Raster Analysis

Ben DeVries, Jan Verbesselt, Loïc Dutrieux, Sytze de Bruin

October 15, 2013

Abstract

In this tutorial, we will explore the raster package and other related packages used for typical raster analyses. We will first look at analysis of RasterLayer objects, exploring functions having to do with raster algebra, focal and zonal statistics and other operations. We will then explore spatio-temporal analysis of raster data using RasterBrick objects. Here, we will extract time series data from a RasterBrick object and derive temporal statistics from these data. Since data from the Landsat archive is becoming increasingly important for environmental research and monitoring, this tutorial will focus on the use of these data.

1. perform typical image preprocessing operations (apply a mask, calculate ndvi, etc.)
2. classify a raster layer
3. overlay with geodata and calculate zonal statistics
4. perform focal operations
5. parse Landsat scene information in a time series
6. explore a raster brick by plotting layers and layers statistics
7. perform raster brick operations to derive statistics (e.g. %no-data in the time series)
8. extract pixel time series and derive various time series statistics

1 The raster package

The raster package is an essential tool for raster-based analysis in R. Here you will find functions which are fundamental to image analysis. The raster package documentation is a good place to begin exploring the possibilities of image analysis within R. There is also an excellent vignette available at `http://cran.r-project.org/web/packages/raster/vignettes/Raster.pdf`.

2 The Landsat archive

Since being released to the public, the Landsat data archive has become an invaluable tool for environmental monitoring. With a historical archive reaching back to the 1970's, the release of these data has resulted in a spur of time series based methods. In this tutorial, we will work with time series data from the Landsat 7 Enhanced Thematic Mapper (ETM+) sensor.

3 Manipulating raster data

This section will cover some of the following areas:

- calculate % of no data in a raster
- crop a raster based on a defined extent, another object's extent, or interactively using drawExtent()
- working with the calc() and overlay() functions:
  - load and apply a mask (cloud mask)
  - raster algebra with two or more raster layers
- focal operations; create a function to 'sieve' a raster object (remove lone pixels)
- create an areal filter for a raster by converting to a SpatialPolygon, removing small features and back to raster using the rasterize() function
- zonal statistics using other geodata (Biosphere Reserve zones, administrative zones, etc...)
- reclassifying (stratifying) a raster based on elevation data (SRTM or ASTER2)

Raster classification

....using the reclassify() function
....section on Random Forest classification, using the Gewata or Tura subset...

Working with thematic rasters

In some cases, the values of a raster may be categorical, meaning they relate to a thematic class (e.g. 'forest' or 'wetland') rather than a quantitative value (e.g. NDVI or % Tree Cover). The raster dataset 'LULC2011_Gewata_classes.tif' is a raster with integer values representing LULC classes from a 2011 classification (using SPOT5 and ASTER source data).

```
> lulc <- raster('path/to/data/LULC2011_Gewata.tif')

> # check out the distribution values
> freq(lulc)

     value  count
[1,]     1 396838
[2,]     2  17301
[3,]     3    943
[4,]     4  13645
[5,]     5 470859
[6,]     6 104616
[7,]    NA 817794

> hist(lulc)
```

This is a raster with integer values between 1 and 6, but for this raster to be meaningful at all, we need a lookup or attribute table to identify these classes. A .csv file has also been provided as part of the package. Read it in as a data.frame:

```
> # Note: the ID column is the first one, hence 'row.names = 1'
> lulc.classes <- read.csv('path/to/data/LULC2011_Gewata_classes.csv, row.names = 1)
```

This data.frame represents a lookup table for the raster we just loaded. The ID column corresponds to the values taken on by the lulc raster, and the 'Class' column describes the LULC classes assigned. In R it is possible to add a attribute table to a raster. In order to do this, we need to coerce the raster values to a factor from an integer.

```
> lulc <- as.factor(lulc)
```

