Applied Geo-Scripting: Lesson 1

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

November 4, 2013

Intro of the intro!

- The course is freshly developed based on needs from people how deal with spatial data
- It is a new course, there is a certain flexibility to adjust the content
- A team of R experts, Loic, Ben, Eliakim, Sytze, who use Geo-scripting languages every day helped to develop this course
- Feedback is welcome and we are there to help you during this *Geo-scripting* learning experience
- Avoid copy paste, and make sure you understand basically all steps
  - We will show you how to find help
  - Try to solve the problem as independent as possible
  - Work together with fellow students but make sure you complete everything yourself
- **You need to speak a language in order to learn it!**

Intro of the intro!

- How many student have a laptop?
- Student ID's
- At end of the week - there will be Go/No Go moment to see if want and can continue with the course based on your motivation and basic scripting language knowledge
- Course planning and practical issues
- The afternoon (14-17u)
  - 2 hours course
  - 1 hour excercise
  - Hand in the exercise the next day 9am.

Geo-scripting learning objectives

- Understand basic concepts of applied scripting for spatial data
- Use functions from a library while writing your script
- Find libraries which offer spatial data handling functions
- Know how to find help (on spatial data handling functions)
- Solve scripting problems (debug, reproducible example, writing functions)
- Read, write, and visualise spatial data (vector/raster) using a script
- Apply learned concepts in a case study with geo-data by solving a spatial/ecological/applied question (e.g. detect forest changes, flood mapping, ocean floor depth analysis, bear movement, etc.) with a raster and vector dataset.

Today's topics

Introduction to the applied geo-scripting:

- Why geo-scripting?
- Getting up to speed with R and loading the 'RASTA' package
- Creating a simple function
- Make a spatial map using a script

RASTA: Reproducible and Applied Spatial and Temporal Analaysis
`http://rasta.r-forge.r-project.org`.
We will install this package and use the data and scripts during this course

Why geo-scripting?

- Reproducible: avoid clicking and you keep track of what you have done
- Efficient: you can write a script to do something for you e.g. multiple times e.g. automatically downloading data
- Build your own tools and functions (e.g. raster filters, MODIS download tool, BFAST package)
- Enable collaboration: sharing scripts, functions, and packages
- Good for finding errors i.e. debugging

This course is fully written with scripting languages (i.e. R and Latex). What are the advantages?

What is a scripting language?

- A scripting language (e.g. R and Python) can interpret and automate the execution of tasks which could alternatively be executed one-by-one by a human operator
- Different from like C/C++/Fortran since these are languages that need to be compiled first.
- Compiling? To transform a program written in a high-level programming language from source code into object code. Programmers write programs in a form called source code. Source code must go through several steps before it becomes an executable program. The first step is to pass the source code through a compiler, which translates the high-level language instructions into object code.
- A scripting language is the glue, between different commands, functions, and objectives without the need to compile it for each OS/CPU Architecture

Different scripting languages for geo-scripting

The main scripting languages for GIS and Remote sensing currently are:

- R
- Python
    - stand-alone (ArcPy and PyQGIS)
    - integrated within ArcGIS, QGIS
- GRASS (grass function are included in QGIS)
- Javascript for geoweb scripting
- Matlab
- IDL (ENVI)

Python versus R

- Python is a general purpose programming language with a clear syntax
- R is particularly strong in statistical computing and graphics
- Installing libraries in Python is sometimes challenging
- Syntactic differences can be confusing (especially when combining these)
- There are many R and Python packages for spatial analyses and for dealing with spatial data
- Scripts in both languages can be combined:
  - call R from Python using **RPy, RPy2**
  - call Python from R `http://rpython.r-forge.r-project.org/`
- Many programs have support for:
  - Python (Blender, Sketchup, QGIS, MySQL, PostGIS)
  - R (GRASS, QGIS, MySQL, PostGIS)

