Spatial Download Script - SNODAS Raster

Josh Nowak

September 29, 2016

### This is a markdown document

[Markdown](https://daringfireball.net/projects/markdown/) is a simple text to html conversion tool. This tool is not R specific but was made availble to R users through the work of [Yihui Xie](http://yihui.name/), the good folks at [RStudio](http://rmarkdown.rstudio.com/) and many others. Markdown documents are simple by design. You can alter text by writing in *italics*, **bold**, superscript2 and using html headers or directly inserting html.

This is html...it is only scary because it's new =)-

Because markdown renders html you can use all of the [HTML Widgets](http://www.htmlwidgets.org/) or any other package or tool that renders HTML. These widgets can make your plots interactive and generally enhance your ability to convey information while making your documents pretty.

A very relevant feature of markdown is how we insert code into these documents. We have two options, we can insert code inline 2 + 5 or write it in blocks

foo <- 2 + 5  
bar <- 3  
foo/bar

We always have the option to just show code (like we did above) or evaluate code. For example, the sum of 2 and 5 is 7 or even

2 + 5

## [1] 7

Another thing you should know is that this type of document can be converted to several other formats, made into a presentation and more...

* Documents
  + PDF
  + Word

1. Presentations
   * Beamer
   * Slidify
   * HTML5 Slides
     + Yeah, no Microsoft
2. Other
   * Websites (like this GitHub page)
   * Dashboards
   * Books
   * Scientific articles
   * Shiny applications
   * And more...

That is it for this quick introduction. If you need more help you can visit the RStudio site linked above or get the cheat sheet from [this link](https://www.rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf). Let's move on to the purpose of this document.

### Spatial data download

**Know thy data**

Before you work with any data you must understand it. The SNODAS data that we are working with here is defined at [this url](http://nsidc.org/data/docs/noaa/g02158_snodas_snow_cover_model/#DailyTable). You can follow along with the code below, but to truly understand why each step was done and in part how, you will need to read that webpage.

In order to download data we first need to know where the data is stored. The online address of this storage location can be a URL, FTP or even an API. Today we will work with an FTP site. I should also note that we will be doing things *manually* to maximize the transparency of the steps, but know that it can all be automated. I will always suggest that small functions be written for each step and then strung together to create a workflow.

SNODAS data is hosted on a ftp site located at <ftp://sidads.colorado.edu/DATASETS/NOAA/G02158/unmasked/>. The same can be said for MODIS, <https://lpdaac.usgs.gov/data_access/data_pool>. MODIS even has an API (based on SOAP) and another webservice to return values at points. Two R packages make it easy to interact with these data. Let's proceed with the SNODAS data.

#### Create a directory to store the data

First, we will load the packages we need to run the rest of the code. Two of my favorite spatial packages are rgdal and raster. We will use these two packages to work with points, lines, polygons, rasters and hybrid data types.

require(leaflet)  
 require(rgdal)  
 require(raster)

Second, create a directory where we can store the data we download. I also included a quick test of whether the directory exists and then verification that it was created. The results of each step will be printed below the code.

# Create an object pointing to your new directory  
 my\_dir <- "C:/sp\_example"   
  
 # Check if our directory exists - useful when we want to control  
 # overwriting  
 dir.exists(my\_dir)

## [1] TRUE

# Create a directory - Windows specific directory location, change for other # OS ( e.g. dir.create("~/etc/sp\_example") )  
 # This could also be dependent on whether the directory exists using an if  
 # statement or similar flow control  
 dir.create(my\_dir)

## Warning in dir.create(my\_dir): 'C:\sp\_example' already exists

# Check if our directory exists  
 dir.exists(my\_dir)

## [1] TRUE

#### Download the data

The data are stored in a hierarchical folder structure that is convenient for generating names on the fly. The base url is:

* <ftp://sidads.colorado.edu/DATASETS/NOAA/G02158/unmasked/>

The next level of directories show us the year. I chose 2015, so the new url looks like:

* <ftp://sidads.colorado.edu/DATASETS/NOAA/G02158/unmasked/2015/>

The next level is the month. I chose January, but notice the format is a two digit number for the month followed by an underscore and then the three letter abbreviation for the month.