If you display the attributes of this raster (just type 'lulc'), it will do so, but will also return an error. This error arises because R expects that a raster with factor values should also have a raster attribute table.

```
> # assign a raster attribute table (RAT)
> levels(lulc) <- lulc.classes
> lulc
```

```
class       : RasterLayer
dimensions  : 1177, 1548, 1821996  (nrow, ncol, ncell)
resolution  : 30, 30  (x, y)
extent      : 808755, 855195, 817635, 852945  (xmin, xmax, ymin, ymax)
coord. ref. : +proj=utm +zone=36 +datum=WGS84 +units=m +no_defs +ellps=WGS84 +towgs84=0,0,
data source : /Users/Ben/R/projects/rasta/www/Kafa_BR/LULC2011_Gewata.tif
names       : LULC2011_Gewata
values      : 1, 6  (min, max)
attributes  :
       ID    Class
 from:  1 cropland
 to  :  6  wetland
```

Zonal Statistics

...to do...

Calculating focal values

...write a function to remove 'lone' pixels from a raster layer using the focal() function... ...write another function (or amend the one above) to apply an area threshold to a raster dataset...

4 Working with multilayered raster data

When working with multispectral or multitemporal raster data, it is convenient to represent multiple raster layers as a single object in R. R works with two types of multilayer raster objects: stacks and bricks. The main difference between the two is that raster stacks can be read from several different data sources (files) and bricks are read from a single file (e.g. a multiband GeoTIFF).

A raster brick from a small area within the Kafa Biosphere Reserve in Southern Ethiopia can be found in the rasta package. Set the working directory to the packages home folder and load the raster brick from file by

```
> # set the working directory to the directory containing the data
> setwd('path/to/the/data')
> tura <- brick('tura.grd')
> # alternatively, without (re)setting the working directory:
> tura <- brick('path/to/data/tura.grd')

> # inspect the data
> class(tura) # the object's class

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

> projection(tura) # the projection

[1] "+proj=utm +zone=36 +ellps=WGS84 +units=m +no_defs"

> res(tura) # the spatial resolution (x, y)

[1] 30 30

> extent(tura) # the extent of the raster brick

class       : Extent
xmin        : 819105
xmax        : 823395
ymin        : 827745
ymax        : 832185
```

Extracting scene information

This RasterBrick was read from a .grd file. One advantage of this file format (over the GeoTIFF format, for example) is the fact that the specific names of the raster layers making up this brick have been preserved, a feature which is important for identifying raster layers, especially when doing time series analysis (where you need to know the values on the time axis). This RasterBrick was prepared from a Landsat 7 ETM+ time series, and the original scene names were inserted as layer names.

```
> names(tura) # displays the names of all layers in the tura RasterBrick
```

