Raster Time Series

Ben DeVries, Jan Verbesselt

November 7, 2013

1 Learning Objectives

The learning objectives for this lesson are:

1. working with time series rasterBricks
2. parsing Landsat scene information
3. interactively extracting data from raster pixels
4. plotting time series data
5. summarizing data in a time series using `ggplot()`
6. using spatial joins to summarize raster time series data

2 Working with raster time series

Since the opening of the Landsat data archive, image time series analysis has become a real possibility. In environmental research and monitoring, methods and tools for time series analysis are becoming increasingly important. With the brick and stack object types, the raster package in R allows for certain types of raster-based time series analysis, and allows for integration with other time series packages in R.

Let's start by loading the necessary packages.

```
> # load the necessary packages
> library(rasta)
> library(raster)
> library(rgdal)
> library(sp)
> library(ggplot2)
> library(rasterVis)
```

A raster brick representing NDVI values over time from a small area within the Kafa Biosphere Reserve in Southern Ethiopia can be found in the rasta package.

```
> data(tura)
> # inspect the data
> class(tura) # the object's class
> projection(tura) # the projection
> res(tura) # the spatial resolution (x, y)
> extent(tura) # the extent of the rasterBrick
> # how many layers are there in this rasterBrick?
> nlayers(tura)
```

These data have been rescaled to integer by multiplying by 10000 and removing the decimal place and converting to an integer data type. For our plots and analyses to be more interpretable, we will scale the NDVI back to its original scale before proceeding.

```
> tura <- tura/10000
> # check the range of values
> minValue(tura)
> maxValue(tura)
```

Extracting scene information

The Tura rasterBrick object comes as part of the rasta package. During preparation of the package, 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.

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

The analyses that follow in this lesson depend on these names, as they contain important information about the sensor, acquisition date, and path and row. We can parse these names using the character splitting function `strsplit()` 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 WRS system[1]. 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
```

[1]A shapefile or kml of the Landsat WRS system can be downloaded at `http://landsat.usgs.gov/tools_wrs-2_shapefile.php`.

```
> dates <- substr(names(tura), 10, 16)
> print(dates)
> # format the date in the format yyyy-mm-dd
> print(as.Date(dates, format="%Y%j"))
```

There is a function in the rasta package, `getSceneinfo()` that will parse Landsat scene names according to the above principles and output a data.frame with all of these attributes, with an added option to write this data.frame to csv file. In the following example, we will parse the scene information contained in the Tura layer names and plot them using `ggplot()`[2].

