Lesson 2

Jan Verbesselt, Loïc Dutrieux, Ben De Vries, Sytze de Bruyn

October 24, 2013

Abstract

This is an introduction to the course Applied Geo-Scripting where we will explore the potential of R and libraries which enable reading, writing, analysis, and visualisation of spatial data.

1 Today's learning objectives

- Read, write, and visualize spatial data (vector/raster) using a script
- Find libraries which offer spatial data handling functions
- Learn to include functions from a library in your script

2 Set Your Working Directory and Load Your Libraries

2.1 Set the Working Directory

Let's do some basic set up first.

- Create a folder which will be your working directory e.g. *Lesson2*
- Create an R script within that folder
- Set your working directory to the *Lesson2* folder
- Create a *data* folder within your working directory

In the code block below type in the file path to where your data is being held and then (if you want) use the setwd() (set working directory) command to give R a default location to look for data files.

```
R> setwd("yourworkingdirectory")
R> #This sets the working directory (where R looks for files)
R> #getwd()
R> # Double check your working directory

R> datdir <- file.path("data") ## path
```

2.2 Load Libraries

Next we will load a series of R packages that will give the functions we need to complete all the exercises in lesson 1 and 2. For this exercise all of the packages should (hopefully) be already installed on your machine (?). We will load them below using the library() command. I also included some comments describing how we use each of the packages in the exercises.

```
R> #----Packages for Reading/Writing/Manipulating Spatial Data---
R> library(rgdal) # reading shapefiles and raster data
R> library(rgeos)
R> library(maptools)
R> library(spdep)   # useful spatial stat functions
R> library(spatstat) # functions for generating random points
R> library(raster)
R> #---Packages for Data Visualization and Manipulation---
R> library(ggplot2)
R> library(reshape2)
R> library(scales)
```

3 Read, plot, and explore spatial data

3.1 Read in a Shapefile

The most flexible way to read in a shapefile is by using the `readOGR` command. This is the only option that will also read in the .prj file associated with the shapefile. NCEAS has a useful summary of the various ways to read in a shapefile: http://www.nceas.ucsb.edu/scicomp/usecases/ReadWriteESRIShapeFiles I recommend always using `readOGR()`.

Read OGR can be used for almost any vector data format. To read in a shapefile, you enter two arguments:

- dsn: the directory containing the shapefile (even if this is already your working directory)
- layer: the name of the shapefile, without the file extension

```
R> download.file('http://rasta.r-forge.r-project.org/kenyashape.zip',
+         file.path(datdir, 'kenyashape.zip'))
R> unzip(file.path(datdir, 'kenyashape.zip'), exdir = datdir)
R> kenya <- readOGR(dsn = datdir, layer = 'kenya')
```

3.2 Plotting the Data

Plotting is easy, use the `plot()` command:

```
R> plot(kenya)
```

Obviously there are more options to dress up your plot and make a proper map/graphic. A common method is to use `spplot()` from the sp package. However I prefer to use the functions available in the ggplot2 package as I think they are more flexible and intuitive. We will address maps and graphics later in the in the class. For now, let us move onto reading in some tabular data and merging that data to our shapefile (similar to the join operation in ArcGIS).

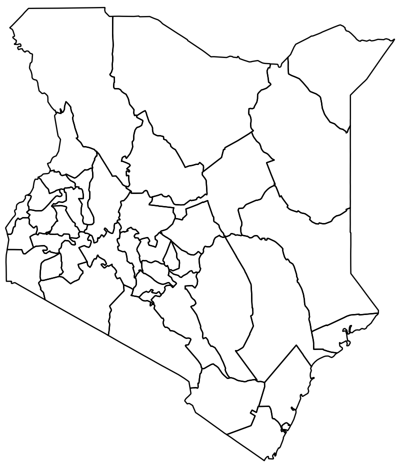

Figure 1: Adminstrative boundaries of Kenya

Here is an example for downloading of administrative boundaries for any country. This will be useful for the exercise.

3.3 Exploring the Data within the vector file

We can explore some basic aspects of the data using `summary()` and `str()`. Summary works on almost all R objects but returns different results depending on the type of object. For example if the object is the result of a linear regression then summary will give you the coefficient estimates, standard errors, t-stats, R^2, et cetera.

```
R> summary(kenya)

Object of class SpatialPolygonsDataFrame
Coordinates:
        min       max
x 33.908859 41.899078
y -4.678047  4.629333
Is projected: FALSE
proj4string : [+proj=longlat +ellps=clrk80 +no_defs]
Data attributes:
    ip89DId               ip89DName
 Min.   :1010   Baringo        : 1
 1st Qu.:3050   Bugoma         : 1
 Median :5030   Busia          : 1
 Mean   :5090   Elgeyo-Marakwet: 1
 3rd Qu.:7060   Embu           : 1
 Max.   :8030   Garissa        : 1
                (Other)        :35

R> str(kenya,2)

Formal class 'SpatialPolygonsDataFrame' [package "sp"] with 5 slots
  ..@ data       :'data.frame':        41 obs. of  2 variables:
  ..@ polygons   :List of 41
  ..@ plotOrder  : int [1:41] 17 36 21 19 12 15 20 14 26 34 ...
  ..@ bbox       : num [1:2, 1:2] 33.91 -4.68 41.9 4.63
  .. ..- attr(*, "dimnames")=List of 2
  ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slots
```