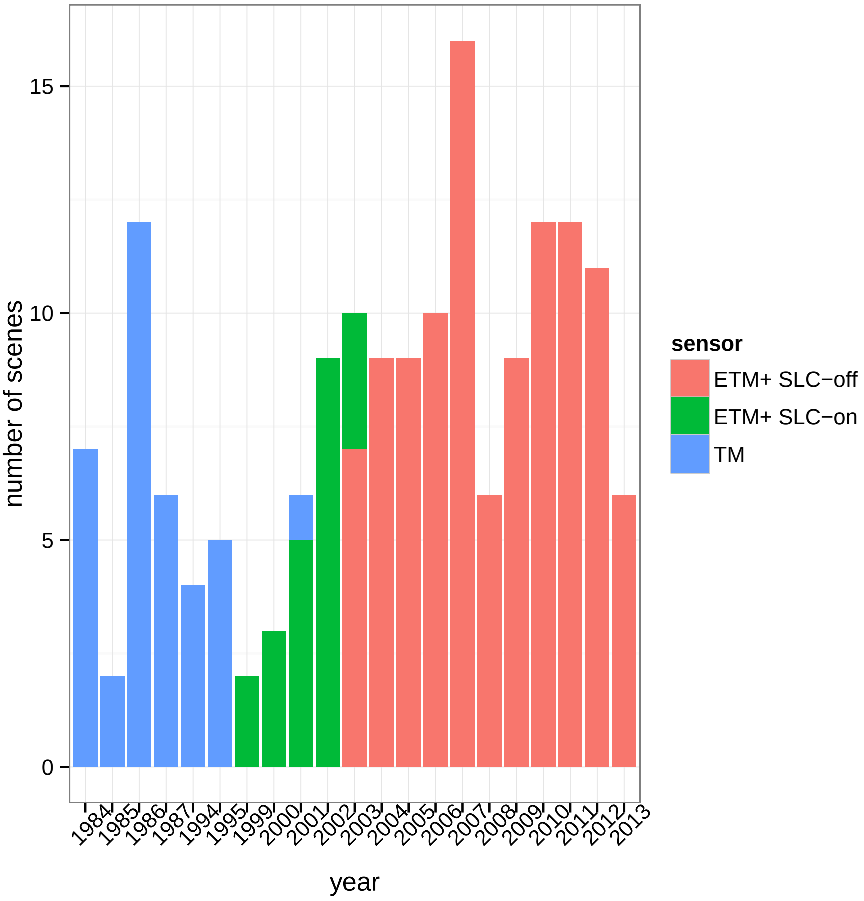
We can parse these names to extract information from them. The first 3 characters indicate which sensor the data come from, with 'LE7' indicating Landsat 7 ETM+ and 'LT5' or 'LT4' indicating Landsat 5 and Landsat 4 TM, respectively. The following 6 characters indicate the path and row (3 digits each), according to the WGS system. The following 7 digits represent the date. The date is formatted in such a way that it equals the year + the julian day. For example, February 5th 2001, aka the 36th day of 2001, would be '2001036'.

```
> # display the 1st 3 characters of the layer names
> sensor <- substr(names(tura), 1, 3)
> print(sensor)
> # display the path and row as numeric vectors in the form (path,row)
> path <- as.numeric(substr(names(tura), 4, 6))
> row <- as.numeric(substr(names(tura), 7, 9))
> print(paste(path, row, sep = ","))
> # display the date
> dates <- substr(names(tura), 10, 16)
> print(dates)
> # format the date in the format yyyy-mm-dd
> as.Date(dates, format = "%Y%j")
```

There is a function in the rasta package, getSceneinfo() that will parse these names and output a data.frame with all of these attributes.

```
> sceneinfo <- getSceneinfo(names(tura))
> print(sceneinfo)
> # add a 'year' column to the sceneinfo dataframe and plot #scenes/year
> sceneinfo$year <- factor(substr(sceneinfo$date, 1, 4), levels = c(1984:2013))
```

```
> # barplot with number of scenes per year
> library(ggplot2)
> ggplot(data = sceneinfo, aes(x = year, fill = sensor)) +
+   geom_bar() +
+   labs(y = "number of scenes") +
+   theme_bw() +
+   theme(axis.text.x = element_text(angle = 45))
```