System set-up and overview

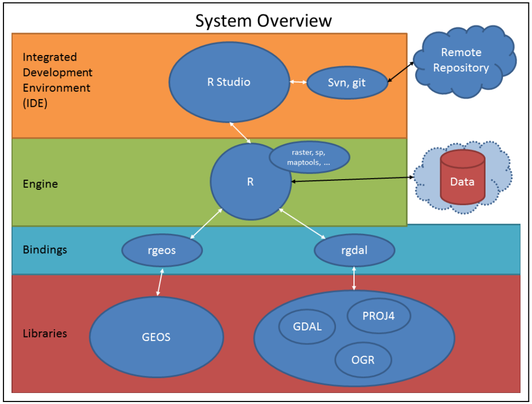

Figure 1 : System set-up

- Version control system (e.g. for scripts): SVN (SubVersioN) and Git
- GDAL/OGR: Geospatial Data Abstraction Library
- GEOS: Geometry Engine and PROJ.4
- R libraries: rgeos, rgdal

Get Your R On

Getting started with Rstudio This preliminary section will cover some basic details about R. For this course we will use Rstudio as an IDE to write and run scripts. Open Rstudio! Now type the following script in the R console:

```
> rm(list = ls())  # Clear the workspace!
> ls() ## no objects left in the workspace
```

```
character(0)
```

A good way to start most R scripts

```
> a <- 1
> a
```

```
[1] 1
```

The first line you passed to the console created a new object named *a* in memory. The symbol '<-' is somewhat equivalent to an equal sign but recommended as it is used internally. In the second line you printed *a* to the console by simply typing it's name.

What is the class of this object?

Get Your R On

```
> class(a)
[1] "numeric"
```

You now have requested the **class** attribute of *a* and the console has returned the attribute: **numeric**. R possesses a simple mechanism to support an object-oriented style of programming. All objects (*a* in this case) have a class attribute assigned to them. **R** is quite forgiving and will assign a class to an object even if you haven't specified one (as you didn't in this case). Classes are a very important feature of the **R** environment. Any function or method that is applied to an object takes into account its class and uses this information to determine the correct course of action.

Set Your Working Directory

Let's do some basic set up first.

- Create a folder which will be your working directory e.g. *Lesson1*
- Create an R script within that folder
- Set your working directory to the *Lesson1* 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.

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

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

Custom Functions

It is hard to unleash the full potential of R without writing your own functions. Luckily it's very easy to do. Here are some trivial examples:

```
> add <- function(x){
+ #put the function arguments in () and the evaluation in {}
+   x + 1
+ }
> add(4)
```

Set the default values for your function

```
> add <- function(x = 5) {
+   z <- x + 1
+   return(z)
+ }
> add()
> add(6)
```

That's about all there is to it. The function will generally return the result of the last line that was evaluated.
How do you write a function that returns x and z?

Now, let's declares a new object, a new function, **newfunc** (this is just a name and if you like you can give this function another name). Appearing in the first set of brackets is an argument list that specifies (in this case) two names. The value of the function appears within the second set of brackets where the process applied to the named objects from the argument list is defined.

```
> newfunc <- function(x, y) {
+   z <- 2*x + y
+   return(c(z,x,y))
+ }
> a2b <- newfunc(2, 4)
> a2b
> rm(a, newfunc, a2b)
```

Next, a new object *a*2*b* is created which contains the result of applying **newfunc** to the two objects you have defined earlier. The second last R command prints this new object to the console. Finally, you can now remove the objects you have created to make room for the next exercise by selecting and running the last line of the code.

Help?!

R is supported by a very comprehensive help system. Help on any function can be accessed by entering the name of the function into the console preceded with a ?. The easiest way to access the system is to open a web-browser. This help system can be started by entering **help.start()** in the R console.
How do you find help about the remove function?