```
> # check out the help file for getSceneinfo()
> ?getSceneinfo
> # extract layer names from the tura rasterBrick
> sceneID <- names(tura)
> # parse these names to extract scene information
> sceneinfo <- getSceneinfo(sceneID)
> print(sceneinfo)
> # look at acquisition dates as a vector
> sceneinfo$date
> # add a 'year' column to the sceneinfo data.frame
> sceneinfo$year <- factor(substr(sceneinfo$date, 1, 4), levels = c(1984:2013))
> # plot the # of scenes available per year
> ggplot(data = sceneinfo, aes(x = year, fill = sensor)) +
+   geom_bar(width=0.5) +
+   labs(y = "number of scenes") +
+   scale_y_continuous(limits=c(0, 20)) +
+   theme_bw() +
+   theme(axis.text.x=element_text(angle = 45),
+         legend.position=c(0.85, 0.85))
```

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 vector. 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(levels(x))`. Or you can omit the `factor()` function in defining the "years" column in the above data.frame.

[2]In this lesson, we will not go into very much depth on the use of `ggplot()` and associated functions. The ggplot2 homepage (`http://ggplot2.org`) has links to some reading material on ggplot2, and especially to the excellent function documentation page (`http://docs.ggplot2.org`).

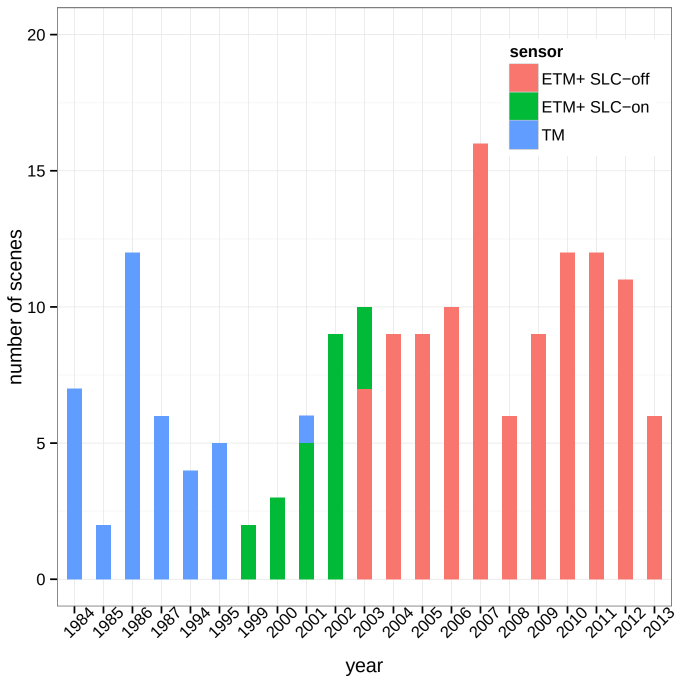

Figure 1: Number of scenes available per year in the time series.

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)
> # create a sequence of (arbitrary) breaks
> bks <- seq(0, 1, by=0.2)
> # we also need to redefine the colour palette to match the breaks
> cols <- rev(terrain.colors(length(bks)))
> # NOTE: col = rev(terrain.colors(255)) is the default colour palette
> #       set in raster plots,
> # ....but there are many more options available!
> # 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 instead of 'main'
> levelplot(tura[[1:6]], names.attr=sceneinfo$date[1:6])
> # define a more logical colour scale
> library(RColorBrewer)
> # this package has a convenient tool for defining colour palettes
> # display available colour palettes
> display.brewer.all()
> # or see ?brewer.pal()
> cols <- brewer.pal(11, 'PiYG')
> # to change the colour scale in levelplot(),
> # we first have to define a rasterTheme object
> # see ?rasterTheme() for more info
> # define a raster theme based on our colour palette
> rtheme <- rasterTheme(region=cols)
> # this is simply a list with plot theme attributes
> class(rtheme)
> names(rtheme)

> # pass the rtheme list to par.settings in the levelplot() function
> levelplot(tura[[1:6]], names.attr=sceneinfo$date[1:6], par.settings=rtheme)
```

This plot (Figure 2) gives us a common scale which allows us to compare values (and perhaps detect trends) from layer to layer. The rasterVis package also has integrated plot types from other packages with the raster package to allow for enhanced visualization 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

The individual layers of the tura rasterBrick have all been individually preprocessed from a raw data format into NDVI values. Part of this process was to remove all pixels obscured by clouds, cloud shadows or SLC-off gaps (for any ETM+ data acquired after March 2003).

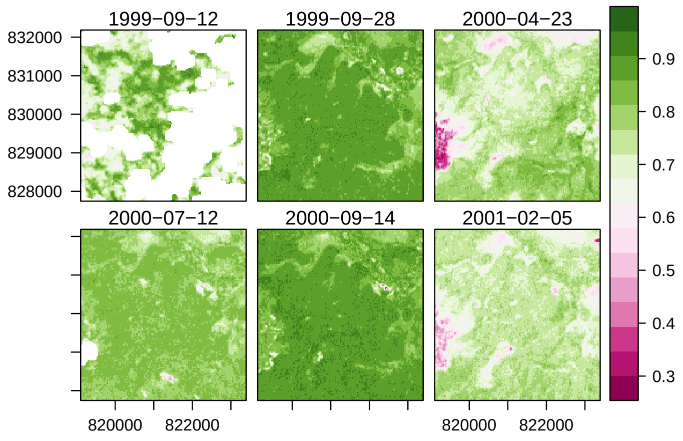

Figure 2: The first six layers of the Tura rasterBrick as visualized using the levelplot() function in rasterVis.

