Lesson 5: Introduction to raster based analysis in R

Loïc Dutrieux, Ben deVries, Jan Verbesselt

October 29, 2013

1 Today's learning objectives

At the end of the lecture, you should be able to:

- Read/write raster data
- Perform basic raster file operations/conversions
- Perform simple raster calculations

2 Assumed knowledge from previous lectures

- Understand system architecture (R, R packages, libraries, drivers, bindings)
- Read and write vector data
- Good scripting habits

3 Reminder on overall system architecture

In a previous lecture you saw the overall system architecture (Figure 1) and you saw how to read and write vector data from/to file into you R environment. These vector read/write operations were made possible thanks to the OGR library. The OGR library is interfaced with R thanks to the rgdal package/binding. By analogy, raster data can be read/written thanks to the GDAL library. Figure 1 provides an overview of the connections between these elements. GDAL stands for Geospatial Data Abstraction Library. You can check the project home page at `http://www.gdal.org/` and you will be surprised to see that a lot of the software you have used in the past to read gridded geospatial data use GDAL. (i.e.: ArcGIS, QGIS, GRASS, etc). In this lesson, we will use GDAL indirectly via the raster package, which uses rgdal extensively. However, it is possible to call GDAL functionalities directly through the command line (a shell is usually provided with the GDAL binary distribution you installed on your system), which is equivalent to calling a `system()` command directly from within R. In addition, if you are familiar with R and its string handling utilities, it should facilitate the building of the expression that have to be passed to GDAL.

Perform system setup checks.

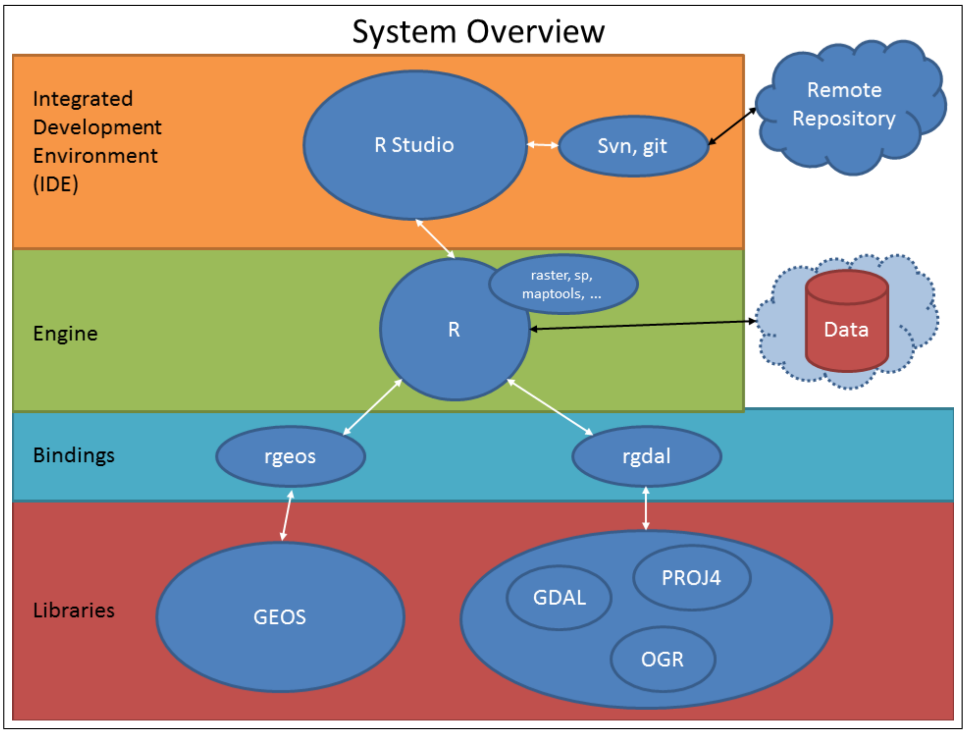

Figure 1: System architecture overview

```
> # Example to perform system set-up checks
> library(rgdal)
> getGDALVersionInfo()

[1] "GDAL 1.9.2, released 2012/10/08"
```

The previous function should return the version number of the current version of GDAL installed on your machine. 1.10.1 is the most recent stable release. In case the function above returns an error, or if you cannot install rgdal at all, you should verify that all required software and libraries are properly installed. Please refer to the system setup vignette in that same package.