* <ftp://sidads.colorado.edu/DATASETS/NOAA/G02158/unmasked/2015/01_Jan/>

Now we can see the actual data files for each day of the month. The naming convention is perfectly consistent. The name consists of *SNODAS\_unmasked\_* followed by the four digit year, 2 digit month and the 2 digit day of the month. Our final url looks like:

* <ftp://sidads.colorado.edu/DATASETS/NOAA/G02158/unmasked/2015/01_Jan/SNODAS_unmasked_20150102.tar>

Note that we should think about how to code this without having to write out all the filenames. We can start thinking about how this workflow might be put into functions by specifying some of the arguments we will need later. One technique that might help us think about this is to decompose the folder structure and file names into their pieces. In this example we have year, month and day. Those are all parts of a date, so maybe we should start with a date and a base url, we can modify as needed later.

# Date of interest - January 02, 2015  
 dt <- as.Date("2015-01-02")  
  
 # Base url where files are stored  
 base\_url <- "ftp://sidads.colorado.edu/DATASETS/NOAA/G02158/unmasked"

With the date defined we can now create the file name and download the data.

# Create file name separate from url so we can use it to name things we save  
 fname <- paste0("SNODAS\_unmasked\_", format(dt, "%Y%m%d"), ".tar")  
 #print(fname)  
  
 # Create full url - dynamic, not having hard coded values is a good thing  
 full\_url <- paste(  
 base\_url,  
 format(dt, "%Y"),  
 paste(format(dt, "%m"), format(dt, "%b"), sep = "\_"),  
 fname,  
 sep = "/"  
 )  
  
 # Print full url so you can see what it looks like  
 full\_url

## [1] "ftp://sidads.colorado.edu/DATASETS/NOAA/G02158/unmasked/2015/01\_Jan/SNODAS\_unmasked\_20150102.tar"

# Download file from ftp site and save it to the directory we created above  
 # A call to system.time is used to show the reader how long operations took  
 # to complete  
 system.time(  
 download.file(  
 full\_url,  
 mode = "wb",  
 destfile = file.path(my\_dir, fname)  
 )  
 )

## user system elapsed   
## 0.59 0.81 21.31

With our file downloaded we now face the problem of dealing with a compressed file. Luckily we have several tools for working with compressed files. The original file is a .tar file and the unpacking function is called untar.

# Untar the file to my\_dir defined above  
 system.time(  
 untar(file.path(my\_dir, fname), exdir = my\_dir)  
 )

## user system elapsed   
## 0.00 0.02 0.09

Now we should take a look at what that did. Since we saved the extracted files to *my\_dir* we can simply list those files.

list.files(my\_dir)

## [1] "SNODAS\_unmasked\_20150102.tar"   
## [2] "states"   
## [3] "states.zip"   
## [4] "zz\_ssmv01025SlL00T0024TTNATS2015010205DP001.dat.gz"  
## [5] "zz\_ssmv01025SlL00T0024TTNATS2015010205DP001.Hdr.gz"  
## [6] "zz\_ssmv01025SlL01T0024TTNATS2015010205DP001.dat.gz"  
## [7] "zz\_ssmv01025SlL01T0024TTNATS2015010205DP001.Hdr.gz"  
## [8] "zz\_ssmv11034tS\_\_T0001TTNATS2015010205HP001.dat.gz"   
## [9] "zz\_ssmv11034tS\_\_T0001TTNATS2015010205HP001.Hdr.gz"   
## [10] "zz\_ssmv11036tS\_\_T0001TTNATS2015010205HP001.dat.gz"   
## [11] "zz\_ssmv11036tS\_\_T0001TTNATS2015010205HP001.Hdr.gz"   
## [12] "zz\_ssmv11038wS\_\_A0024TTNATS2015010205DP001.dat.gz"   
## [13] "zz\_ssmv11038wS\_\_A0024TTNATS2015010205DP001.Hdr.gz"   
## [14] "zz\_ssmv11039lL00T0024TTNATS2015010205DP000.dat.gz"   
## [15] "zz\_ssmv11039lL00T0024TTNATS2015010205DP000.Hdr.gz"   
## [16] "zz\_ssmv11044bS\_\_T0024TTNATS2015010205DP000.dat.gz"   
## [17] "zz\_ssmv11044bS\_\_T0024TTNATS2015010205DP000.Hdr.gz"   
## [18] "zz\_ssmv11050lL00T0024TTNATS2015010205DP000.dat.gz"   
## [19] "zz\_ssmv11050lL00T0024TTNATS2015010205DP000.Hdr.gz"