As mentioned above, the `summary()` command works on virtually all R objects. In this case it gives some basic information about the projection, coordinates, and data contained in our shapefile

The `str()` or structure command tells us how R is actually storing and organizing our shapefile. This is a useful way to explore complex objects in R. When we use `str()` on a spatial polygon object, it tells us the object has five 'slots':

1. *data*: This holds the data.frame
2. *polygons*: This holds the coordinates of the polygons
3. *plotOrder*: The order that the coordinates should be drawn
4. *bbox*: The coordinates of the bounding box (edges of the shape file)

5. *proj4string*: A character string describing the projection system

The only one we want to worry about is data, because this is where the data.frame() associated with our spatial object is stored. We access slots using the @ sign.

```
R> #--------------------------ACCESS THE SHAPEFILE DATA------------------------------
R> dsdat <- as(kenya, "data.frame")  # extract the data into a regular data.frame
R> head(dsdat)
```

```
  ip89DId ip89DName
0    1010   Nairobi
1    2010    Kiambu
2    2020 Kirinyaga
3    2030   Muranga
4    2040 Nyandaura
5    2050     Nyeri
```

```
R> kenya$new <- 1:nrow(dsdat) # add a new colunm, just like adding data to a data.frame
R> head(kenya@data)
```

```
  ip89DId ip89DName new
0    1010   Nairobi   1
1    2010    Kiambu   2
2    2020 Kirinyaga   3
3    2030   Muranga   4
4    2040 Nyandaura   5
5    2050     Nyeri   6
```

4 Read in a .csv File and Join it to the Shapefile

4.1 Read in a .csv file

First lets read in a *.csv* file using `read.csv()`

```
R> #--------------------------READ AND EXPLORE A CSV------------------------------
R> filepath <- system.file("extdata/kenpop89to99.csv", package ="rasta")
R> d <- read.csv(filepath)
```

Before we merge the csv file to our shapefile, let's do some basic cleaning. The csv file has some excess columns and rows. Let's get rid of them. We access rows and columns by using square brackets [,].

Here are some examples using are data.frame 'd':

- d[1,] first row, all columns
- d[,1] first column all rows
- d[1,1] item in the first row and first column
- d[,1:5] columns 1 through 5 (also works with rows)

- d[,c(1,4,5)] columns 1,4 and 5 (also works with rows)
- d[,'variable'] column names 'variable'
- d$variable same as above, but returns the column as a vector
- d[d$variable>10,] rows from all columns that correspond where the values in 'variable' are greater than 10

Hopefully you get the idea. See the R cheat sheet: http://cran.r-project.org/doc/contrib/Short-refcard.pdf for more information. Now we extract only the columns we we want and then use the unique() command to get rid of duplicate rows.

```
R> #--------------------------EXTRACT COLUMNS FROM CSV------------------------------
R> d <- d[,c('ip89DId','PopChg','BrateChg','Y89Pop','Y99Pop')]
R> #Grab only the colunms we want
R> names(d)

[1] "ip89DId"  "PopChg"   "BrateChg" "Y89Pop"   "Y99Pop"

R> #str(d)
R> nrow(d)

[1] 48

R> d <- unique(d)  #get rid of duplicate rows
R> nrow(d) #note we now have less rows

[1] 41
```

4.2 Join the csv file to our Shapefile

In R there a variety of options available for joining data sets. The most simple and intuitive is the `merge()` command (see ?merge for details). Merge takes two data.frames and matches them based on common attributes in columns. If the user does not specify the name(s) of the columns then merge will just look for common column names and perform the join on those. However with spatial objects the process is a little more tricky. Unfortunately merge automatically re-orders the new merged data.frame based on the common columns. This will not work for a spatial object as the associated shapes (points, lines, or polygons) would have to be reordered as well. There are a variety of ways around this and I will show a simple one below.