4 Overview of the raster package

The raster package is the reference R package for raster processing, Robert J. Hijmans is the original developer of the package. The introduction of the raster package to R has been a revolution for geo processing and analysis using R. Among other things the raster package allows to:

- Read and write raster data of most commonly used format (thanks to extensive use of rgdal)
- Perform most raster operations (creation of raster objects, performing spatial/geometric operations (re-projections, resampling, etc), filtering and raster calculations)

- Work on large raster datasets thanks to its built-in block processing functionalities
- Perform fast operations thanks to optimized back-end C code
- Visualize and interact with the data
- etc...

Check the home page of the raster package (http://cran.r-project.org/web/packages/raster/), the package is extremely well documented, including vignettes and demos.

Explore the raster objects

The raster package produces and uses R objects of three different classes. The **RasterLayer**, the **RasterStack** and the **RasterBrick**. A RasterLayer is the equivalent of a single layer raster, as an R workspace variable. The data themselves, depending on the size of the grid can be loaded in memory or on disk. The same stands for RasterBrick and RasterStack objects, which are the equivalent of multi-layer RasterLayer objects. RasterStack and RasterBrick are very similar, the difference being in the virtual characteristic of the RasterStack. While a RasterBrick has to refer to one multi-layer file or is in itself a multi-layer object with data loaded in memory, a RasterStack may "virtually" connect several raster objects written to different files or in memory. Processing will be more efficient for a RasterBrick than for a RasterStack, but RasterStack has the advantage of facilitating pixel based calculations on separate raster layers.

Let's take a look into the structure of these objects.

```
> library(raster)
> # Generate a RasterLAyer object
> r <- raster(ncol=40, nrow=20)
> class(r)

[1] "RasterLayer"
attr(,"package")
[1] "raster"

> # Simply typing the object name displays its general properties / metadata
> r

class       : RasterLayer
dimensions  : 20, 40, 800  (nrow, ncol, ncell)
resolution  : 9, 9  (x, y)
extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

From the metadata displayed above, we can see that the RasterLayer object contains all the properties that geo-data should have; that is to say a projection, an extent and a pixel resolution.
RasterBrick and RasterStack objects can also fairly easily be generated directly in to R, as shown in the example below. Being able to generate such objects without reading them from

files is particularly important for the generation of "reproducible examples" which was covered in general terms in Lesson 2.

```
> # Using the previously generated RasterLAyer object
> # Let's first put some values in the cells of the layer
> r[] <- rnorm(n=ncell(r))
> # Create a RasterStack objects with 3 layers
> s <- stack(x=c(r, r*2, r))
> # The exact same procedure works for creating a RasterBrick
> b <- brick(x=c(r, r*2, r))
> # Let's look at the properties of of one of these two objects
> b
```

```
class       : RasterBrick
dimensions  : 20, 40, 800, 3  (nrow, ncol, ncell, nlayers)
resolution  : 9, 9  (x, y)
extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
data source : in memory
names       :   layer.1,   layer.2,   layer.3
min values  : -2.972906, -5.945812, -2.972906
max values  :  2.790868,  5.581736,  2.790868
```

The RasterBrick metadata displayed above are mostly similar to what we saw earlier for the RasterLayer object, with the exception that these are multi-layer objects.

5 Raster objects manipulations

Reading and writing from/to file

The actual data used in geo processing projects often comes as geo-data, stored on files such as GeoTiff or other comonly used file formats. Reading data directly from these files into the R working environment (as objects belonging to one of the 3 raster objects classes), is made possible thanks to the raster package. The three main commands for reading raster objects from file are the `raster()`, `stack()`, and `brick()` functions, refering to RasterLayer, RasterStack and RasterBrick objects respectively.
Writing one of the three raster objects classes to file is achieved with the `writeRaster()` function.
To illustrate the reading and writing of raster files, we will use the data stored in the /inst/extdata/ directory within the rasta package. The `system.file()` command is used to access these "external" data saved into packages.
Gewata is the name of the data set added, it is a multi Layer tiff object, its file name is LE71700552001036SGS00_SR_Gewata_INT1U.tif, informing us that this is a subset from a scene acquired by the Landsat 7 sensor. Let's not worry about the region that the data cover for now, we will find a nice way to discover that later on in that tutorial. See the example below.