Ahhh, we created a bunch of files. Looking at the documentation for the data we see that the naming convention for the snow depth data includes *ssmv11036tS*. We can subset the list to that file, but then we also have to deal with the .gz file type. Not unlike .tar there is a nice function that helps us deal with that.

# Start by subsetting our list of files to the desired file  
 snow\_depth\_file <- list.files(my\_dir, pattern = ".\*ssmv11036tS.\*.dat.gz")  
  
 # Now that we have the file name we want we need to convert the .gz to a   
 # more friendly format.   
 # Establish a connection to the file  
 depth\_con <- gzcon(file(file.path(my\_dir, snow\_depth\_file), "rb"))  
   
 # Once the connection is made we can read the binary data.  
 # We found out the dimensions and characteristics of the data by reading the  
 # documentation.  
 system.time(  
 depth\_unscaled <- readBin(  
 depth\_con,   
 integer(),   
 n = 8192 \* 4096,   
 size = 2,  
 signed = TRUE,   
 endian = "big"  
 )  
 )

## user system elapsed   
## 1.58 0.03 1.63

# The data are scaled, so we need to divide by 1,000 per the documentation  
 depth <- depth\_unscaled/1000  
   
 # Set -9.999 to NA  
 depth[depth < 0] <- NA  
   
 # These objects are big, so close connection and remove unscaled data from   
 # memory  
 close(depth\_con);rm(depth\_unscaled);gc()

## used (Mb) gc trigger (Mb) max used (Mb)  
## Ncells 1013173 54.2 1770749 94.6 1442291 77.1  
## Vcells 34564605 263.8 98122173 748.7 101673206 775.8

Our data are now a vector, but we need a flat data format, something like a spreadsheet, with the number of columns and rows that the SNODAS website told us it should have. Recall that we should have 8,192 columns and 4,096 rows.

# Ok, we have a large numeric vector and we need something square, use a  
 # matrix for the sake of memory efficiency  
 snow\_mat <- matrix(depth, nrow = 4096, ncol = 8192, byrow = T)

This is about spatial data right? Finally, we can create a raster using our data and the wonderful [raster package](https://cran.r-project.org/web/packages/raster/raster.pdf). An important resource when working with spatial data is <http://spatialreference.org/>. This is where you can find all the projection definitions that you will ever need. In this case our documentation tells us that the projection is Geographic (lat/long) and the datum is WGS84. EPSG 4326 should be a good choice for this data. It is defined by the following proj4 string *+proj=longlat +ellps=WGS84 +datum=WGS84 +no\_defs*.

# Create a raster form the matrix by specifying the projection and the  
 # extent of the raster (xmn = minimum x value, xmx = maximum x value...)  
 # We can find the projection defintion, xmin, xmax, etc. on the data's  
 # documentation site linked above  
 r <- raster(  
 snow\_mat,   
 crs = "+proj=longlat +ellps=WGS84 +datum=WGS84 +no\_defs",  
 xmn = -130.5171,  
 xmx = -62.2504,  
 ymn = 24.0996,  
 ymx = 58.2329  
 )  
   
 # Show print method for rasters  
 r

## class : RasterLayer   
## dimensions : 4096, 8192, 33554432 (nrow, ncol, ncell)  
## resolution : 0.008333337, 0.008333325 (x, y)  
## extent : -130.5171, -62.2504, 24.0996, 58.2329 (xmin, xmax, ymin, ymax)  
## coord. ref. : +proj=longlat +ellps=WGS84 +datum=WGS84 +no\_defs +towgs84=0,0,0   
## data source : in memory  
## names : layer   
## values : 0, 32.767 (min, max)

Finally some real spatial data. But of course data never arrives in the right format, projection, etc. In this case we have a raster of the entire United States, but maybe we only want to work with data from Idaho. Now we will crop the raster to the state of ID. To do this we will need to download and process a shapefile of Idaho. We will download the zip directory containing the shapefile from census.gov.