First I will demonstrate the basic merge() function. Then I will show one method for merging tabular to spatial data.

```
R> #----------------EXPLORE MERGE AND DO A TABLE JOIN------------------------------
R> #--------------------First a basic Merge Just to Demonstrate-------------
R> d2 <- kenya@data
R> d3 <- merge(d,d2)
R> #They have common column names so we don't have to specify what to join on
R> head(d3)
```

```
  ip89DId PopChg BrateChg  Y89Pop  Y99Pop ip89DName new
1    1010     57      -12 1325620 2085820   Nairobi   1
2    2010     52      -14  908120 1383300    Kiambu   2
3    2020     16      -15  389440  452180 Kirinyaga   3
4    2030    -14      -31  862540  737520   Muranga   4
5    2040     34      -21  348520  468300 Nyandaura   5
6    2050      6      -23  607980  644380     Nyeri   6

R> d4 <- merge(d,kenya) #This will produce the same result.
R> head(d4)

  ip89DId PopChg BrateChg  Y89Pop  Y99Pop ip89DName new
1    1010     57      -12 1325620 2085820   Nairobi   1
2    2010     52      -14  908120 1383300    Kiambu   2
3    2020     16      -15  389440  452180 Kirinyaga   3
4    2030    -14      -31  862540  737520   Muranga   4
5    2040     34      -21  348520  468300 Nyandaura   5
6    2050      6      -23  607980  644380     Nyeri   6

R> #-------------Now lets do the Table Join: Join csv data to our Shapefile-----
R> #--We can do the join in one line by using the match() function
R> ds1 <- kenya #make a copy so we can demonstrate 2 ways of doing the join
R> kenya@data <- data.frame(as(kenya,'data.frame'),
+                        d[match(kenya@data[,'ip89DId'], d[,'ip89DId']),])

R> #---Alternativley we can do this :
R> #This is the preferred method but will only work if kenya and d have
R> # the same number of rows, and the row names are identical and in the same order
R> row.names(d) <- d$ip89DId
R> row.names(ds1) <- as.character(ds1$ip89DId)
R> d <- d[order(d$ip89DId),]
R> ds1 <- spCbind(ds1,d)
R> head(kenya@data)
```

Note that the values from our csv are not in the data attributes of the shapefile. Note also that we have duplicated the join field 'ip89DId'. We can delete it afterwards but it's a nice way to double check and make sure our join worked correctly. I will go over the details of this approach in class and you can also see an explanation here: http://stackoverflow.com/questions/3650636/how-to-attach-a-simple-data-frame-to-a-spatialpolygondataframe-in-r

5 Create Random Points and Extract as a Text File

We are going to do a point in polygon spatial join. However before we do that we are going to generate some random points. We will use the function `runifpoint()` from the spatstat package. This function creates N points drawn from a spatial uniform distribution (complete spatial randomness) within a given bounding box. The bounding box can be in a variety of forms but the most straightforward is simply a four element vector with *xmin* (the minimum x coordinate), *xmax*, *ymin*, and *ymax*. In the code below we will extract this box from our Kenya data set, convert it to a vector, generate the points, and then plot the points on top of the Kenya map.

```
R> #--------------------------GENERATE RANDOM POINTS------------------------------
R> win <- bbox(kenya) #the bounding box around the Kenya dataset
R> win

        min       max
x 33.908859 41.899078
y -4.678047  4.629333

R> win <- t(win) #transpose the bounding box matrix
R> win

           x         y
min 33.90886 -4.678047
max 41.89908  4.629333

R> win <- as.vector(win) #convert to a vector for input into runifpoint()
R> win

[1] 33.908859 41.899078 -4.678047  4.629333

R> dran1 <- runifpoint(100, win = as.vector(t(bbox(kenya)))) #create 100 random points

R> win <- extent(kenya)
R> dran2 <- runifpoint(n = 100, win = as.vector(win))

R> plot(kenya)
R> plot(dran1, add = TRUE, col = "red")
R> plot(dran2, add = TRUE, col = "blue", pch = 19, cex = 0.5)
```

Now that we have created some random points, we will extract the x coordinates (longitude), y coordinates (latitude), and then simulate some values to go with them.

```
R> #--------------------------CONVERT RANDOM POINTS TO DATA.FRAME------------------
R> dp <- as.data.frame(dran1) #This creates a simple data frame with 2 colunms, x and y
R> head(dp)

         x          y
1 36.41245 -1.0873490
2 34.78763 -4.0107463
3 38.39446 -1.1131782
4 37.71781 -3.5016160
5 35.82809 -0.5046566
6 34.61053  0.4828169
```

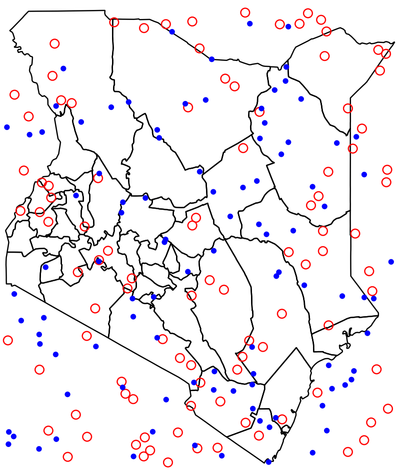

Figure 2: Random points within the Kenya shape file

```
R> #Now we will add some values that will be aggregated in the next exercise
R> dp$values<-rnorm(100,5,10)
R> #generates 100 values from a Normal distribution with mean 5, and sd-10
R> head(dp)

         x          y      values
1 36.41245 -1.0873490   0.2206701
2 34.78763 -4.0107463  19.9842607
3 38.39446 -1.1131782   7.4135826
4 37.71781 -3.5016160  -0.7501701
5 35.82809 -0.5046566 -15.9941109
6 34.61053  0.4828169   3.2820879
```