```
> # Reading a multilayer raster object
> gewata <- brick(system.file("extdata/LE71700552001036SGS00_SR_Gewata_INT1U.tif",
+                              package="rasta"))
```

Let's take a look at the structure of this object.

```
> gewata

class       : RasterBrick
dimensions  : 593, 653, 387229, 6  (nrow, ncol, ncell, nlayers)
resolution  : 30, 30  (x, y)
extent      : 829455, 849045, 825405, 843195  (xmin, xmax, ymin, ymax)
coord. ref. : +proj=utm +zone=36 +ellps=WGS84 +units=m +no_defs
data source : C:\Program Files\R\R-3.0.0\library\rasta\extdata\LE71700552001036SGS00_SR_Ge
names       : LE7170055//ta_INT1U.1, LE7170055//ta_INT1U.2, LE7170055//ta_INT1U.3, LE71700
min values  :                     4,                     6,                     3,
max values  :                    39,                    56,                    71,
```

The metadata above informs us that the gewata object is a relatively small (593x653 pixels) RasterBrick with 6 layers.
Similarly, single layer objects can be read using the `raster()` function. Or if you try using the `raster()` function on a multi layer object, by default the first layer only will be read.

```
> gewataB1 <- raster(system.file("extdata/LE71700552001036SGS00_SR_Gewata_INT1U.tif",
+                                 package="rasta"))
> gewataB1
```

Note that in addition to supporting most commonly used geodata formats, the raster package has its own format. Saving a file using the `.grd` extension (`'filename.grd'`) will automatically save the object to the raster package format. This format has great advantages when performing geo processing in R (one advantage for instance is that it conserves original filenames as layer names in multilayer objects), however, saving your final results in this file format might be risky as the GDAL drivers for that file format do not seem to exist yet.

Geo processing, in memory Vs. on disk

When looking at the documentation of most functions of the raster package, you will notice that the list of arguments is almost always ended by ... It is called and ellipsis, and means that extra arguments can be passed to the function. Often these arguments are those that can be passed to the `writeRaster()` function; meaning that most geo-processing functions are able to write their output directly to file, on disk. This reduces the number of steps and is always a good consideration when working with big raster objects that tend to overload the memory if not written directly to file.

Croping a raster object

`crop()` is the raster package function that allows you to crop data to smaller spatial extents. A great advantage of the crop function is that it acceps almost all spatial object class of R as its "extent" input argument. But the "extent" argument also simply accepts objects of class "extent". One way of obtaining such an extent object interactively is by using the `drawExtent()` function. In the example below, we will manually draw a regular extent that we will use later to crop the gewata RasterBrick.

```
> # PLot the first layer of the RasterBrick
> plot(gewata, 1)
> e <- drawExtent(show=TRUE)
```

Now you have to define a rectangular bounding box that will define the spatial extent of the extent object. Click twice, for the two opposite corners of the rectangle. Now we can crop the data following the boundaries of this extent.

```
> # Crop gewata using e
> gewataSub <- crop(gewata, e)
> # Now visualize the new cropped object
> plot(gewataSub, 1)
```

You should see on the resulting plot that the orinal image has been cropped.

Creating layer stacks

To end this section on general files and raster objects manipulations, we will see how multi layer objects can be created from single layer objects. The object created as part of the example below is the same that we will use in Lesson 6 to perform time series analysis on raster objects. It is composed of NDVI layers derived from Landat acquisitions at different dates. The objective is therefore to create a multi layer NDVI object, for which each layer corresponds to a different date. But first we need to fetch the data, since they were a bit too big to be kept into the rasta package, we left them online.

```
> # start by making sure that your working directory is properly set
> # If not you can set it using setwd()
> getwd()
> download.file(url='http://rasta.r-forge.r-project.org/tura.zip', destfile='tura.zip')
> unzip(zipfile='tura.zip')
> # Retrieve the content of the tura sub-directory
> list <- list.files(path='tura/', full.names=TRUE)
```

The object "list" contains the file names of all the single layers we have to stack. Let's open the first one to visualize it.

