Chapter 1: GDAL

Chris Holden

03/24/2015

Geographic Data Abstraction Library

From http://www.gdal.org/:

> GDAL is a translator library for raster and vector geospatial data formats that is released under an X/MIT style Open Source license by the Open Source Geospatial Foundation. As a library, it presents a single raster abstract data model and vector abstract data model to the calling application for all supported formats. It also comes with a variety of useful commandline utilities for data translation and processing.

Basically, GDAL is your best friend for working with any type of raster or vector data. You have two avenues for working with GDAL: the Application Programming Interface (API) or the command line utilities (CLI).

The GDAL API is a library of classes and functions for dealing with raster and vector data. The API itself is written in C/C++, but bindings are available to the API for many languages, including Python, Ruby, Java, and Perl. These classes and functions allow you to write your own custom applications that leverage the pre-existing GDAL code for all details involved in input and output of spatial data, as well as other functions including reprojection, resampling, stretching, and more.

The GDAL command line utilities are a suite of programs written in C/C++ or Python by the GDAL authors that provide an enormous amount of functionality right at your terminal. From querying for metadata information of a single image to resampling, resizing, subsetting, clipping, and reprojecting an image all in one command, you can accomplish virtually all of your preprocessing workflows with the simple combination of GDAL's command line utilities and the Bash shell or similar. Some frequently used GDAL command line utilities include:

- gdalinfo
  - raster data data and metadata querying
- gdal_translate
  - raster data format conversion, datatype conversion, clipping, and scaling
- gdalwarp
  - raster data reprojection, resampling, resizing, clipping by extent, clipping by polygons, and mosaicing
- gdal_merge.py
  - raster data mosaicing and layer stacking
- gdal_calc.py
  - raster data map algebra at the command line
- ogrinfo
  - vector data data and metadata querying, including SQL syntax queries on attribute tables
- ogr2ogr
  - vector data format conversion, merging, subsetting, reprojection, clipping, and more

A complete listing of utilities can be found here for raster data or here for vector data.

Raster data in R

The basic package we will be using within R is the `raster` package. The `raster` package by itself can read a few raster formats, but when combined with `rgdal` it can read virtually every raster format in existence. For a more thorough description, visit the following CRAN webpages:

- CRAN
- Reference Manual
- Vignette: Introduction to the raster package
- Vignette: Writing functions for large raster files
- Vignette: Description of the raster file format

```
library(raster)
```

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

Objects

There are three ways to represent raster data within `raster`:

- `raster`: a single layer raster object
- `stack`: a multiple layer raster object
- `brick`: a multiple layer raster object

What are the differences between `stack` and `brick`?

- `brick` is less flexible
  - all layers must be from the same file
- `brick` is faster than `stack` if restrictions are acceptable

Input/Output

When data are opened in R using `raster`, the image data itself is not immediately read into memory. Instead, the `raster` package only reads in image metadata and delays reading of actual image data until raster cell values are specifically accessed. The largest benefit to this approach is that very large rasters can be processed using small amounts of memory by operating in small chunks or tiles.

To access pixel values from raster data containers (`stack`, `brick`, etc.) with the `raster` package, use the `getValues` and `extract` functions. Assignment of raster values can be performed using the `setValues` and `replacement` functions.

Math

Arithmetic can be performed on `raster` package `Raster*` objects as if they were simple integers or floats. These simple arithmetic commands, however, do not try to be cautious of memory limitations and may fail when performing large calculations.

```r
# From `raster:Arith-methods` help
r1 <- raster(ncols=5, nrows=5)
r1[] <- runif(ncell(r1))
r2 <- setValues(r1, 1:ncell(r1) / ncell(r1) )
r3 <- r1 + r2
r2 <- r1 / 10
r3 <- r1 * (r2 - 1 + r1^2 / r2)
r3
```

```
## class       : RasterLayer
## dimensions  : 5, 5, 25  (nrow, ncol, ncell)
## resolution  : 72, 36  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84
## data source : in memory
## names       : layer
## values      : -0.02233212, 9.028224  (min, max)
```