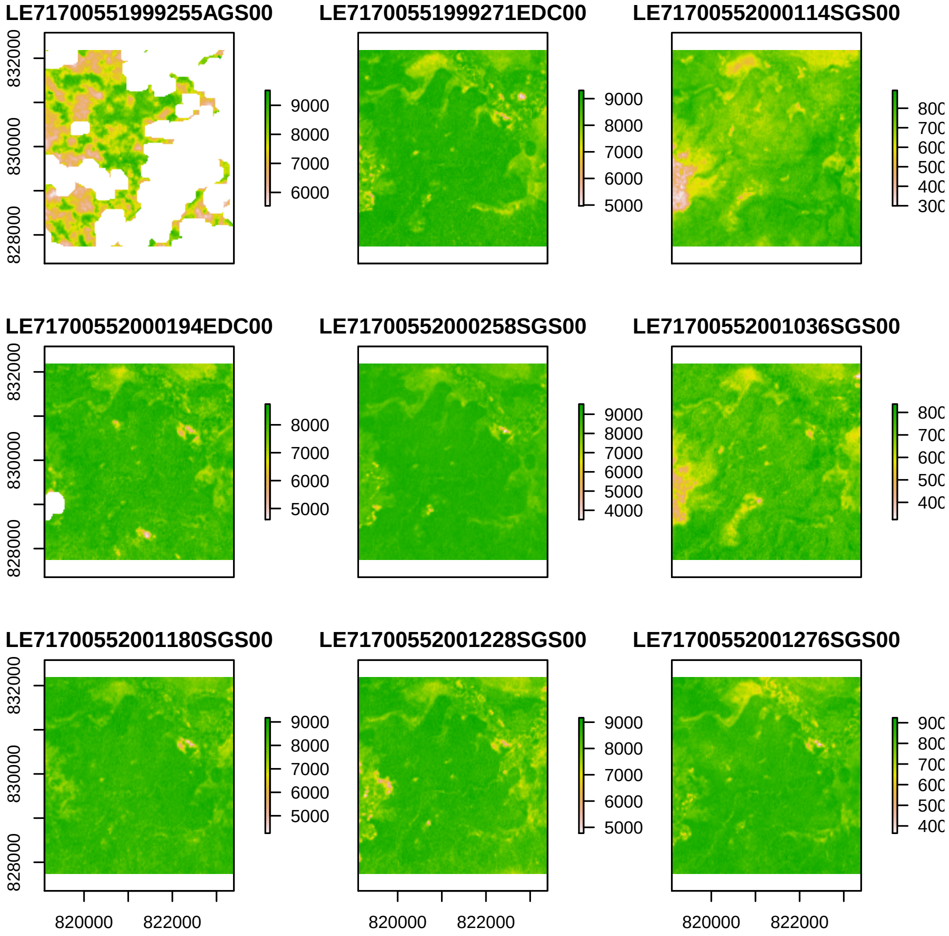
Note that the values along the x-axis of this plot are evenly distributed, even though there are gaps in the values (e.g. between 1987 and 1994). The spacing is due to the fact that we defined sceneinfo$year as a vector of *factors* rather than a numeric vectors. Factors act as thematic classes and can be represented by numbers or letters. In this case, the actual values of the factors are not recognized by R. Instead, the levels defined in the factor() function define the hierarchy of the factors (in this case we have defined the levels from 1984 up to 2013, according to the range of acquisition dates). For more information on factors in R, check out `http://www.stat.berkeley.edu/classes/s133/factors.html`.

Try to generate the plot above with the years (x-axis) represented as a numeric vector instead of as a factor. Hint: it is not as straightforward as you might think - to convert a factor x to a numeric vector, try *x <- as.numeric(as.character(x))*.

Plotting RasterBricks

A RasterBrick can be plotted just as a RasterLayer, and the graphics device will automatically split into panels to accommodate the layers (to an extent: R will not attempt to plot 100 layers at once!). To plot the first 9 layers:

```
> plot(tura, c(1:9))
> # alternatively, you can use [[]] notation to specify layers
> plot(tura[[1:9]])
> # use the information from sceneinfo data.frame to clean up the titles
> plot(tura[[1:9]], main = sceneinfo$date[c(1:9)])
```



Unfortunately, the scale is different for each of the layers, making it impossible to make any meaningful comparison between the raster layers. This problem can be solved by specifying a breaks argument in the plot() function.

```
> # we need to define the breaks to harmonize the scales (to make the plots comparable)
> bks <- seq(0, 10000, by = 2000) # (arbitrarily) define the breaks
> # we also need to redefine the colour palette to match the breaks
> cols <- rev(terrain.colors(length(bks))) # col = rev(terrain.colors(255)) is the default
> # (opt: check out the RColorBrewer package for other colour palettes)
> # plot again with the new parameters
> plot(tura[[1:9]], main = sceneinfo$date[1:9], breaks = bks, col = cols)
```