```
> plot(raster(list[1]))
```

We see a NDVI layer, with the cloud masked out. Let's now create the stack, the function for doing that is called `stack()`. Looking at the help page of the function , you can see that it can accep a list of file names as argument, which is exactly what the object "list" is. So we can very simply create the layer stack by running the function.

```
> turaStack <- stack(list)
> turaStack
```

Now that we have our 166 layers RasterStack in memory, let's write it to disk using the `writeRaster()` function. Note that we decide here to save it as .grd file (the native format of the raster package); the reason for that is that this file format conserves original file names (in which information on dates is written) in the individual band names. The data range is comprised between -10000 and +10000, therefore such a file can be stored as signed double integer (INT2S).

```
> # Write this file at the root of the working directory
> writeRaster(x=turaStack, filename='turaStack.grd', datatype='INT2S')
```

Now this object is stored on your computer, ready to be archived for later use.

6 Simple raster arithmetic

Adding, substracting, multiplying and dividing RasterLayers

Performing simple raster operations with raster objects is fairly easy. For instance, if you want to substract two RasterLayers of same extent, r1 and r2; simply doing r1 - r2 will give the expected output, which is, every pixel value of r2 will be substracted to the matching pixel value of r1. These types of pixel based operations almost always require a set of conditions to be met in order to be executed; the two RasterLayers need to be identical in term of extent, resolution , projection, etc.

Subsetting layers from from RasterStack and RasterBrick

Different spectral bands of a same satellite image scene are often stored in multi layers objects. This means that you will very likely import them in you R working environment as RasterBrick or RasterStack objects. As a consequence, to perform calculations between these bands, you will have to write an expression refering to individual layers of the object. Referring to individual layers in a RasterBrick or RasterStack object is done by using double square brackets "`[[]]`". Let's look for instance at how the famous NDVI index would have to be calculated from the gewata RasterBrick object read earlier, and that contains the spectral bands of Landsat 7 sensor. And in case you have forgotten, the ndvi formula is as follows.

$$NDVI = \frac{NIR - R}{NIR + R} \tag{1}$$

with NIR and R being band 4 and 3 of Landsat respectively.

```
> ndvi <- (gewata[[4]] - gewata[[3]]) / (gewata[[4]] + gewata[[3]])
```

The `plot()` function automatically recognises the objects of raster classes and returns an appropriate spatial plot.

```
> plot(ndvi)
```

The resulting NDVI can be viewed in Figure 2. As expected the NDVI ranges from about 0.2, which corresponds to nearly bare soils, to 0.9 which means that there is some dense vegetation in the area.

Although this is a quick way to perform the calculation, directly adding, substracting, multiplying, etc, the layers of big raster objects is not recommended. When working with big

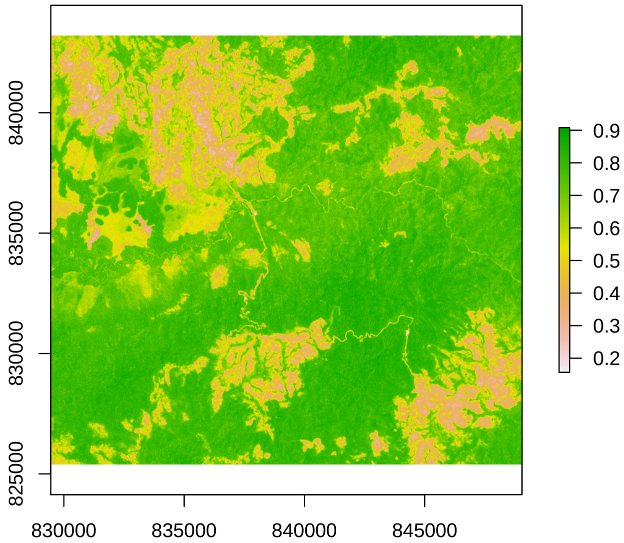

Figure 2: NDVI calculated with the gewata data set

objects, it is advisable to use the `calc()` function to perform these types of calculaions. The reason is that R needs to load all the data first into its internal memory beform performing the calculation and then runs everything in one block. It is really easy to "flood" the RAM when doing that. A big advantage of the `calc()` function is that it has a built-in block processing option for any vectorized function, allowing such calculations to be fully "RAM friendly". The example below illustrates how to calculate NDVI from the same date set using the `calc()` function.