# URL of shapefile from census.gov  
 states\_url <- "http://www2.census.gov/geo/tiger/GENZ2015/shp/cb\_2015\_us\_state\_500k.zip"  
  
 # Download file  
 system.time(  
 download.file(  
 states\_url,   
 mode = "wb",   
 destfile = file.path(my\_dir, "states.zip")  
 )  
 )

## user system elapsed   
## 0.04 0.02 0.37

The data are now downloaded and in *my\_dir*. From there we need to decompress the folder into files and then they will be useful. This time we will use unzip to extract the files.

# Unzip folder  
 unzip(  
 zipfile = file.path(my\_dir, "states.zip"),  
 exdir = file.path(my\_dir, "states")  
 )  
  
 # Show files in newly created folder  
 list.files(file.path(my\_dir, "states"))

## [1] "cb\_2015\_us\_state\_500k.cpg"   
## [2] "cb\_2015\_us\_state\_500k.dbf"   
## [3] "cb\_2015\_us\_state\_500k.prj"   
## [4] "cb\_2015\_us\_state\_500k.shp"   
## [5] "cb\_2015\_us\_state\_500k.shp.ea.iso.xml"  
## [6] "cb\_2015\_us\_state\_500k.shp.iso.xml"   
## [7] "cb\_2015\_us\_state\_500k.shp.xml"   
## [8] "cb\_2015\_us\_state\_500k.shx"

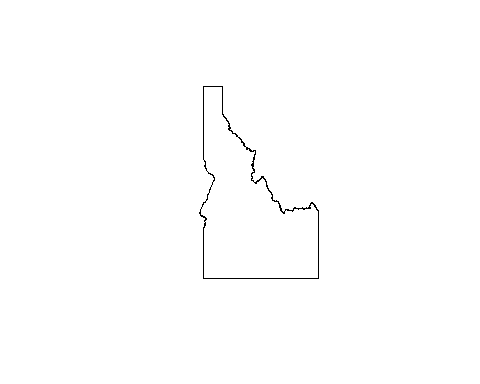
With the unzipping complete we can move on to using the data. A nice function for reading polygon, point and line data types is readOGR from the rgdal package. We specify the folder containing the data as the first argument and then the name of the layer without a file extension as the second argument. Note the data came to us a NAD83 based projection and we need the WGS84 type projection from above.

# Read in shapefile  
 system.time(  
 states\_NAD83 <- readOGR(  
 dsn = file.path(my\_dir, "states"),   
 layer = "cb\_2015\_us\_state\_500k"  
 )  
 )

## OGR data source with driver: ESRI Shapefile   
## Source: "C:/sp\_example/states", layer: "cb\_2015\_us\_state\_500k"  
## with 56 features  
## It has 9 fields

## user system elapsed   
## 0.37 0.03 0.41

# Subset to Idaho  
 id\_NAD83 <- states\_NAD83[states\_NAD83$NAME == "Idaho",]  
   
 # Same thing, but using subset  
 #id\_NAD83 <- subset(states\_NAD83, NAME == "Idaho")  
   
 # Leaflet plot of outline of Idaho...had it to make it less boring  
 # When knitting to Word interactive elements don't work, so replace the   
 # code below with a base plot  
# leaflet(id\_NAD83) %>%  
# addPolygons(stroke = T, fillOpacity = 0) %>%  
# addProviderTiles("OpenStreetMap.Mapnik",   
# group = "Street Map") %>%  
# addProviderTiles("OpenTopoMap", group = "Topo") %>%  
# addProviderTiles("Esri.WorldImagery", group = "Aerial") %>%  
# addProviderTiles("NASAGIBS.ViirsEarthAtNight2012",   
# group = "Night Time") %>%  
# addLayersControl(  
# baseGroups = c("Street Map (default)", "Topo", "Aerial", "Night Time"))  
 plot(id\_NAD83)