For this reason, it may be useful to know how much of the data has been lost in the masking process. 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 easier to represent this as a data.frame to access column values.

```
> # try for one single layer first
> y <- freq(tura[[1]])
> # this returns a 2-column matrix
> head(y)
> # coerce to a data.frame
> y <- as.data.frame(y)
> head(y)
> # how many pixels are there with value=NA?
> y$count[is.na(y$value)]
> # alternatively, using the with() function:
> with(y, count[is.na(value)])
> # as a %
> with(y, count[is.na(value)]) / ncell(tura[[1]]) * 100
> # We can compute this more efficiently
> # by using the 'value' argument in freq()
> y <- freq(tura[[1]], value=NA)
> print(y)
> # apply over all layers of the rasterBrick
> y <- freq(tura, value=NA)
> print(y)
> # convert this to % pixels per scene
> y <- freq(tura, value=NA) / ncell(tura) * 100
> # add this vector as a column in the sceneinfo data.frame
> #(rounded to 2 decimal places)
> sceneinfo$nodata <- round(y, 2)

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

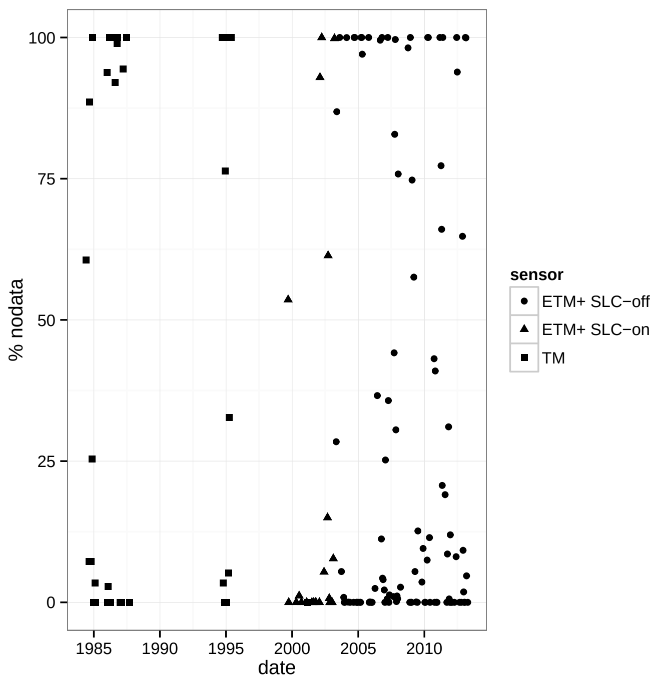

Figure 3: % NAs found in each scene of a Landsat time series.

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$nodata == 100)
> # supply these indices to the dropLayer() command to get rid of these layers
> tura <- dropLayer(tura, which(sceneinfo$nodata == 100))
> # redefine our sceneinfo data.frame so to correspond with rasterBrick
> 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. extract all values of the 53rd pixel in the raster grid
> y <- as.numeric(tura[53])
> # this is an integer vector from time series values
> print(y)

  [1] 0.7403 0.8746 0.7174 0.8003 0.8468 0.7512 0.8281 0.8652 0.8254 0.8431
 [11] 0.7126     NA 0.8437 0.8572     NA 0.8407 0.8123 0.7589 0.7867 0.7474
 [21]     NA     NA     NA 0.8407 0.8057 0.8293 0.8059     NA 0.6895 0.7526
 [31] 0.8227 0.8382 0.8487 0.7737 0.6304 0.7664     NA 0.8264 0.8312 0.8286
 [41] 0.7702 0.7758 0.7919 0.7138     NA 0.7948 0.8769 0.8200 0.8293 0.8275
 [51] 0.8614 0.8227 0.7510 0.6730 0.8277 0.8446 0.8527 0.8687     NA     NA
 [61]     NA 0.8311 0.8221 0.7801     NA 0.7394     NA 0.8295 0.8026 0.7907
 [71] 0.7796 0.7572 0.6462 0.8071 0.8407     NA 0.7687 0.8007 0.7865 0.7982
 [81] 0.7637 0.7664 0.7939 0.8226 0.8127 0.7429 0.8324 0.8253     NA     NA
 [91] 0.6638     NA 0.8154 0.8365 0.7370 0.7920 0.8181 0.8260 0.7983 0.7649
[101] 0.8265 0.7652     NA 0.8479 0.8325 0.7194 0.7939 0.8329 0.7805 0.7858
[111]     NA 0.6796 0.8047 0.7438 0.7142 0.7904 0.8104 0.3645 0.7681 0.7361
[121] 0.6037 0.6363 0.6483 0.6831 0.7500 0.8029 0.7582     NA     NA 0.7844
[131] 0.7249 0.6918 0.5954     NA 0.7673 0.8278 0.8144     NA 0.6710 0.7332
[141] 0.8237 0.7461

> # 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)
> # it's more readable to define the function first...
> percNA <- function(x){
+   y <- length(x[is.na(x)]) / length(x) * 100
+   return(y)
+ }
> # ...and then pass it to calc()
> nodata <- calc(tura, fun=percNA)
> # plot the nodata raster
> plot(nodata)
```