```
> # Define the function to calculate NDVI from
> ndvCalc <- function(x) {
+   ndvi <- (x[[4]] - x[[3]]) / (x[[4]] + x[[3]])
+   return(ndvi)
+ }
> ndvi2 <- calc(x=gewata, fun=ndvCalc)
>
```

Note that `overlay()` can also be used in that case to obtain the same result, with the same level of RAM friendlyness. The advantage of overlay being that the number of input RasterLayers is less limiting. As a consequence specifying the layers does not happen in the function call but in the `overlay()` call instead.

```
> ndvOver <- function(x, y) {
+   ndvi <- (y - x) / (x + y)
+   return(ndvi)
```

```
+ }
> ndvi3 <- overlay(x=gewata[[3]], y=gewata[[4]], fun=ndvOver)
```

We can verify that the three layers ndvi, ndvi2 and ndvi3 are actually identical using the `all.equal()` function from the raster package.

```
> all.equal(ndvi, ndvi2)

[1] TRUE

> all.equal(ndvi, ndvi3)

[1] TRUE
```

In the simple case of calculating NDVI, we were easily able to produce the same result with `calc()` and `overlay()`, however, it is often the case that one function is preferable to the other. As a general rule, a calculation that needs to refer to multiple individual layers separately will be easier to set up in `overlay()` than in `calc()`.

Re-projections

By the way, we still don't know where this area is. In order to investigate that, we are going to try projecting it in Google Earth. As you know Google Earth is all in Lat/Long, so we have to get our data re-projected to lat/long first. The `projectRaster()` function allows re-projection of raster objects to any projection one can think of. As the function uses the PROJ.4[1] library to perform that operation, the `crs=` argument should receive a "proj4" expression. "proj4" expressions are strings that provide the projection parameters of cartographic projections. A central place to search for projections is the spatial reference website (http://spatialreference.org/), from this database you will be able to query almost any reference and retrieve it in any format, including its Proj4 expression.

```
> # One single line is sufficient to project any raster to any projection
> ndviLL <- projectRaster(ndvi, crs='+proj=longlat')
```

Note that if re-projecting and mosaicking is really a large part of your project, you may want to consider using directly the `gdalwarp` command line utility (http://www.gdal.org/gdalwarp.html). Although there is no R interface to `gdalwarp`, the string handling capabilities of R can be of great help when building `gdalwarp` expressions.

Now that we have our ndvi layer in lat/long, let's write it to a kml file, which is one of the two Google Earth formats.

```
> # Since this function will write a file to your working directory
> # you want to make sure that it is set where you want the file to be written
> # It can be changed using setwd()
> getwd()
> # Note that we are using the filename argument, contained in the elypsis of
> # the function, since we want to write the output directly to file.
> KML(x=ndviLL, filename='gewataNDVI.kml')
```

[1]PROJ.4 (https://trac.osgeo.org/proj/) is the reference library (external to R) that handles cartographic projections and performs projections transformations. The rgdal package is the interface between that library and R

Note that you need to have Google Earth installed on your system in order to perform the following step. Now let's find that file that we have just written and double click it, and watch how Google Earth brings us all the way to ... Ethiopia. More information will come in Lesson 6 about that specific area.

We are done with this data set for this lesson. So let's explore another data set, from the newly launched Landsat 8 sensor. This dataset will allow us to find other interesting raster operations to perform.

More raster arythmetics, perform simple replacements

The Landsat 8 data delivered up to now come with a Quality Assessment (QA) band; it is an extra layer, matching the extent and the resolution of the other bands, and containing information about the quality of the data. Such extra QA layers are becomming increasingly common (Landsat 8, most moderate resolution sensors) and are very valuable information. In the following section we will use that QA layer to mask out remaining clouds in the scene.

About the area

The area selected for this exercise is located in Tahiti, French Polynesia. It is located right in the middle of the island, where the two volcanoes forming the island intersect (on the isthmus). According to wikipedia, about 5000 people live in the municipality of Afaahiti-Taravao.
For convenience, this scene subset has been added as a build in data set of the rasta package. Therefore it can simply be loaded using the `data()` command. Let's load it.

