bfastSpatial

Loïc Dutrieux, Ben DeVries, Jan Verbesselt

October 7, 2014

1 Preamble

The `bfastSpatial` provides utilities to performs change detection analysis on time-series of spatial gridded data, such as time-series of remote sensing images (Landsat, MODIS and the likes). The tools provided by `bfastSpatial` allows a user to perform all the steps of the change detection workflow, from pre-processing raw surface reflectance Landsat data, inventorying and preparing them for analysis to the production and formatting of change detection results. The present document aims at providing guidance to the users of `bfastSpatial` by detailing every steps of the process.

2 Set up your system

2.1 Windows

There are no specific system requirements for Windows, besides having (a recent version of) R installed; installing the `bfastSpatial` package will automatically install the required dependencies.

2.2 Linux & Mac

Some pre-processing functions require the `rgdal` package, which is a binding between the gdal library and R. For `rgdal` to work on Linux and Mac systems, gdal should be installed and properly set-up. For that, please refer to the web page of the gdal project as well as rgdal.

2.3 Installing bfastSpatial

The package can be installed directly from github using devtools, provided you have git already installed on your system:

```
library(devtools)
install_github('dutri001/bfastSpatial')
# load the package
library(bfastSpatial)
```

3 Downloading Landsat data from Earth Explorer

Working with Landsat time-series often involves downloading large amounts of data. The full Landsat archive for one path/row may very well exceed 500 scenes in some parts of the world. The most common way to access the data is via the Earth Explorer platform. You need to have an account before being able to request data to be pre-processed.

3.1 Placing your order

USGS is planning to support the delivery of data in different formats (HDF, GeoTiff, binary). GeoTiff is a common and easy to handle format of raster data and should be your preferred choice if you are working in Windows.

3.2 Downloading the data

Once you receive the e-mail from USGS notifying you that your order has been pre-processed, you will be able to proceed to the download of the data. Because the amount of scenes is usually large, the use of a download manager is highly recommended. Two options:

- The Bulk Download Application
- The DownloadThemAll plugin of the firefox browser

4 Data pre-processing

The `bfastSpatial` package includes utilities to pre-process Landsat surface reflectance data and prepare them for subsequent analysis with the change detection functions such as `bfmSpatial`. The overall Landsat pre-processing requires to:

- Extract data from the tar.gz archive
- Calculate Vegetation Indices from surface reflectance bands (when not provided by USGS)
- Crop the data to a desired spatial extent
- Apply one of the cloud/land mask supplied with the data
- Create a the spatio-temporal object to be used in subsequent analyses

The two important functions to perform the above mentioned steps are `processLandsat` and `processLandsatBatch`. The former is a wrapper that performs the above steps sequentially, automatically generating file names and deleting unnecessary intermediary outputs. The latter (`processLandsatBatch`) allows a user to run `processLandsat` in batch mode, over several scenes, with parallel support. Several vegetation indices are supported by both `processLandsat` and `processLandsatBatch`. Let's illustrate that with an example. We added subsets of Landsat data to the package in order to illustrate the pre-processing steps. The test datasets are stored under the `external/` directory of the installed package. To list the content of that directory, run the following comand.

```
list.files(system.file('external', package='bfastSpatial'), full.names=TRUE)
```

Using the `full.names=TRUE` argument allows you to directly use the elements of the list create without having to set working directories.

Once the vegetation index layers have been produced for several dates, they can be stacked, in order to create a multilayer raster object, with time dimention written in the file as well. The function to perform this operation on Landsat data is the `timeStack` function. By simply providing a list of file names or a directory containing the files, the function will create a multilayer object and directly parse through the file names to extract temporal information from them and write that information to the object created.

5 Data Inventory

A number of functions are available in the bfastSpatial package to help keep an inventory of data in a raster time series stack. These functions range from basic scene information (`getSceneinfo()`) to summary of pixel values per year in the time series (`annualSummary()`).