6 Do a Point in Polygon Spatial Join

In the last exercise we generated some random points along with some random values. Now we will read that data in, convert it to a shapefile (or a SpatialPointsDataFrame object) and then do a point in polygon spatial join. The command for converting coordinates to spatial points is *SpatialPointsDataFrame()*

```
R> #--------------------------CONVERT RANDOM POINTS TO SPATIAL POINTS DATAFRAME----
R> dsp <- SpatialPointsDataFrame(coords = dp[,c('x','y')], data = data.frame('values' = dp
R> summary(dsp)
```

```
Object of class SpatialPointsDataFrame
Coordinates:
        min       max
x 33.955488 41.764508
y -4.664786  4.584642
Is projected: NA
proj4string : [NA]
Number of points: 100
Data attributes:
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-16.5900   0.1363   5.5320   5.5600  12.1700  31.1800
```

```
R> #---Since the Data was Generated from a source with same projection as our Kenya data,
R> dsp@proj4string <- kenya@proj4string
```

Now that we have created some points and defined their projection, we are ready to do a point in polygon spatial join. We will use the `over()` command (short for overlay()).

In the over() command we feed it a spatial polygon object (ds), a spatial points object (dsp), and tell it what function we want to use to aggregate the spatial point up. In this case we will use the mean (but we could use any function or write our own). The result will give us a data.frame, and we will then put the resulting aggregated values back into the data.frame() associated with ds (ds@data).

See ?over() for more information.

```
R> #--------------------------POINT IN POLY JOIN------------------------------
R>
R> #--The data frame tells us for each point the index of the polygon it falls into
R> dsdat <- over(kenya, dsp, fn = mean) #do the join
R> head(dsdat) #look at the data
```

```
  values
0     NA
1     NA
2     NA
3     NA
4     NA
5     NA
```

```
R> inds <- row.names(dsdat) #get the row names of dsdat so that we can put the data back i
R> head(inds)
```

```
[1] "0" "1" "2" "3" "4" "5"
```

```
R> str(kenya@data)
```

```
'data.frame':        41 obs. of  8 variables:
 $ ip89DId  : int  1010 2010 2020 2030 2040 2050 3010 3020 3030 3040 ...
 $ ip89DName: Factor w/ 41 levels "Baringo","Bugoma",..: 26 11 13 25 30 31 12 17 19 24 ...
```

```
 $ new      : int  1 2 3 4 5 6 7 8 9 10 ...
 $ ip89DId.1: int  1010 2010 2020 2030 2040 2050 3010 3020 3030 3040 ...
 $ PopChg   : int  57 52 16 -14 34 6 37 31 25 40 ...
 $ BrateChg : int  -12 -14 -15 -31 -21 -23 -11 -1 -16 -18 ...
 $ Y89Pop   : int  1325620 908120 389440 862540 348520 607980 593260 375320 57960 459740 .
 $ Y99Pop   : int  2085820 1383300 452180 737520 468300 644380 813060 490400 72380 643240
```

```
R> kenya@data[inds, 'pntvals'] <- dsdat #use the row names from dsdata to add the aggregat
R> head(kenya@data)
```

```
  ip89DId ip89DName new ip89DId.1 PopChg BrateChg  Y89Pop
0    1010   Nairobi   1      1010     57      -12 1325620
1    2010    Kiambu   2      2010     52      -14  908120
2    2020 Kirinyaga   3      2020     16      -15  389440
3    2030   Muranga   4      2030    -14      -31  862540
4    2040 Nyandaura   5      2040     34      -21  348520
5    2050     Nyeri   6      2050      6      -23  607980
   Y99Pop pntvals
0 2085820      NA
1 1383300      NA
2  452180      NA
3  737520      NA
4  468300      NA
5  644380      NA
```

7 Do a Pixel in Polygon Spatial Join

In this section we will explore another common spatial join operation. In this case you you have raster data that you want to aggregate up to the level of the polygons. A common example is that you have a surface of observed or interpolated temperature measurements and you want to find out what the average (or sum, max, min, et cetera) temperature is for each polygon (which could represent states, counties, et cetera).

```
R> #--------------------------READ AND CROP A RASTER--
R> library(rasta)
R> filepath <- system.file("extdata", "anom.2000.03.tiff", package ="rasta")
R> g <- raster(filepath)
R> # plot
R> plot(g)
R> plot(kenya, add = TRUE) #plot kenay on top to get some sense of the extent
```

```
R> #------Crop the Raster Dataset to the Extent of the Kenya Shapefile
R> gc <- crop(g, kenya) #clip the raster to the extent of the shapefile
R> #Then test again to make sure they line up
R> plot(gc)
R> plot(kenya, add = TRUE)
```