```
> library(rasta)
> data(taravao)
> taravao
```

```
class       : RasterBrick
dimensions  : 219, 180, 39420, 9  (nrow, ncol, ncell, nlayers)
resolution  : 30, 30  (x, y)
extent      : 252435, 257835, -1965255, -1958685  (xmin, xmax, ymin, ymax)
coord. ref. : +proj=utm +zone=6 +datum=WGS84 +units=m +no_defs +ellps=WGS84 +towgs84=0,0,0
data source : in memory
names       : LC80530722013233LGN00_B1, LC80530722013233LGN00_B2, LC80530722013233LGN00_B3
min values  :                     8910,                     7910,                     6730
max values  :                    29370,                    30335,                    30555
```

The 9th band has the suffixe QA, which means that it is the one containing the quality information. The information contained in this layer is codded bitwise (see http://landsat.usgs.gov/L8QualityAssessmentBand.php for further details), which means that we need an extra step to extract this information into something we can simply use and understand. This step will be automatically performed using the `QA2cloud{rasta}` function since the details of that step are slightly beyond the scope of this course. When applied to the QA layer, the function returns for each pixel, either the value 1 or the value 0. 1 meaning presence and 0 absence of cloud. As seen earlier, applying a pixel based function to a RasterLayer is what the `calc()` function does well; so let's do it.

```
> # Generate the cloud mask using QA2cloud wrapped into calc()
> cloud <- calc(taravao[[9]], fun=QA2cloud)
> # Replace 0s by NAs (to improve visual display)
> cloud[cloud == 0] <- NA
```

A nice preview of the result can be achived using the `plotRGB()` function. The resulting plot is presented in figure 3.

```
> # Display data and the cloud mask
> plotRGB(taravao, 5, 3, 4)
> # Note the clouds at the top and bottom of the image.
> # Let's plot the cloud mask over them and see how they match
> plot(cloud, add=TRUE, legend=FALSE)
```

Figure 3: RGB colour composite of the Taravao area with cloud mask overlay, the band combination used is 5, 3, 4 (Landsat 8)

These cloudy pixels are not really usefull to us, therefore we will exclude them from the RasterBrick. But first, we do not need that band 9 anymore, so let's drop it from the brick. We use the `dropLayer()` function for that, which returns a RasterStack object.

```
> taravao8 <- dropLayer(x=taravao, i=9)
> # Let's duplicate that RasterStack and save it for later
> taravao8_2 <- taravao8
```

Excluding data from a raster object using a mask can be done either by directly performing vector replacement operations, or with the `calc()` function. The same considerations of data size and RAM management than above apply when selecting one method or the other. For the purpose of the example, the two methods are illustrated below. The dataset is small enough to be safely handled in memory. What we want to do is replace the cloudy pixels by a NA.

```
> taravao8[cloud == 1] <- NA
```

```
> cloud2NA <- function(x, y){
+   x[y == 1] <- NA
+   return(x)
+ }
> taravao8_3 <- overlay(taravao8_2, cloud, fun=cloud2NA)
```

7 Hdf files (Not part of the course)

A file format that is getting increasingly common with geo-spatial gridded data is the hdf format. Hdf stands for hierarchical data format. For instance MODIS data have been delivered in hdf format since its beginning, Landsat has adopted the hdf format for delivering its Level 2 surface reflectance data recently. This data format has an architecture that makes it very convenient to store data of different kind in one file, but requires slightly more effort from the scientist to work with conveniently.

Windows

The rgdal package does not include hdf drivers in its pre-built binary. Data can therefore not be read directly from hdf into R (as object of class rasterLayer). A workaround is to convert the files externally to a different data format. Since you probably have gdal installed on your system (either via FWTools or OSGeo4W), you can use the command line utility gdal_translate to perform this operation. One easy way to do that is by calling it directly from R, via the command line utility.