Data Structures

There are several ways that data are stored in R. Here are the main ones:

- **Vectors**: is the most generic data structure. In R, any variable of an atomic data type (numeric, integer, logical, character) is a vector. See examples below.
- **Data Frames**: is the most common format and similar to a spread sheet. A data.frame() is indexed by rows and columns and store numeric and character data. The data.frame is typically what is used when reading in csv files, do regressions, etc.
- **Matrices and Arrays** Similar to data.frames but slightly faster computation wise while sacrificing some of the flexibility in terms of what information can be stored. In R a matrix object is a special case of an array that only has 2 dimensions. i.e., an array is n-dimensional matrix while a matrix only has rows and columns (2 dimensions)
- **Lists** The most common and flexible type of R object. A list is simply a collection of other objects. For example a regression object is a list of: 1) Coefficient estimates 2) Standard Errors and other results.

We will look at examples of these objects in the next sectionl

R packages and the rasta package

R ‘packages’ are user contributed functions. There are about 5000 or so (with a constantly expanding list). If a package is already installed you load the package with the `library()` command. If you want to install a package you can use the `install.packages()` command (you have to provide the url of the CRAN mirror to download the package. If you are using R Studio you can also just click on **Tools>Install Packages**, and type in the name(s) of the package you want to install.
Now install and load the rasta package:

```
> install.packages("rasta", repos="http://R-Forge.R-project.org")
```

```
> library(rasta) ## load the rasta library
```

```
> ?mysummary
> mysummary
```

What does the function do?

Reading Data in and Out

The most common way to read in spread sheet tables is with the *read.csv*() command. However you can read in virtually any type of text file. Type ?read.table in your console for some other examples.

```
> f <- system.file(file.path("extdata", "kenpop89to99.csv"), package ="ra
> mydat <- read.csv(f)
```

We can explore the data using the names(), summary(), head(), and tail() commands (we will use these frequently through out the exercise)

```
> names(mydat)[1:3]
> summary(mydat$Y89Pop)[1:3]
> head(mydat$Y89Births)[1:2]
```

- What is the class of the *mydat*?
- How can we write out the data.frame to a csv file in our working directory?

Lets do a basic regression so you can see an example of a list.

Basic regression and example of a list

We use the `lm()` command to do a basic linear regression. The ~ symbol separates the left and right hand sides of the equation and we use '+' to separate terms and '*' to specify interactions. *Regress the Population in 1999 on the population and birthrate in 1989*

```
> myreg<-lm(Y99Pop ~ Y89Births + Y89Brate, data = mydat)
> myreg[c(1,8)]

$coefficients
(Intercept)   Y89Births    Y89Brate
502592.5928     38.0455 -14369.0931

$df.residual
[1] 45
```

Basic regression and example of a list

A regression object is an example of a list. We can use the `names()` command to see what the list contains. We can use the summary() command to get a standard regression output (coefficients, standard errors, et cetera) and we can also create a new object that contains all the elements of a regression summary.

```
> names(myreg)[1:3]

[1] "coefficients" "residuals"    "effects"

> myregsum <- summary(myreg)
> myregsum[["adj.r.squared"]] #extract the adjusted r squared

[1] 0.8937546

> myregsum$adj.r.squared # does the same thing

[1] 0.8937546
```

why is *myregsum* **a list object? What is the advantage of a list?** That concludes our basic introduction to data.frames and lists. There is alot more material out on the web if you are interested.

Create a spatial map using a script

Here is an example of how you can create a map in R:

```
> ## Load required packages
> library(rasta)
```

Download data from the Global Adminstrative Areas Data base (GADM). There is a function do get *public* data from anywhere in the world. See help of the getData funciton in the **raster** package.

```
> ?getData
> ?raster::getData
```

Read the help to find out how we can find the country codes?

What is the country code of Belgium?