Alternatively, the rasterVis package has some enhanced plotting functionality for raster objects, including the levelplot() function, which automatically provides a common scale for the layers.

```
> library(rasterVis)
> levelplot(tura[[1:6]])
> # NOTE:
> # for rasterVis plots we must use the [[]] notation for extracting layers
```

```
> # providing titles to the layers is done using the 'names.attr' argument in place of 'ma
> levelplot(tura[[1:8]], names.attr = sceneinfo$date[1:8])
> # define a more logical colour scale
> library(RColorBrewer)
> # this package has a convenient tool for defining colour palettes
> ?brewer.pal
> display.brewer.all()
> cols <- brewer.pal(11, 'PiYG')
> # to change the colour scale in levelplot(), we first have to define a rasterTheme objec
> # see ?rasterTheme() for more info
> rtheme <- rasterTheme(region = cols)
> levelplot(tura[[1:8]], names.attr = sceneinfo$date[1:8], par.settings = rtheme)
```

This plot gives us a common scale which allows us to compare values (and perhaps detect trends) from layer to layer. In the above plot, the layer titles do not look very nice – we will solve that problem a bit later.

The rasterVis package has integrated plot types from other packages with the raster package to allow for enhanced analysis of raster data.

```
> # histograms of the first 6 layers
> histogram(tura[[1:6]])
> # box and whisker plot of the first 9 layers
> bwplot(tura[[1:9]])
```

More examples from the rasterVis package can be found @ http://oscarperpinan.github.io/rastervis/

Calculating data loss

In this RasterBrick, the layers have all been individually preprocessed from the raw data format into NDVI values. Part of this process was to remove all pixels obscured by clouds or SLC-off gaps (for any ETM+ data acquired after March 2003). For this reason, it may be useful to know how much of the data has been lost to cloud cover and SLC gaps. First, we will calculate the percentage of no-data pixels in each of the layers using the freq() function. freq() returns a table (matrix) of counts for each value in the raster layer. It may be easer to represent this as a data.frame to access column values.

```
> # try for one layer first
> y <- freq(tura[[1]]) # this is a matrix
> y <- as.data.frame(y)
> # how many NA's are there in this table?
> y$count[is.na(y$value)]
```

```
[1] 11340
```

```
> # alternatively, using the with() function:
> with(y, count[is.na(value)])
```