The following functions are included in the Data Inventory module:

1. `getSceneinfo()`
2. `countObs()`
3. `summaryBrick()`
4. `annualSummary()`

5.1 Basic Scene Information: `getSceneinfo()`

`getSceneinfo()` allows the user to list the information contained within a scene ID. Currently, only Landsat scene ID's are supported. For example, the scene ID "LE71700552007309ASN00" tells us that the scene is from the Landsat 7 ETM+ sensor ('LE7'), path-row 170-55 ('170055') and was acquired on the 309th day of the year 2007 ('2007309'). Calling the following code will give a data.frame with one row showing all of this information:

```
getSceneinfo('LE71700552007309ASN00')

##                         sensor path row       date
## LE71700552007309 ETM+ SLC-off  170  55 2007-11-05
```

When working with Landsat data, it is a good idea to assign and keep these sceneID's as layer names (see `?raster::names`) so the relevant information is associated to each raster layer.

```
# show scene info from tura layers
data(tura)
head(names(tura))

## [1] "LE71700551999255AGS00" "LE71700551999271EDC00" "LE71700552000114SGS00"
## [4] "LE71700552000194EDC00" "LE71700552000258SGS00" "LE71700552001036SGS00"
```

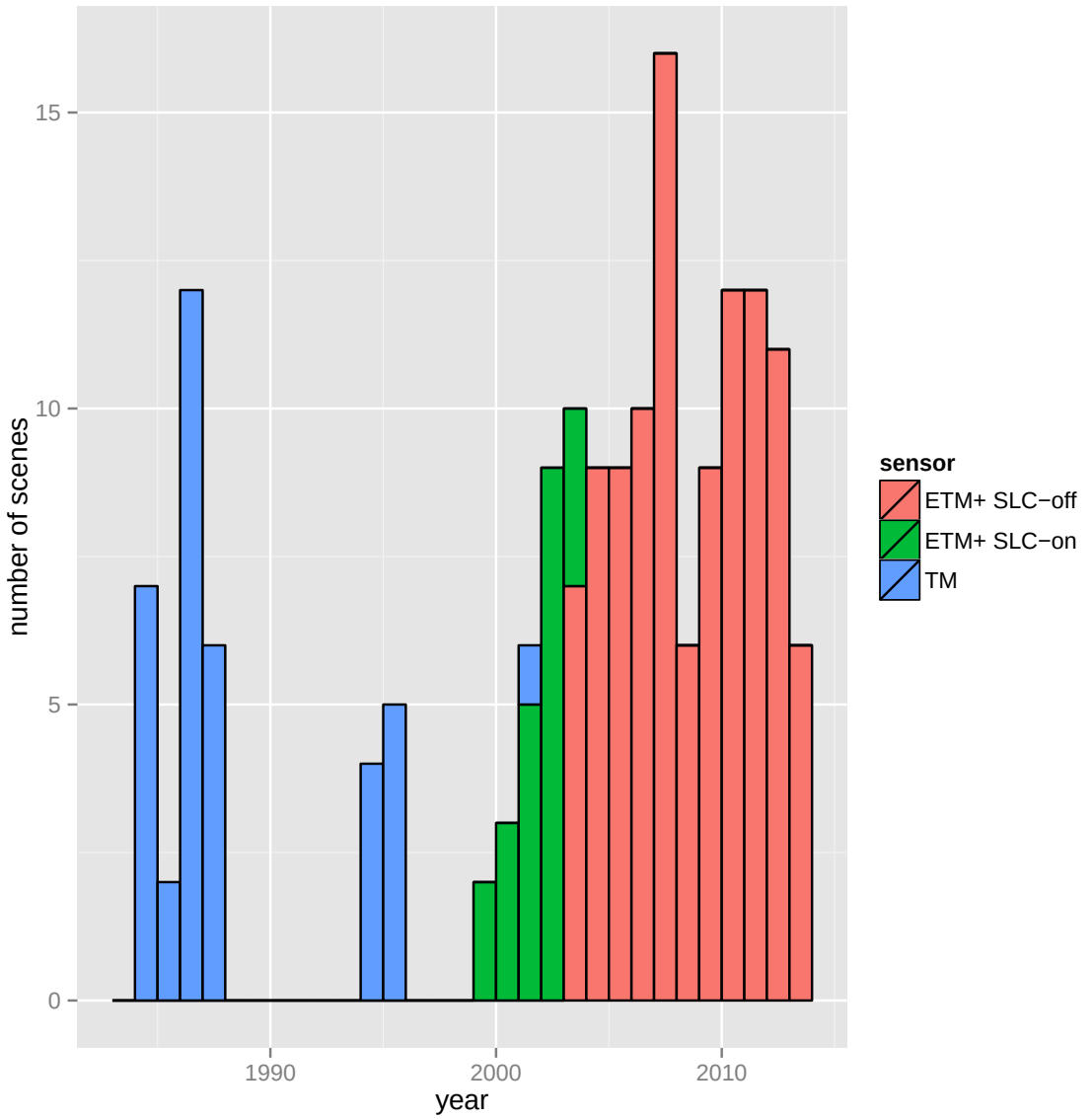
```
s <- getSceneinfo(names(tura))
head(s)

##                        sensor path row       date
## LE71700551999255 ETM+ SLC-on  170  55 1999-09-12
## LE71700551999271 ETM+ SLC-on  170  55 1999-09-28
## LE71700552000114 ETM+ SLC-on  170  55 2000-04-23
## LE71700552000194 ETM+ SLC-on  170  55 2000-07-12
## LE71700552000258 ETM+ SLC-on  170  55 2000-09-14
## LE71700552001036 ETM+ SLC-on  170  55 2001-02-05

# add a column for years and plot # of scenes per year
s$year <- as.numeric(substr(s$date, 1, 4))
hist(s$year, breaks=c(1984:2014), main="p170r55: Scenes per Year",
     xlab="year", ylab="# of scenes")
```