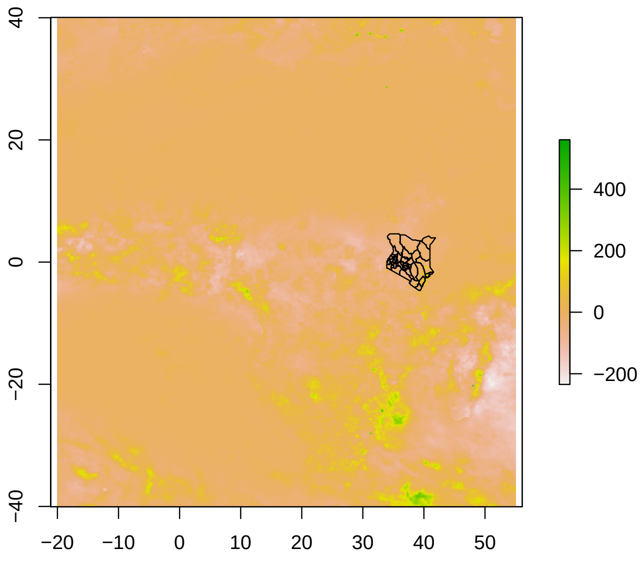

Figure 3: Temperature anomaly for Africa (for March 2003)

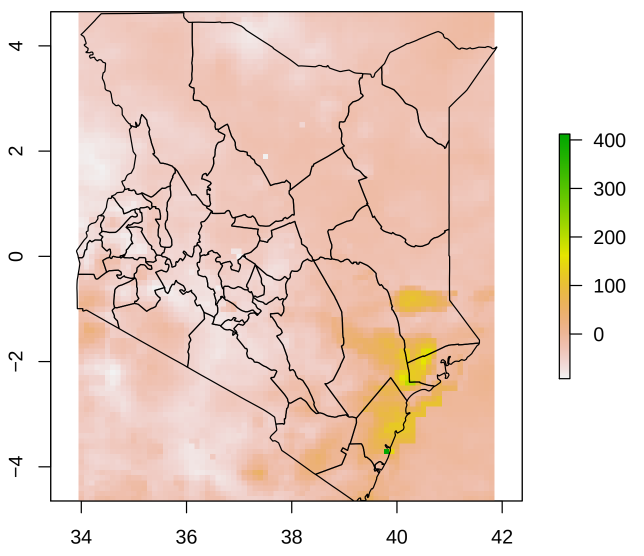

Figure 4: Kenian temperature anomaly for March 2003

In the last step we read in a raster file, cropped it to the extent of the Kenya data (just to cut down on the file size and demonstrate that function). Now we will aggregate the pixel values up the polygon values using the `extract()` function.

```
R> #--------------------------PIXEL IN POLY SPATIAL JOIN------------------------------
R> #Unweighted- only assigns grid to district if centroid is in that district
R> kenya@data$precip <- extract(gc, kenya, fun = mean, weights=FALSE)
```

Weighted (more accurate, but slower) weights aggregation by the amount of the grid cell that falls within the district boundary:

```
R> kenya@data$precip_wght <- extract(gc, kenya, fun = mean, weights = TRUE)
R> #If you want to see the actual values and the weights associated with them do this:
R> rastweight <- extract(gc, kenya, weights = TRUE)
```

Now that we've added all this data to our shapefile, we'll write it out as a new shapefile and then load it in to make some maps in the next exercise.

8 Make Maps with ggplot2()

If you have not already done so, load ggplot2 and some related packages. For more info on the ggplot2 and the grammar of graphics see the resources at http://had.co.nz/ggplot2/. The 'gg' in the ggplot2 is short for *The Grammar of Graphics* which references a famous book by the same name. The idea behind the book and the software is to try and decompose any graphic into a set of fundamental elements. We can then use these elements to construct any type of graphic we want (the elements are the grammar), rather than having a different command for every type of graphic out there. We do not have time to do a full overview of ggplot2 but if you click on the link above and scroll down there is a good visual overview of how ggplot2 works. If you have time take a minute to visit the website.