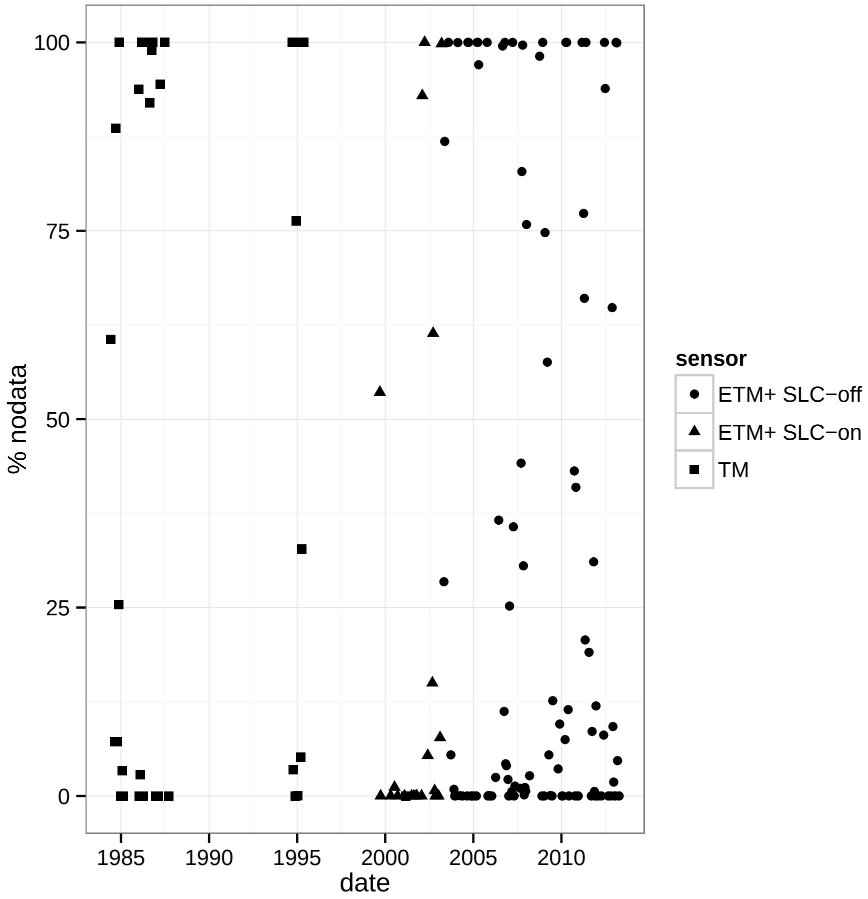
```
[1] 11340
```

```
> # as a %
> with(y, count[is.na(value)]) / ncell(tura[[1]]) * 100
```

```
[1] 53.58155
```

```
> # apply this over all layers in the RasterBrick
> # first, prepare a numeric vector to be 'filled' in
> nas <- vector(mode = 'numeric', length = nlayers(tura))
> for(i in 1:nlayers(tura)){
+   y <- as.data.frame(freq(tura[[i]]))
+   # if there are no NAs, then simply assign a zero
+   # otherwise, grab the # of NAs from the frequency table
+   if(!TRUE %in% is.na(y$value)){
+     nas[i] <- 0
+   } else {
+     nas[i] <- with(y, count[is.na(value)]) / ncell(tura[[i]]) * 100
+   }
+ }
> # add this vector as a column in the sceneinfo data.frame (rounded to 2 decimal places)
> sceneinfo$nodata <- round(nas, 2)
```

```
> # plot these values
> ggplot(data = sceneinfo, aes(x = date, y = nodata, shape = sensor)) +
+   geom_point(size = 2) +
+   labs(y = "% nodata") +
+   theme_bw()
```



We have now derived some highly valuable information about our time series. For example, we may want to select an image from our time series with relatively little cloud cover to perform a classification. For further time series analysis, the layers with 100% data loss will be of no use to us, so it may make sense to get rid of these layers.

```
> # which layers have 100% data loss?
> which(sceneinfo$nodata == 100)

 [1]  13  25  34  35  41  42  44  54  62  78  92  93 101 105 117 126 135 143 147
[20] 149 155 157 162 165

> # supply these indices to the dropLayer() command to get rid of these layers
> tura <- dropLayer(tura, which(sceneinfo$nodata == 100))
> # redefine our sceneinfo data.frame as well
> sceneinfo <- sceneinfo[which(sceneinfo$nodata != 100), ]
> # optional: remake the previous ggplots with this new dataframe
```

With some analyses, it may also be desireable to apply a no-data threshold per scene, in which case layer indices would be selected by:

```
> which(sceneinfo$nodata > some_threshold)
```

In some cases, there may be parts of the study area with more significant data loss due to persistant cloud cover or higher incidence of SLC-off gaps. To map the spatial distribution of data loss, we need to calculate the % of NA in the time series for each *pixel* (ie. looking 'through' the pixel along the time axis). To do this, it is convenient to use the calc() function and supply a special function which will count the number of NA's for each pixel along the time axis, divide it by the total number of data in the pixel time series, and output a percentage. calc() will output a raster with a percentage no-data value for each pixel.