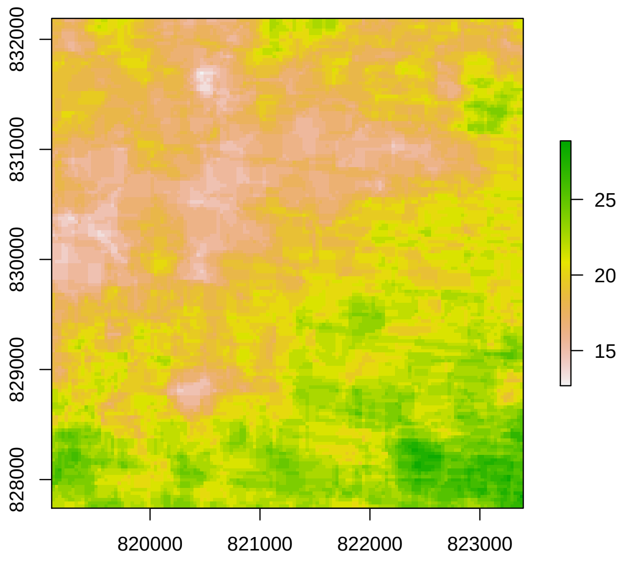

Figure 4: Percent NA throughout the time series for each pixel in the Tura rasterBrick.

From Figure 4 we can now see that after removing the scenes with 100% data loss, our time series is reasonably dense, with a loss of between 10% to 30% of the data due to clouds, cloud shadows and SLC-off gaps. We can also begin to describe spatial patterns of data availability, and while the variance is not very striking in this case, areas with sharp elevation gradients may also show stark differences in data availability. Since availability of data in the temporal domain can be a critical constraint for raster time series analysis, producing plots like Figures 3 and 4 are important initial steps in evaluating the suitability of the data for such analyses.

Extracting pixel times series

To gain insight on the behaviour of our data over time, we will need to look at how the value of a single pixel (or group of pixels) changes throughout the time series. This observation can tell us about possible phenology or possible abrupt changes (e.g. deforestation, urban development, etc.). We saw earlier that we can pull out a time series vector of any pixel in the rasterBrick simply by supplying the cell number in a single square bracket.

```
> # time series of the 53rd pixel in the grid
> x <- tura[53]
> print(x)
> # returns a matrix with 1 row
> class(x)
> nrow(x)
> # we can extract multiple pixels at one time
> x <- tura[c(53, 100)]
> print(x)
> nrow(x)
> x <- tura[c(100:199)]
> head(x)
> nrow(x)
> # etc...
```

In the data exploration phase, it would be more useful to select a time series directly from a map, without having to know the cell numbers. 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 also select more than one cell at a time by passing the argument `n=...`.

```
> plot(tura, 101)
> click(n=2)
            x        y
[1,] 819549.3 831687.2
[2,] 819916.8 830001.8
```

So far, calling `click()` has only resulted in the return of (x,y) coordinates. We can extract more meaningful information by passing the name of a raster 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)
> x <- click(tura, n=1)
> # returns a matrix with n=1 rows
> class(x)
> # convert it to an integer vector
> x <- x[1,]
> # or....
> x <- as.numeric(x)
> # note that the 2nd method removes the names of the vector!
> # plot the vector
> plot(x)
```

In this case, all data from that pixel are extracted. Plotting the values gives a sense of the behaviour of the pixel's values over time, but we need more information to make any real conclusions about the pixel's values over time. To this end, this time series vector 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')))
> # e.g.
> print(forest)
```

We can easliy determine the cell number in our raster grid by using the `cellFromXY()` function from the raster package.

```
> forestCell <- cellFromXY(tura, forest)
> print(forestCell)
> croplandCell <- cellFromXY(tura, cropland)
> wetlandCell <- cellFromXY(tura, wetland)
> # return a 1-row matrix with all time series values from this cell
> tura[forestCell]
```