To perform memory safe calculations on large rasters by operating in chunks, utilize either the `calc` or `overlay` functions.

```r
# From `raster:calc` help
r <- raster(ncols=36, nrows=18)
r[] <- 1:ncell(r)

# multiply values with 10
fun <- function(x) { x * 10 }
rc1 <- calc(r, fun)
rc1
```

```
## class       : RasterLayer
## dimensions  : 18, 36, 648  (nrow, ncol, ncell)
## resolution  : 10, 10  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84
## data source : in memory
## names       : layer
## values      : 10, 6480  (min, max)
```

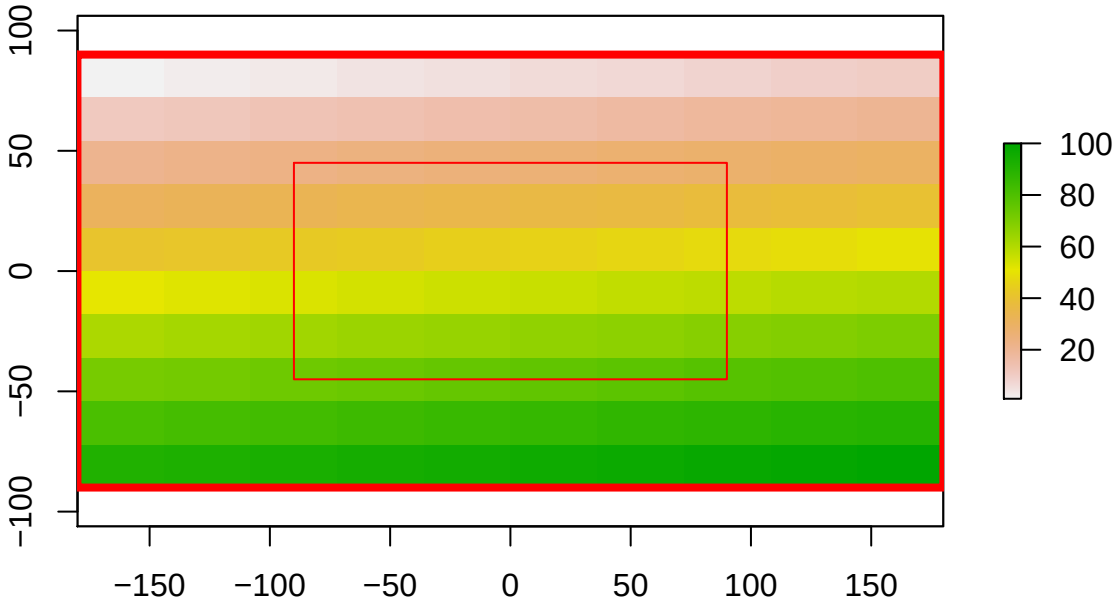
Visualize

You can visualize `Raster*` objects within R by using either the `plot` or `plotRGB`.