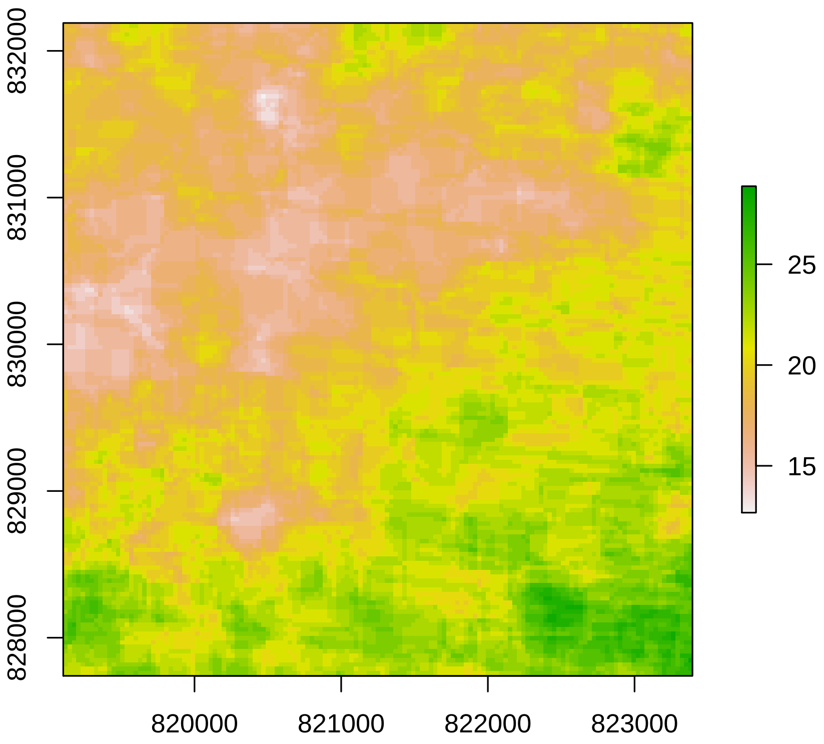
```
> # calc() will apply a function over each pixel
> # in this case, each pixel represents a time series of NDVI values
> # e.g. all values of the 53rd pixel in the raster grid:
> y <- as.numeric(tura[53])
> # how many of these values have been masked (NA)?
> length(y[is.na(y)])
```

```
[1] 23
```

```
> # as a %
> length(y[is.na(y)]) / length(y) * 100
```

```
[1] 16.19718
```

```
> # now wrap this in a calc() to apply over all pixels of the RasterBrick
> nodata <- calc(tura, fun = function(x) length(x[is.na(x)]) / length(x) * 100)
```



Summary statistics

In addition to calculating the % of NA in the time series, there are many other ways we can describe this raster time series. For example, we may want to know the mean or median NDVI value over the whole time series, for example.

...to do....

5 Times Series Analysis

- making time series plots for selected pixels (interactively or otherwise)
- deriving time series statistics
  - linear regression from pixel time series
  - deriving seasonality parameters?
  - creating figures maps using these statistics

There is an easy way to interactively extract data from a single pixel using the raster package. First, a raster plot should be made from which to identify the pixel of interest. Then, the click() function allows for extraction of the data contained within that pixel. Simply calling click() with no further arguments will only return the (x, y) coordinates of that point as a 1-row matrix.

```
 > plot(tura, 101)
 > click()
            x        y
[1,] 819695.2 830544.3
```

We can extract more meaningful information by passing the name of the object (in this case, a RasterBrick) as the first argument, followed by the argument 'n = 1' (or the number of desired points to identify).

```
> plot(tura, 101)
> click(tura, n = 1)
```