8.1 Setting up the Data with fortify()

The ggplot2() package separates spatial data into 2 elements: (1) the data.frame and 2) the spatial coordinates. If you want to make a map from a shapefile you first have to use the fortify() command which converts the shapefile to a format readable by ggplot2:

```
R> #------------------------PREP SPATIAL DATA FOR GGPLOT WITH FORTIFY()------------
R> pds <- fortify(kenya, region='ip89DId') #convert to form readable by ggplot2
R> pds$ip89DId <- as.integer(pds$id)
R> head(pds)

      long       lat order  hole piece  group   id ip89DId
1 36.90520 -1.164938     1 FALSE     1 1010.1 1010    1010
2 36.91353 -1.165222     2 FALSE     1 1010.1 1010    1010
3 36.91662 -1.165453     3 FALSE     1 1010.1 1010    1010
4 36.93624 -1.175885     4 FALSE     1 1010.1 1010    1010
5 36.93929 -1.178597     5 FALSE     1 1010.1 1010    1010
6 36.93855 -1.180768     6 FALSE     1 1010.1 1010    1010
```

Now, we will build the map step by step using ggplot2. We could do it all in one line, but it's easier to do it one step at a time so you can see how the different elements combine to make the final graphic. In the code below we will first create the basic layer using the ggplot command, and then we customize to it.

```
R> #--------------------------MAKE A BASIC MAP-----------------------------
R> p1 <- ggplot(d)
R> p1 <- p1 + geom_map(aes(fill = PopChg, map_id = ip89DId), map = pds)
R> p1 <- p1 + expand_limits(x = pds$long, y = pds$lat) #this is sometimes necessary to kee
R> p1 <- p1 + coord_equal() #this keeps the map from having that 'squished' look
R> p1
```

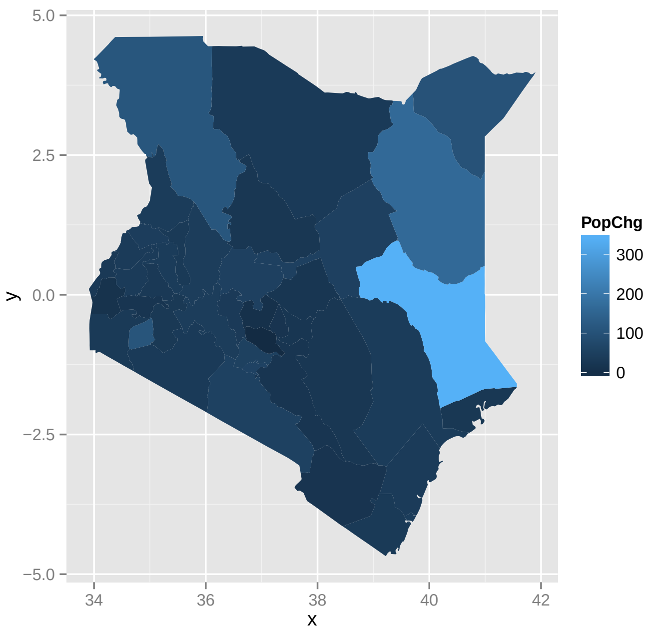

Figure 5: Basic Map with Default Elements

Now we have a basic map, let's make some tweaks to it.

```
R> #--------------------------CHANGE THE COLOR SCHEME, TWEAK THE LEGEND------------
R> #---Change the Colour Scheme-----
R> p1 <- p1 + scale_fill_gradient(name='Population \nChange',
+                    low ='wheat', high = 'steelblue')
R> #to set break points, enter in breaks=c(...,..)
R> #The \n in Population \nChange' indicates a carriage return
R> p1
```

Now we will get rid of all the unnecessary information in the background.

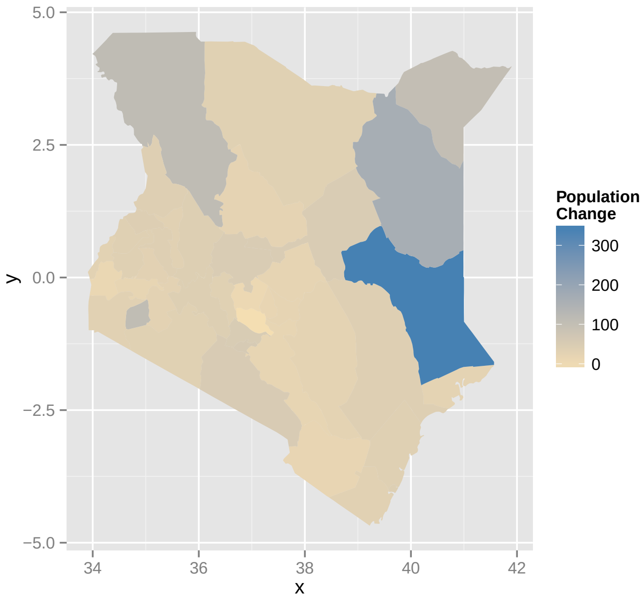

Figure 6: We Changed the Color Scale and Gave the Legend a Proper Name

```
R> #--------------------------EDIT THE BACKGROUND-----------------------------
R> #-----Get Rid of the Background-----
R> #Blank Grid, Background,Axis,and Tic Marks
R> bGrid<-theme(panel.grid = element_blank())
R> bBack<-theme(panel.background = element_blank())
R> bAxis<-theme(axis.title.y = element_blank())
R> bTics<-theme(axis.text = element_blank(), axis.text.y = element_blank(), axis.ticks = e
R> p1<-p1 + bAxis + bTics + bGrid + bBack + xlab('')
R> p1
```