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 with `ggplot()`. In this data.frame, we will combine data regarding sensor type and acquisition date returned earlier from `getSceneinfo()` with time series values extracted from different grid cells. With the date column, we will be able to make a plot with a meaningful x-axis, and with the sensor column, we can

```
> # prepare the data.frame
> ts <- data.frame(sensor = getSceneinfo(names(tura))$sensor,
+                  date = getSceneinfo(names(tura))$date,
+                  forest = t(tura[forestCell]),
+                  cropland = t(tura[croplandCell]),
+                  wetland = t(tura[wetlandCell])
+                  )
> print(ts)
```

Note the use of `t()` to transpose the time series matrices into columns. To see why this is necessary, try remaking the above data.frame without including `t()`. Alternatively, each of the time series matrices could first be coerced to vectors using `x[1,]` or `as.numeric(x)` as we saw earlier before inserting into the data.frame.

```
> # simple plot of forest NDVI time series
> plot(ts$date, ts$forest, xlab="date", ylab="NDVI")
> # same thing, but using with() instead of '$' operator
> with(ts, plot(date, forest, xlab="date", ylab="NDVI"))
```

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"), ]
> # alternatively, using subset()
> ts <- subset(ts, sensor!="TM")
> # plot the new time series
> with(ts, plot(date, forest, xlab="date", ylab="NDVI"))
> # time series for a cropland pixel
> with(ts, plot(date, cropland, xlab="date", ylab="NDVI"))
> # wetland
> with(ts, plot(date, wetland, xlab="date", ylab="NDVI"))
```

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 which `ggplot()` can interpret. An additional column will be used to identify the class (forest, cropland or wetland) of each data point, and this class column 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)
> # 'melt' the data.frame
> tsmelt <- melt(ts)
> # inspect the new data.frame
> head(tsmelt)
> # change the 'variable' column heading to 'class'
> names(tsmelt)[3] <- "class"
> # convert tsplot$date back to Date class to enable formatting of the plot
> tsmelt$date <- as.Date(tsmelt$date)
> # inspect the data.frame
> tsmelt
```

We can now pass this “reshapen” data.frame to `ggplot()` to make a facet_wrap plot.

```
> ggplot(data = tsmelt, aes(x = date, y = value)) +
+   geom_point() +
+   scale_x_date() +
+   labs(y = "NDVI") +
+   facet_wrap(~ class, nrow = 3) +
+   theme_bw()
```

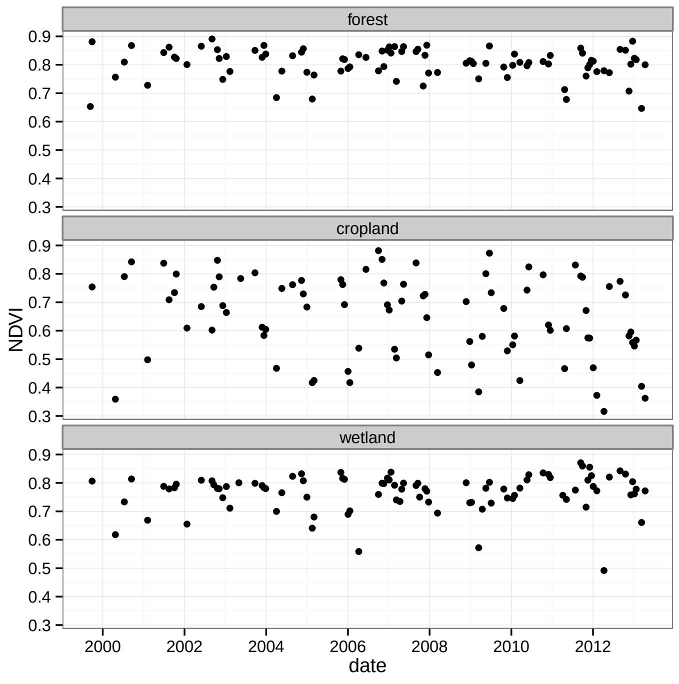

Figure 5: Single pixel NDVI time series from 1999 to 2012 for three land cover classes: forest, cropland, and wetland.

3 Spatial Joins: zonal() and extract()

Now that we are able to inspect time series for individual pixels, we will now extract summary time series for regions within our scene. We will do this by using *a priori* information on our scene. First, we will extract summary time series using the Land Use and Land Cover (LULC) raster we worked with in Lesson 6 using a raster to raster spatial join approach.