We have now read in the shapefile, which in R is a SpatialPolygonsDataFrame. We also subset polygon data of the US to just the state of Idaho. The problem now is that our raster data (SNODAS) is in a different projection than the polygon of the state. Before we can crop the raster with the state polygon we need to reproject one of the two data types. In general, we want to project non-raster data to the projection of the raster data for speed. That said, we usually choose projections for a reason and should consider the tradeoffs inherent to such choices. If you need to project a large raster I highly recommend looking at the [gdalUtils package](https://cran.r-project.org/web/packages/gdalUtils/index.html) package. There is a function to reproject rasters in the raster package too, but I have had better success with gdal based tools. Anyway, let's get on with reprojecting our spatial polygon of Idaho.

# Reproject to WGS84 - take projection definition from raster  
 system.time(  
 id <- spTransform(id\_NAD83, projection(r))  
 )

## user system elapsed   
## 0.02 0.00 0.02

# Print id to show projection and the like  
 id

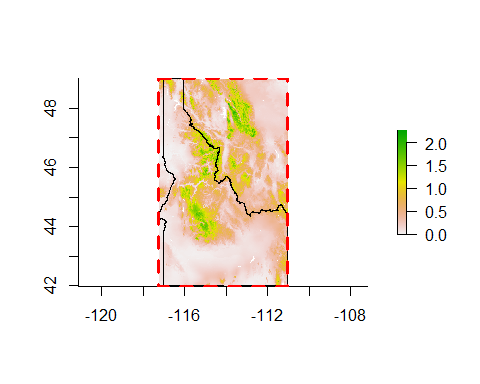
## class : SpatialPolygonsDataFrame   
## features : 1   
## extent : -117.243, -111.0436, 41.98806, 49.00115 (xmin, xmax, ymin, ymax)  
## coord. ref. : +proj=longlat +ellps=WGS84 +datum=WGS84 +no\_defs +towgs84=0,0,0   
## variables : 9  
## names : STATEFP, STATENS, AFFGEOID, GEOID, STUSPS, NAME, LSAD, ALAND, AWATER   
## min values : 16, 01779783, 0400000US16, 16, ID, Idaho, 00, 2147483647, 2147483647   
## max values : 16, 01779783, 0400000US16, 16, ID, Idaho, 00, 2147483647, 2147483647

With the polygon of Idaho reprojected we can crop the raster to the shapefile. This will return a large square raster that is guaranteed to cover Idaho. It also dramatically reduces the amount of data.

# Crop raster to shapefile  
 system.time(  
 id\_snodas <- crop(r, id)  
 )

## user system elapsed   
## 0.03 0.00 0.03

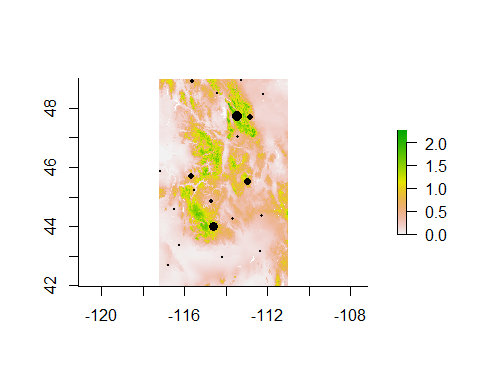
# Plot the raster under the polygon of the state to show the relationship  
 par(bty = "l")  
 plot(id\_snodas)  
 lines(id)  
 plot(extent(id), add = T, lty = 2, lwd = 3, col = "red")



The plot above depicts the cropped raster of snow depth data, the outline of the state of Idaho and the extent of two objects in red.