We can combine the dates and sensor information to get more of an idea of where our data are coming from.

```
library(ggplot2)
p <- ggplot(data = s, aes(x = year))
p <- p + geom_bar(aes(fill = sensor), binwidth=1, col="black")
p <- p + labs(y="number of scenes")
p
```

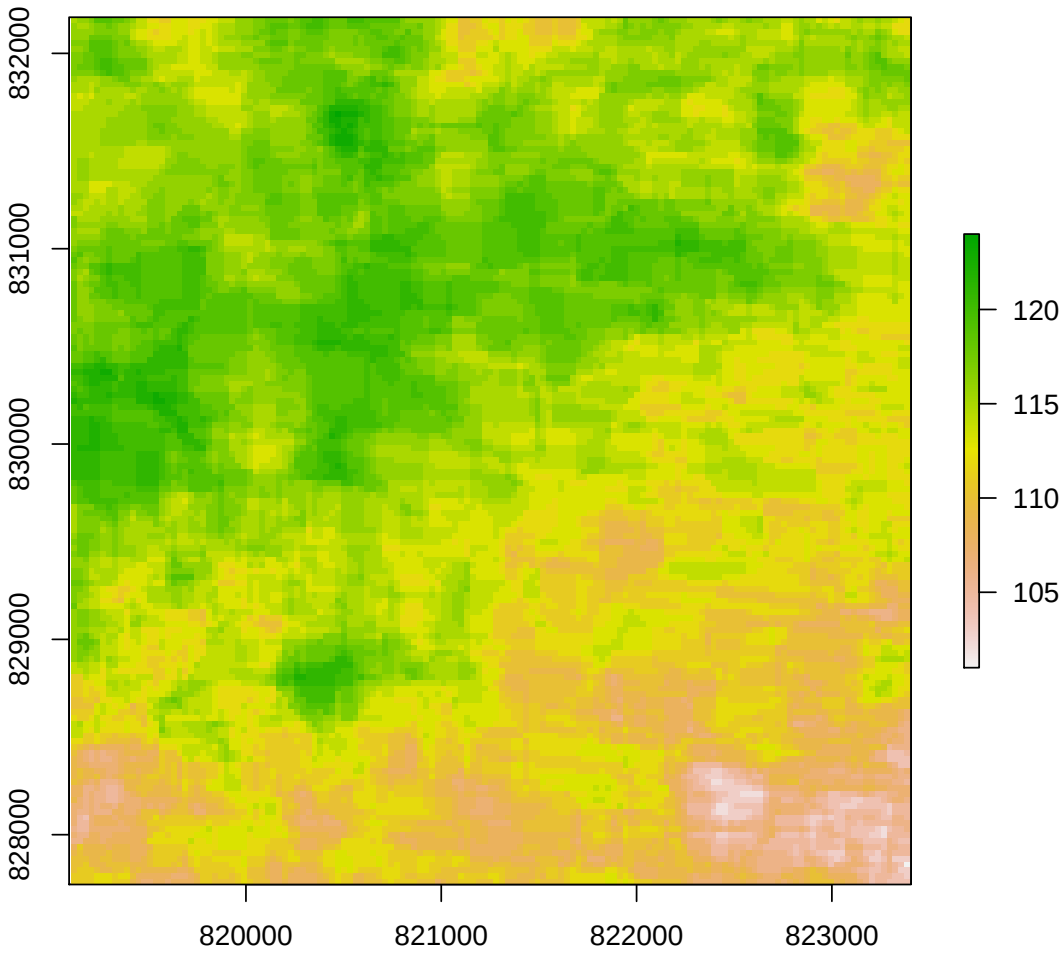


More examples can be found under `?getSceneinfo`. Many other functions in the bfastSpatial package rely on getSceneinfo to extract relevant scene information, such as acquisition dates to be passed to `bfmSpatial()` or `bfmPixel()`.