Raster-Raster joins using `zonal()`

In Lesson 6 we worked with a Land Use and Land Cover (LULC) raster covering Gewata in 2011. Using the principles of a raster to raster spatial join, we can describe the time series for these classes. First, we will need to load in the raster and crop it to the same extent as the tura rasterBrick.

```
> # load in the data
> data(lulcGewata)
> # crop to the same extent as the tura rasterBrick
> lulc <- crop(lulcGewata, extent(tura))
> plot(lulc)
> freq(lulc)
> # load in the Look-up Table (LUT) to identify classes
> data(LUTGewata)
> print(LUTGewata)
```

At this extent, only 3 classes seem to be represented: cropland, forest and wetland. To demonstrate how `zonal()` works, let's just take one layer of the tura rasterBrick and test out the function.

```
> # extract a layer from tura
> x <- raster(tura, 101)
> # distribution of values in this layer
> histogram(x)
> # mean NDVI per LULC class
> meanNDVI <- zonal(x=x, z=lulc, fun='mean')
> print(meanNDVI)
> # standard deviation of the NDVI per class
> sdNDVI <- zonal(x=x, z=lulc, fun='sd')
> print(sdNDVI)
```

We can see that the cropland (zone 1) has a lower mean NDVI and forest (zone 5) has the highest in the 101st layer of the tura rasterBrick. From the standard deviation, we see that wetland (zone 6) has the highest, followed closely by cropland, which is likely due to the dynamic phenologies of these classes. In contrast, the forest class has a relatively lower standard deviation, which is consistent with the time series plots we saw in Figure 5.

So far we have only extracted single values for each zone, since we only applied the function to a single raster layer. Using `zonal()` on a rasterBrick, however, we can extract the entire time series data not only from one pixel, but an entire LULC class. Let's execute `zonal()` on the tura rasterBrick using the default function (mean), meaning the function will return the mean value in each of the three zones for *each* layer of the rasterBrick.

```
> zonestats <- zonal(x=tura, z=lulc, fun="mean")
> head(zonestats)
> class(zonestats)
```

The output is a matrix with 3 rows representing each of the land cover classes, and as many columns as there are layers in the rasterBrick. In effect, each row is analogous to one of the time series columns in the data.frame we passed to `ggplot()` earlier. Let's do the same with these time series. This time, we will prepare three separate data.frames for each land cover class, and merge them afterwards using the commonly used `rbind()` function. As before, we will include Landsat scene information in the data.frames to assist with the plotting. NOTE: it is important to remember which row of the zonestats matrix corresponds to which class, otherwise we risk making a serious error in our analysis!

```
> # note the first column of zonestats represents the labels
> zonestats[,1]

[1] 1 5 6

> # remove this column so it doesn't get inserted into data.frames
> zonestats <- zonestats[,-1]
> # prepare the data.frames for each LC class
> # forest ts is in the 2nd row of the matrix
> foreststs <- data.frame(date = sceneinfo$date,
+                         sensor = sceneinfo$sensor,
+                         ndvi = zonestats[2,],
+                         class = "forest")
> # cropland is in the 1st row of the matrix
> croplandts <- data.frame(date = sceneinfo$date,
+                           sensor = sceneinfo$sensor,
+                           ndvi = zonestats[1,],
+                           class = "cropland")
> # wetland is in the 3rd row of the matrix
> wetlandts <- data.frame(date = sceneinfo$date,
+                          sensor = sceneinfo$sensor,
+                          ndvi = zonestats[3,],
+                          class = "wetland")
> # merge these data.frames into one
> ts <- rbind(foreststs, croplandts, wetlandts)
> # remove the other data.frames from the workspace
> rm(foreststs, croplandts, wetlandts)
> # as before, only include data from the ETM+ sensor (remove all TM)
> ts <- subset(ts, sensor!="TM")
```

We now have a single data.frame with one ndvi column as before, so there is no need to use `melt()` this time to reshape the data.frame. We can pass this directly to `ggplot()` to produce our time series plots.

```
> ggplot(data=ts, aes(x=date, y=ndvi)) +
+   geom_point() +
+   facet_wrap(~ class, nrow=3) +
+   theme_bw()
```