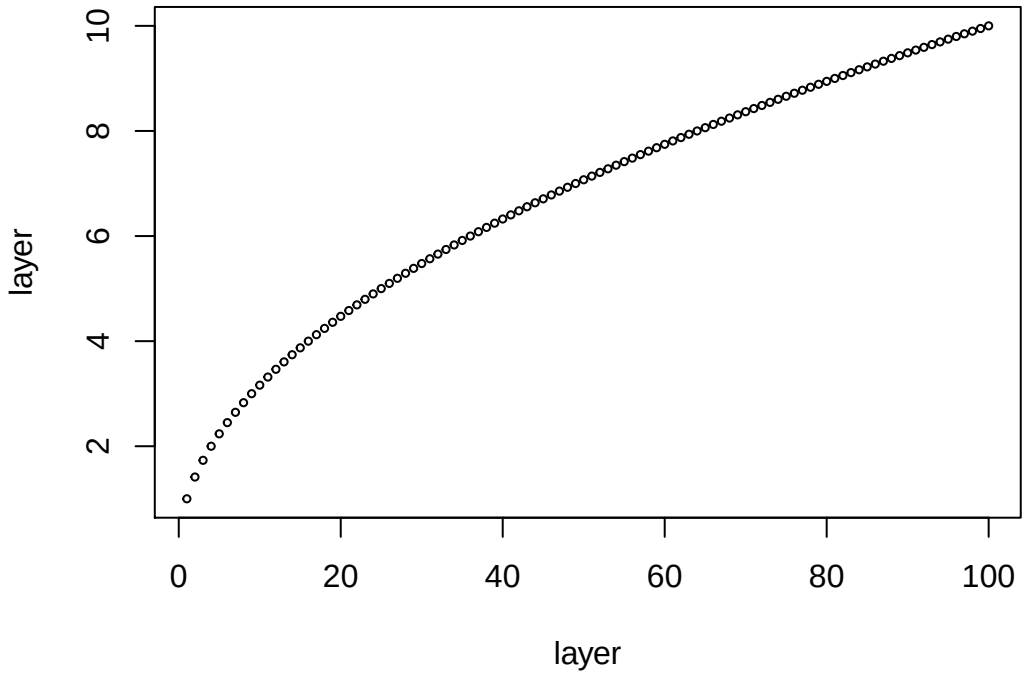
The `plot` function can either make an image of a `Raster*` object, or create a scatterplot of the `Raster*` object values.

```r
# From `raster:plot` help
# RasterLayer
r <- raster(nrows=10, ncols=10)
r <- setValues(r, 1:ncell(r))
plot(r)
```

```
e <- extent(r)
plot(e, add=TRUE, col='red', lwd=4)
e <- e / 2
plot(e, add=TRUE, col='red')
```




```
# Scatterplot of 2 RasterLayers
r2 <- sqrt(r)
plot(r, r2)
```



The `plotRGB` function allows for three channel image representations (raster bands mapped to RGB channels). The function also can assist display by providing image enhancing transforms such as stretching of image values or pixel interpolation.

```r
# From `raster:plotRGB` help
b <- brick(system.file("external/rlogo.grd", package="raster"))
plotRGB(b)
```

```r
plotRGB(b, 3, 2, 1)
```

Example

For a more interesting example, we'll download a subset of a Landsat 7 image from an area in Chiapas, Mexico from the tutorial Github repository.

```
download.file(url='https://raw.githubusercontent.com/ceholden/open-geo-tutorial/master/example/LE702049
              destfile='LE70220492002106EDC00_stack.gtif', method='curl')
```

Since this is a multispectral image, we'll use the `brick` function to load it:

```
le7 <- brick('LE70220492002106EDC00_stack.gtif')
le7
```

```
## class       : RasterBrick
## dimensions  : 250, 250, 62500, 8  (nrow, ncol, ncell, nlayers)
## resolution  : 30, 30  (x, y)
## extent      : 462405, 469905, 1734315, 1741815  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=utm +zone=15 +datum=WGS84 +units=m +no_defs
## data source : /home/ceholden/Documents/open-geo-tutorial/R/LE70220492002106EDC00_stack.gtif
## names       : band.1.reflectance, band.2.reflectance, band.3.reflectance, band.4.reflectance, band.5
## min values  :             -32768,             -32768,             -32768,             -32768,
## max values  :              32767,              32767,              32767,              32767,
```

Notice the attributes of the `RasterBrick` class, including the pixel dimensions of the image (nrow, ncol, nlayers), the spatial resolution, the geographic transform (extent), and the coordinate reference system (projection).

We can refer to the bands within this `RasterBrick` either by the band index or by the layer names (when defined):

```
all.equal(le7[[3]], le7$band.3.reflectance)
```

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

Now, plot it in 5-4-3:

```
plotRGB(le7, r=5, g=4, b=3, stretch="lin")
```