5.2 Valid Observations: `countObs()`

The number of available observations in a raster time series can be calculated by using `countObs()`. This function "drills" through pixels of a time series and counts the number of pixels with a non-NA value. Optionally, any other value can be supplied as a substitute for NA (e.g. the number of non-zero values per pixel can also be queried).

```
data(tura)
obs <- countObs(tura)
plot(obs)
```




```
summary(obs)

##         layer
## Min.      101
## 1st Qu.   112
## Median    114
## 3rd Qu.   117
## Max.      124
## NA's        0
```

Values can also be expressed as a percentage if `as.perc` is set to `TRUE`.

```r
# valid observations
obs <- countObs(tura, as.perc=TRUE)
summary(obs)
# \% NA per pixel
percNA <- 100 - countObs(tura, as.perc=TRUE)
plot(percNA, main="percent NA per pixel")
summary(percNA)
```

5.3 Summary Statistics: `summaryBrick()` and `annualSummary()`

In this section, we will calculate summary statistics for the Tura RasterBrick. First, general statistics for the entire brick can be calculated using **`summaryBrick()`**. In fact, **`countObs()`** is essentially a specific version of **`summaryBrick`**. **`summaryBrick()`** takes a RasterBrick as its first argument and a function as its second argument, and all others are optional arguments. Calculating the mean of each pixel is simple.

```r
meanVI <- summaryBrick(tura, fun=mean) # na.rm=FALSE by default
plot(meanVI)
```

The fact that there are no values in the result stems from the fact that an **`na.rm`** argument has not been supplied. By default, this is set to **`NULL`** to ensure that other functions that do not take an **`na.rm`** argument (like **`length`**, for example) can also be used as **`fun`**. Since there are many **`NA`** values scattered through our RasterBrick as a result of cloud masking, **`na.rm`** must be set to **`TRUE`** for functions like **`mean`**, **`median`** and other common stats functions.

```r
meanVI <- summaryBrick(tura, fun=mean, na.rm=TRUE)
plot(meanVI)
```