Now we will download the administrative boundaries of the Philippines:

```
> adm <- raster::getData("GADM", country = "PHL",
+                         level = 2, path = datdir)
> plot(adm)
```

Create a spatial map using a script

Try to understand the code below, and let me know if you have questions.
Feel free to use this code as an example and use this for the excercise below.

```
> mar <- adm[adm$NAME_1 == "Marinduque",]
> plot(mar, bg = "dodgerblue", axes=T)
> plot(mar, lwd = 10, border = "skyblue", add=T)
> plot(mar, col = "green4", add = T)
> grid()
> box()
> invisible(text(getSpPPolygonsLabptSlots(mar),
+ labels = as.character(mar$NAME_2), cex = 1.1, col = "white", font = 2)
> mtext(side = 3, line = 1, "Provincial Map of Marinduque", cex = 2)
> mtext(side = 1, "Longitude", line = 2.5, cex=1.1)
> mtext(side = 2, "Latitude", line = 2.5, cex=1.1)
> text(122.08, 13.22, "Projection: Geographic\n
+ Coordinate System: WGS 1984\n
+ Data Source: GADM.org", adj = c(0, 0), cex = 0.7, col = "grey20")
```

Create a spatial map using a script

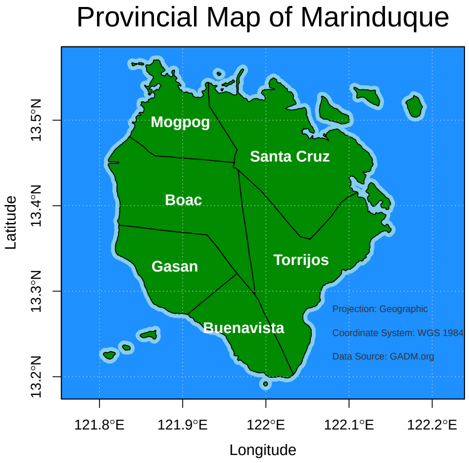

Figure 2 : Adminstrative boundaries of Marinduque

Ex.Lesson 1: Write you own function to create a map

- Submit a clear, reproducible, and documented script containing a function to create a spatial map
    - define a function
    - demonstrate the function
    - make sure the script is tested by somebody else
    - keep it simple (!) e.g. just plot the adminstrative boundaries
- The function should accept *country* and *level* as input arguments
- Filename: *lastnamefirstname.R*

More information

For more information about R please refer to the following links:

- http://www.statmethods.net/index.html This is a great website for learning R function, graphs, and stats.
- the book on Applied spatial Data analysis with R http://www.asdar-book.org/ (Bivand et al., 2013).
- Visit http://www.r-project.org/ and check out the Manuals i.e an introductions to R
- Overview of R functionality for spatial data analysis: http://cran.r-project.org/web/views/Spatial.html
- http://gis.stackexchange.com/questions/45327/tutorials-to-handle-spatial-data-in-r
- More info about R code and style guide

Extra challenge - for ggplot lovers

Optional challenge

```
> require(ggmap)
> shp.spdf <- adm[adm$NAME_1=="Marinduque",]
> shp.df <- fortify(shp.spdf)
> shp.centroids.df <- data.frame(long = coordinates(shp.spdf)[,1],
+                                 lat = coordinates(shp.spdf)[,2])
> # Get names and id numbers corresponding to administrative areas
> shp.centroids.df[, "ID_1"] <- shp.spdf@data[,"ID_1"]
> shp.centroids.df[, "NAME_2"] <- shp.spdf@data[,"NAME_2"]
> q <- qmap(location = "Marinduque", zoom = 10, maptype = "satellite")
> q = q +  geom_polygon(aes(x = long, y = lat, group = group),
+     data = shp.df,
+     colour = "white", fill = "black", alpha = .4, size = .3)
> q = q + geom_text(data = shp.centroids.df,
+     aes(label = NAME_2, x = long, y = lat, group = NAME_2),
+                   size = 3, colour= "white")
> print(q)
```

Bivand, R. S., Pebesma, E. J., & Rubio, V. G. (2013). Applied spatial data analysis with R, .