In this case, all data from that pixel are extracted. These time series could then easily be coerced (or inserted) into a data.frame to plot the data and further analyze them. Let's take a look at a the time series of a few different pixels. Instead of using the interactive click() function to extract these data, suppose that we know the (x, y) coordinates of a land cover type (or other points of interest, from a field campaign or ground truth dataset, for example).

```
> # several pixel coordinate pairs expressed as separate 1-row matrices
> forest <- matrix(data = c(819935, 832004), nrow = 1, dimnames = list(NULL, c('x', 'y')))
> cropland <- matrix(data = c(819440, 829346), nrow = 1, dimnames = list(NULL, c('x', 'y')
> wetland <- matrix(data = c(822432, 832076), nrow = 1, dimnames = list(NULL, c('x', 'y'))
> # recall that we can extract pixel data if we know the cell #
> # we can easily convert from xy matrix to cell number with cellFromXY()
> cellFromXY(tura, forest)
> tura[cellFromXY(tura, forest)] # returns a 1-row matrix with all ts values
```

Now we are able to extract the time series data given a set of (x, y) coordinates. Let's put the data from these three points into a data.frame to facilitate plotting of the data.

```
> # prepare the data.frame
> ts <- data.frame(sensor = getSceneinfo(names(tura))$sensor,
+                  date = getSceneinfo(names(tura))$date,
+                  forest = t(tura[cellFromXY(tura, forest)]),
+                  cropland = t(tura[cellFromXY(tura, cropland)]),
+                  wetland = t(tura[cellFromXY(tura, wetland)])
+                  )
> print(ts)
> # simple plot of forest time series
> plot(ts$date, ts$forest)
> # same thing but using with()
> with(ts, plot(date, forest))
```

Note the two large gaps in the time series during the 1990's, during which time there are no Landsat data available from the USGS. While we could still use these data to understand historical trends, we will only look at time series data from the ETM+ sensor (ie. data acquired after 1999) for the following exercises.

```
> # remove all data from the TM sensor and plot again
> ts <- ts[which(ts$sensor != "TM"), ]
> with(ts, plot(date, forest))
```

A more informative plot would show these time series side by side with the same scale or on the same plot. These are possible with either the base plot() function or using ggplot2. Either way, there is some preparation needed, and in the case of ggplot2, this may not be immediately obvious. In the following example, we are going to make a facet_wrap plot. In order to do so, we need to merge the time series columns to make a data.frame with many rows indeed. An additional column will be used to identify the class (forest, cropland or wetland) of each data point, and this class will be used to 'split' the data into 3 facets. The reshape package has a convenient function, melt(), which will 'automatically' reshape the data.frame to make it passable to the ggplot framework.

```
> library(reshape)
> # convert dates to characters, otherwise melt() returns an error
> ts$date <- as.character(ts$date)
> tsmelt <- melt(ts)
> head(tsmelt)

       sensor       date variable value
1 ETM+ SLC-on 1999-09-12   forest  6532
2 ETM+ SLC-on 1999-09-28   forest  8812
3 ETM+ SLC-on 2000-04-23   forest  7564
4 ETM+ SLC-on 2000-07-12   forest  8095
5 ETM+ SLC-on 2000-09-14   forest  8677
6 ETM+ SLC-on 2001-02-05   forest  7279

> names(tsmelt) <- c('sensor', 'date', 'class', 'value')
> # convert tsplot$date back to Date class to enable formatting of the plot
> tsmelt$date <- as.Date(tsmelt$date)
> tsplot <- ggplot(data = tsmelt, aes(x = date, y = value / 10000)) +
+   geom_point() +
+   scale_x_date() +
+   labs(y = "NDVI") +
+   facet_wrap(~ class, nrow = 3) +
+   theme_bw()
> tsplot
```

6 Exercise

- to be doable in 3 hours....
- combine concepts from previous lessons as well
- example 1: produce a figure with maximum/minimum/median/mean NDVI per year; figure should have a common scale and be properly labelled

- example 2: compute a time series metric over an entire RasterBrick using the calc() function