Now let's label the polygon names and data values.

```
R> #--------------------------ADD SOME LABELS--------------------------------------
R> #--------Add Some Polygon labels------
R> #-Polygon Labels
R> cens <- as.data.frame(coordinates(kenya)) #extract the coordinates for centroid of each
R> cens$Region <- kenya$ip89DName
R> cens$ip89DId <- kenya$ip89DId
R> head(cens)  #we will use this file to label the polygons

        V1         V2    Region ip89DId
0 36.85894 -1.2985245   Nairobi    1010
1 36.82240 -1.0743964    Kiambu    2010
2 37.31793 -0.5266225 Kirinyaga    2020
3 37.03273 -0.8108003   Muranga    2030
```

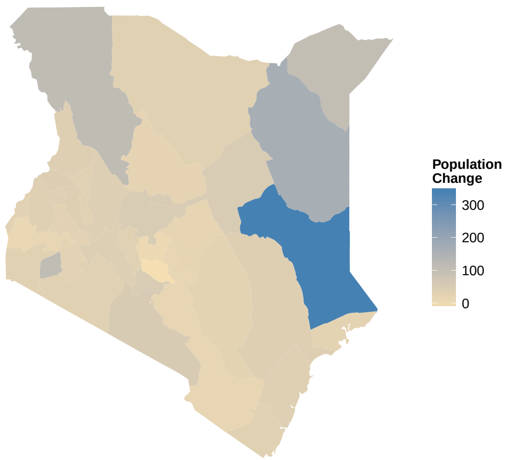

Figure 7: We got rid of all the unneccessary background material

```
4 36.48166 -0.3224750 Nyandaura    2040
5 36.95420 -0.3395780      Nyeri    2050

R> p1 <- p1 + geom_text(data = cens, aes(V1,V2,label = Region), size = 2.5, vjust=1) +
+    labs(title='Population Change in Kenya \n (1989-1999)')
R> p1
```

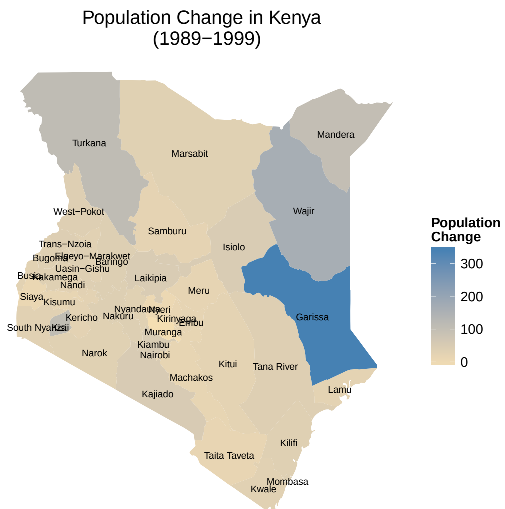

Figure 8: We added text labels and a title

```
R> #-----Add Some value Labels------------
R> pdlab <- merge(cens,d) #Merge the centroids without data
R> head(pdlab) #We will use this to label the polygons with their data values

  ip89DId       V1         V2    Region PopChg BrateChg
1    1010 36.85894 -1.2985245   Nairobi     57      -12
2    2010 36.82240 -1.0743964    Kiambu     52      -14
3    2020 37.31793 -0.5266225 Kirinyaga     16      -15
4    2030 37.03273 -0.8108003   Muranga    -14      -31
5    2040 36.48166 -0.3224750 Nyandaura     34      -21
6    2050 36.95420 -0.3395780     Nyeri      6      -23
   Y89Pop  Y99Pop
1 1325620 2085820
2  908120 1383300
3  389440  452180
4  862540  737520
5  348520  468300
6  607980  644380

R> p1 <- p1 + geom_text(data = pdlab,
+    aes(V1, V2, label = paste("(",PopChg,")",sep="")),
+                        colour = "black" ,size = 2,vjust = 3.7)
R> p1
```