Now id\_snoads is a square raster, as they all are, covering the state of Idaho. One common task might be to extract the raster values at certain points in space. In the next step we create some fake points and then sample the raster at those points.

# Extract raster values at 20 fake xy coordinates, store the coordinates,   
 # unique id and the snodas value in a data.frame  
 # Extent of id\_snodas, this is the something like the smallest box the   
 # shape can fit inside   
 ext <- extent(id\_snodas)  
   
 # Create fake point data, maybe study sites  
 fake\_xy <- cbind(  
 x = runif(20, ext@xmin, ext@xmax),   
 y = runif(20, ext@ymin, ext@ymax)  
 )  
   
 # Organize output in a data.frame, notice the call to extract on the last  
 # line. This is the function from the raster package that extracts  
 # values.  
 snodas\_vals <- data.frame(  
 pt\_id = 1:length(fake\_xy),  
 x = fake\_xy[,1],  
 y = fake\_xy[,2],  
 snodas = extract(id\_snodas, fake\_xy)  
 )  
   
 # Plot the output  
 par(bty = "l")  
 plot(id\_snodas)  
 points(  
 snodas\_vals[,c("x", "y")],   
 pch = 19,  
 cex = snodas\_vals$snodas  
 )

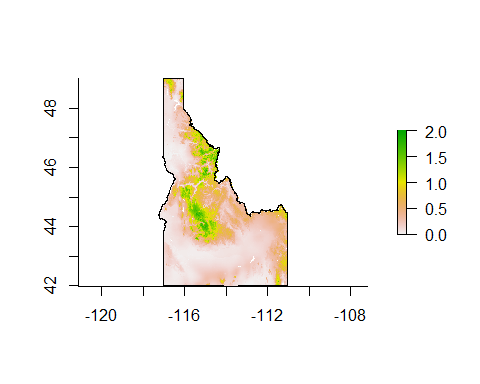


head(snodas\_vals)

## pt\_id x y snodas  
## 1 1 -114.4016 48.49457 0.173  
## 2 2 -115.5051 45.23270 0.254  
## 3 3 -112.8387 47.69132 0.900  
## 4 4 -115.6315 48.90207 0.531  
## 5 5 -112.2719 44.37552 0.367  
## 6 6 -112.2195 48.47270 0.207

Sampled SNODAS locations shown by the black dots where the size of the dot represents the amount of snow at that point in meters. The table below the plot shows the first few lines of the data that was created.

# Sometimes for the sake of presentation we want to create a raster   
 # that only has values within the polygon of interest. Here we "cut out"  
 # Idaho ignoring all values that are not within the state boundary  
 snodas <- mask(id\_snodas, id)  
 par(bty = "l")  
 plot(snodas)  
 lines(id)



A masked plot of the SNODAS raster showing only those values within the state of Idaho.

### Workflow

Here we have shown how to

* Download raster data
* Untar
* Read binary data stored in a .gz file type
* Transform vector into a raster
* Download polygon data in the form of a compressed shapefile
* Unzip
* Read a shapefile into R
* Subset Spatial\* object
* Crop raster data
* Extract values at points from raster
* Mask raster for the sake of presentation

#### Closing Thoughts

R can perform many spatial operations, but it was not created for this purpose and at times may not be optimized for the task at hand. A reason I use R for spatial manipulations is the consistency of my workflow. Not having to change languages or tool sets midstream is a big advantage to me and if the cost is not too high then I am happy. That said, if a more optimized solution to a problem is desired you might consider QGIS, GRASS or many of the other tools with which R can communicate. This type of workflow may offer the best of both worlds, but in my work I don't find much need to improve on R's functionality.

Why program when you can click buttons? One reason I much prefer programming is that I always have a record of what was done at each step. Another reason I prefer programming is that for operations that take an excessive amount of time I do not need to be present to click the next button. And last, while difficult at times I prefer to not be restricted to "whatever the developer of the software thought I should do". What I mean by that is that within a programming language I am free to develop whatever method I want or need and am not restricted to what the software can do.