```
> library(MODIS)
> # Get a list of sds names
> sds <- getSds('full/path/filename.hdf')
> # Isolate the name of the first sds
> name <- sds$SDS4gdal[1]
> # Optional: In case a sds name contains spaces, these 2 steps remove them
> name <- gsub("\"", "", name)
> name <- sprintf('\"%s\"', name)
> # Define output filename and create gdal_translate command
> filename <- 'name/of/output/file.tif'
> translate <- sprintf('gdal_translate %s %s', name, filename)
> system(translate)
> # Load the Geotiff created into R
> r <- raster(filename)
```

Linux

```
> library(MODIS)
> # Get a list of sds names
> sds <- getSds('full/path/filename.hdf')
> # Any sds can then be read directly using the raster function
> r <- raster(sds$SDS4gdal[1])
```

8 Further reading

9 Packages you should know about

The newly released gdalUtils package should provide interesting wrappers facilitating the use of gdal functions from within R.

10 Exercise

Context

Idea of exercise, using a second image of taravao, that I will provide, taken at a different date, also cloudy, write a compositing function. Taravao2 is a Landsat 8 image covering the same area than taravao, but acquired at a different date. We saw earlier that taravao has residual clouds; the same is true for taravao2. However, since the clouds are not at the same location on the two images, it should be possible to create a composite image, maximizing the number of observations.

Expected output

Using the two taravao Landsat 8 images as example datasets (taravao2 can be loaded using the `data()` command), write a mean value compositing function. The function should accepts 2 typical Landsat 8 `RasterBrick` or `RasterStack` objects - containing 9 bands and band 9 being the QA layer - and return an 8 layers `RasterStack` or `RasterBrick` object. In case of cloud occurence at both dates, pixel value should be NA, if both dates are cloud free, the mean value will be returned.

Some pointers (??)

- Prepare the datasets (generate the two cloud masks, get rid of QA layer)
- Write a simple function that works for 4 input numeric arguments
- Vectorize the function using mapply
- Use overlay to run the vectorized function, with 4 arguments

The first chunk below is how I got to the solution. The second chunk is the solution (a function that takes 2 rasterBricks as input and that returns one rasterStack/brick). Another solution (probably easier actually) is to first apply the cloud mask to the bricks and average them controling well the na.rm argument.

```
> # Download and unpack
>
> # Read data taravao
> data(taravao)
> # taravao2 <- taravao
> taravao2 <- brick('filename2.tif') # that does not exists yet, use line above for tests
> taravao2 <- crop(x=taravao2, y=taravao)
> cloud <- calc(taravao[[9]], fun=QA2cloud)
> cloud2 <- calc(taravao2[[9]], fun=QA2cloud)
> taravao <- dropLayer(x=taravao, i=9)
> taravao2 <- dropLayer(x=taravao2, i=9)
> # Write function for single values
> compose <- function(c1, c2, b1, b2){
+   if(c1 == 1 & c2 == 1) {
+     x <- NA
+   } else if (c1 == 0 & c2 == 0) {
+     x <- mean(c(b1, b2))
+   } else if (c1 == 1 & c2 == 0) {
+     x <- b2
+   } else if (c1 == 0 & c2 == 1) {
+     x <- b1
+   }
+   return(x)
+ }
> composeV <- function(c1, c2, b1, b2) {
+   v <- mapply(FUN=compose, c1, c2, b1, b2)
+   return(v)
+ }
> taravaoComposit <- overlay(cloud, cloud2, taravao, taravao2, fun=composeV)
```

If wrapped into a function taking 2 rasterBricks as input

```
> # Vectorize the function
> Landsat8Compose <- function(x, y) { # x and y are 2 RasterBricks
+   c1 <- calc(x[[9]], fun=QA2cloud)
+   c2 <- calc(y[[9]], fun=QA2cloud)
+   x <- dropLayer(x=taravao, i=9)
+   y <- dropLayer(x=taravao2, i=9)
+
+   compose <- function(c1, c2, b1, b2){
+     if(c1 == 1 & c2 == 1) {
+       x <- NA
+     } else if (c1 == 0 & c2 == 0) {
+       x <- mean(c(b1, b2))
+     } else if (c1 == 1 & c2 == 0) {
+       x <- b2
+     } else if (c1 == 0 & c2 == 1) {
+       x <- b1
```

```
+     }
+     return(x)
+   }
+
+   composeV <- function(a, b, c, d) {
+     v <- mapply(FUN=compose, a, b, c, d)
+     return(v)
+   }
+
+   out <- overlay(c1, c2, x, y, fun=composeV)
+   return(out)
+ }
> # Let's test
> data(taravao)
> taravao2 <- taravao
> taravaoC <- Landsat8Compose(taravao, taravao2)
```