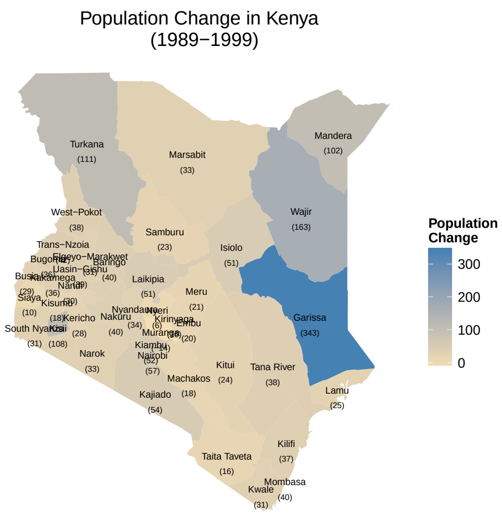

Figure 9: Now we added the actual value labels for the data

8.2 Plotting Panel Maps

So now we have made a basic map with a legend, location labels, and value labels. One of the advantages of ggplot is the ease with which you can create panel graphics, or to use the ggplot terminology 'faceting'. Imagine for example that you have a spatial panel data set-multiple observations of the same spatial feature over several years. Ggplot gives you several options for displaying this data using either the `facet_wrap()` or `facet_grid()` commands. In the example below we will make panel maps for the population data in the Kenya data set.

```
R> #--------------------------RESHAPE THE DATA AND MAKE A PANEL MAP----------------
R> pd <- d[,c('ip89DId','Y89Pop','Y99Pop')] #select out certain colunms
R> pd <- melt(pd, id.vars='ip89DId') # convert the data to 'long' form
R> head(pd) # take a look at the data

  ip89DId variable   value
1    1010   Y89Pop 1325620
2    2010   Y89Pop  908120
3    2020   Y89Pop  389440
4    2030   Y89Pop  862540
5    2040   Y89Pop  348520
6    2050   Y89Pop  607980

R> pmap <- ggplot(pd)
R> p2 <- pmap + geom_map(aes(fill = value,map_id = ip89DId), map=pds) + facet_wrap(~variab
R> p2 <- p2 + expand_limits(x = pds$lon, y = pds$lat) + coord_equal()
R> p2 + labs(title='Basic Panel Map')
```

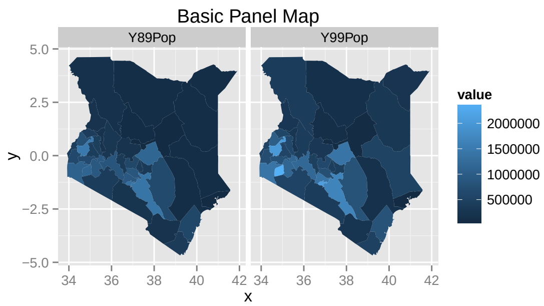

Figure 10: Basic Panel Map

We can use the `ncols' (number of columns) argument in `facet_wrap()` to make the panels stack vertically instead of horizontally.

```
R> #--------------------------TWEAK THE PANEL MAP----------------------------------
R> #If we want to stack the panels vertically we change the options in facet_wrap()
R> p2 <- p2 + facet_wrap(~variable, ncol=1) #have only 1 column of panels
```

Finally we can use the same options we used above to make our final map.

```
R> #--------------------------MORE PANEL MAP TWEAKS------------------------------
R> #--We can add all the other tweaks as before
R> p2 <- p2 + scale_fill_gradient(name='Population',low ='wheat',high = 'steelblue') #to s
R> p2 <- p2 + bAxis+bGrid+bTics+bBack
R> p2 <- p2 + theme(panel.border=element_rect(fill=NA)) #this removes the background but k
R> #--We can also adjust the format, theme, et cetera of the panel lables with 'strip.text
R> p2 <- p2 + theme(strip.background=element_blank(),strip.text.x=element_text(size=12))
R> p2
R> #================================================================================
```

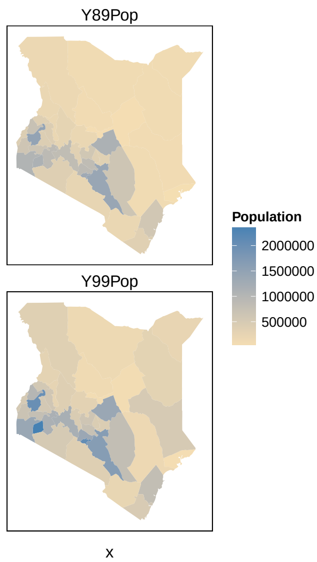

Figure 11: Basic Panel Map

8.3 Plot a Shapefile on Top of a Raster in ggplot

Excercise Lesson 2: Not ready yet – any suggestions?

Random sample the raster, extract temperature data for the point in a buffer of xxxx meters around them and then visualise the temperature data in a 3-D scatterplot. Please provide a clean and documented R script.

```
R> #---Set up the data for ggplot
R> df <- rasterToPoints(gc) #convert the raster to a points object
R> df <- data.frame(df) #and then to a data.frame
R> pds <- fortify(kenya,region='ip89DId')
R> str(df)
R> p<-ggplot(pds)+geom_raster(data=df,aes(x=x,y=y,fill=anom.2000.03))+theme_bw() #use geom
R> p<-p+geom_map(map=pds,aes(map_id=id,x=long,y=lat),fill=NA,colour='black') #then plot a
R> p<-p+coord_equal()
```

```
R> p<-p+scale_fill_gradient(low='wheat',high='blue') #adjust the colors
R> p<-p+labs(x='Longitude',y='Latidude')
R> p
R>
```

9 Excercise

10 Special thanks and more info

Special acknowledgments go to Frank Davenport (Spatial R class) for excellent R spatial introduction on which this lesson is based.