```
### ggsave() saves the most recent ggplot object
ggsave(file.path(dir_spatial, 'bc_harvest.png'), width = 10, height = 6)
```