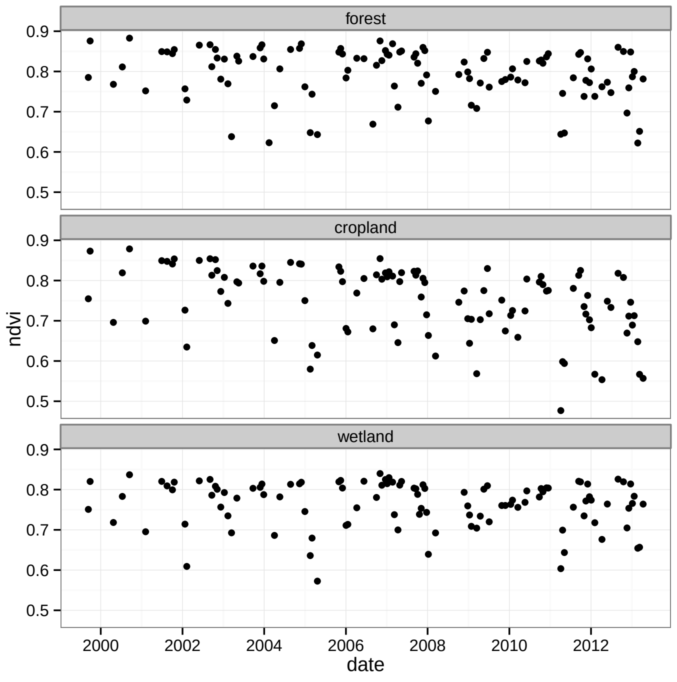

Figure 6: Time series for three land cover classes: forest, cropland and wetland. NDVI was calculated in this case from the mean NDVI per class per time step.

From Figure 6, we can see a similar relationship between the classes as we saw in Figure 5, except that in this case, the difference between cropland and forest is not so clear, especially at the beginning of the time series. Why might this be? Can you see any peculiar patterns in these time series plots (particularly that of cropland)? What do you think could be happening here? (Hint: the land cover classes we see in lulcGewata were defined statically for the year 2011).

Raster-Polygon joins using `extract()`

Raster to polygon spatial joins can similarly be done using the `extract()` function. We will explore this function further in the exercise, but the prinicple is similar to that of the `zonal()` function in the sense that summary statistics for a region in space can be calculated based on this join operation.

4 Exercise

To qualitatively identify regions with possible changes, multi-temporal RGB composites are often used. If three images from the same area acquired at times t1, t2 and t3 are plotted in a RGB composite, resulting colour combinations could be interpreted as possible changes. For example, red regions indicate higher values in the red channel, and lower values in the green and blue channels, which might be due to a change after the first time point. Similarly, yellow regions indicate higher values in the red and green channels, and lower values in the blue channel possibly due to a change between the 2nd and 3rd time points. As such, multi-temporal RGB composite plots are useful qualitative tools in identifying changes over time.

In the raster package, there is a useful function for plotting RGB composites: `plotRGB()`. The function takes a multi-layered raster object (ie. either a rasterBrick or rasterStack) as its first argument, followed by the indices of the layers (bands) to be plotted in the red, green and blue channels, respectively. A stretch (linear or histogram) can optionally be performed. See `?plotRGB()` for more information.

Calculate three new raster layers from the tura rasterBrick based on the mean or median NDVI for years 2000, 2005 and 2010. Put these into a new rasterBrick with nlayers=3. Produce a plot of these layers with a common scale (using either `plot()` or `levelplot()`). Then produce an RGB composite with these three layers such that NDVI-2000 is in the red channel, NDVI-2005 is in the green channel and NDVI-2010 is in the blue channel (hint: when plotting this composite, you will need to use the `stretch="hist"` argument).

From this composite plot, identify two change regions based on the colour combinations resulting from the compositing: (1) possible change between 2000 and 2005 and (2) possible change between 2005 and 2010. To verify these changes, extract a time series vector for each of these regions by using the `drawPoly()` command to interactively draw two SpatialPolygons objects. Use these objects to extract mean or median NDVI for each layer of the tura brick using a polygon-raster spatial join (hint: use `extract()`).

In your exercise, provide the following:

- a `plot()` or `levelplot()` of the three mean or median layers from 2000, 2005 and 2010
- an RGB composite plot of the three layers as follows: 2000 (red), 2005 (green) and 2010 (blue). Overlay the two SpatialPolygon objects on this plot (hint: use the argument `add=TRUE` when plotting the polygons).
- time series plots (preferably a facet_wrap plot using `ggplot()`) for each of the two chosen regions.