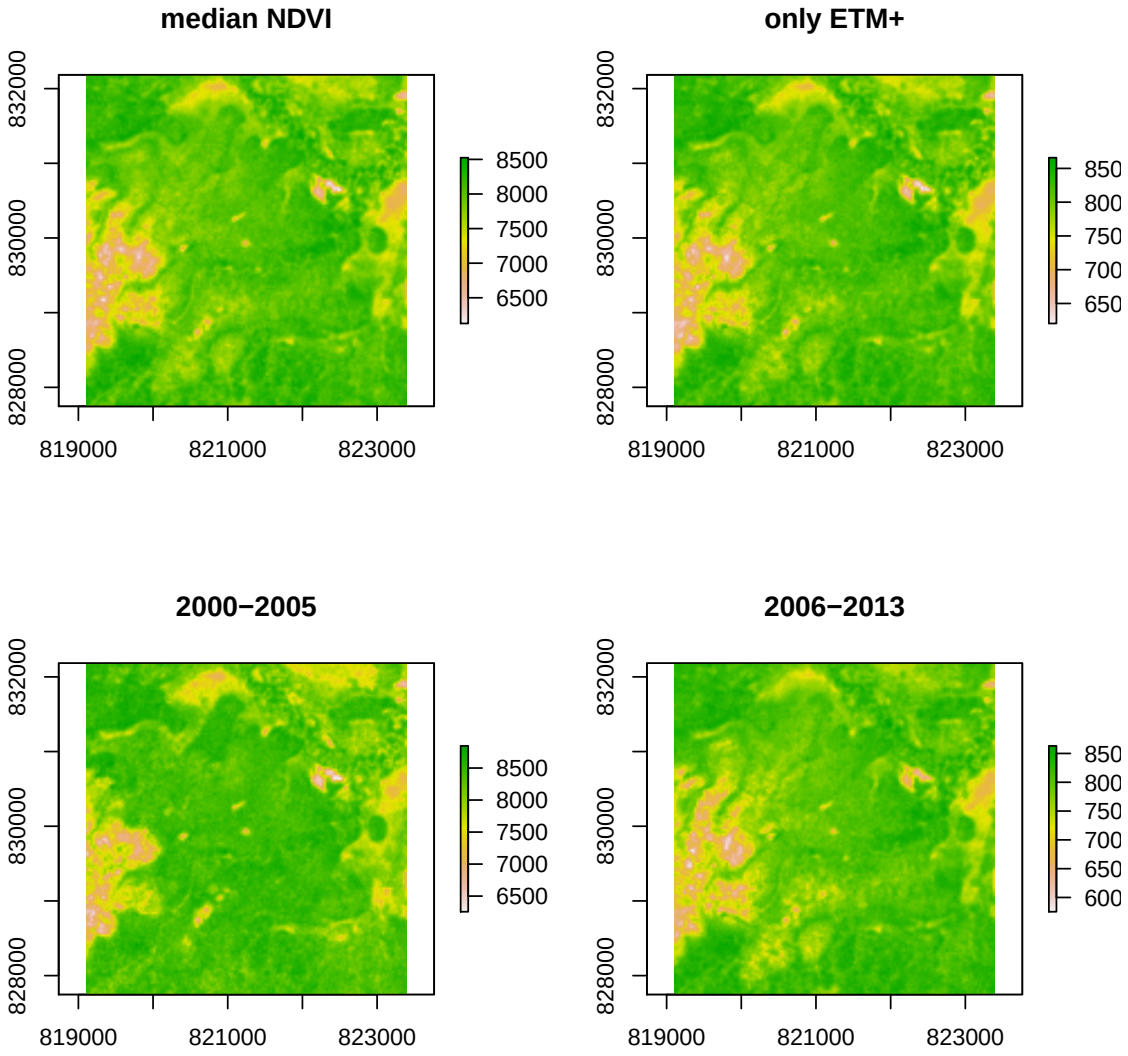
`summaryBrick` can also take custom functions, provided these functinos are vectorized - in other words, they should be applicable to a vector extracted from a single pixel time series (for example, by taking a single cell number of the Tura RasterBrick: **`x <- tura[500]`**). Suppose we want to know how many observations per pixel exceed a certain threshold. First, we should define a function that can count the number of elements in a vector that exceed this threshold, and then pass this to **`summaryBrick`**. In this example, we will use 7000 as our threshold (remember that the values are scaled by 10000, so this corresponds to an NDVI of 0.7).

```r
# define a function that takes a vector as an argument
checkThresh <- function(x){
    # first, get rid of NA's
    x <- x[!is.na(x)]
    # if there still values left, count how many are above the threshold
    # otherwise, return a 0
    if(length(x) > 0){
        y <- length(x[x > 7000])
    } else {
```

```
        y <- 0
    }
    # return the value
    return(y)
}
# pass this functino to summaryBrick
customStat <- summaryBrick(tura, fun=checkThresh)
plot(customStat, main = "# of observations where NDVI > 0.7")
```

Note that in this case we handled NA values within our function, so there was no need to specify a value for `na.rm`. `summaryBrick` has some optional arguments that allow constraints according to dates and sensor (currently only for Landsat data). `sensor` takes a character vector of sensors to constrain the data, while `minDate` and `maxDate` allow a date range to be set to constrain the data. In the following example, we will calculate median pixel values using different date and sensor ranges.

```
# median values for all layers
medVI <- summaryBrick(tura, fun=median, na.rm=TRUE)
# only ETM+ layers
medVI_ETM <- summaryBrick(tura, fun=median, na.rm=TRUE, sensor="ETM+")
# all layers between 2000 and 2005 (inclusive)
medVI_00_05 <- summaryBrick(tura, fun=median, na.rm=TRUE, minDate="2000-01-01",
                            maxDate="2005-12-31")
# all layers after 2005
medVI_06_13 <- summaryBrick(tura, fun=median, na.rm=TRUE, minDate=c(2006, 1))
# plot and compare
op <- par(mfrow=c(2, 2))
plot(medVI, main = "median NDVI")
plot(medVI_ETM, main = "only ETM+")
plot(medVI_00_05, main = "2000-2005")
plot(medVI_06_13, main = "2006-2013")
```




```
par(op)
```

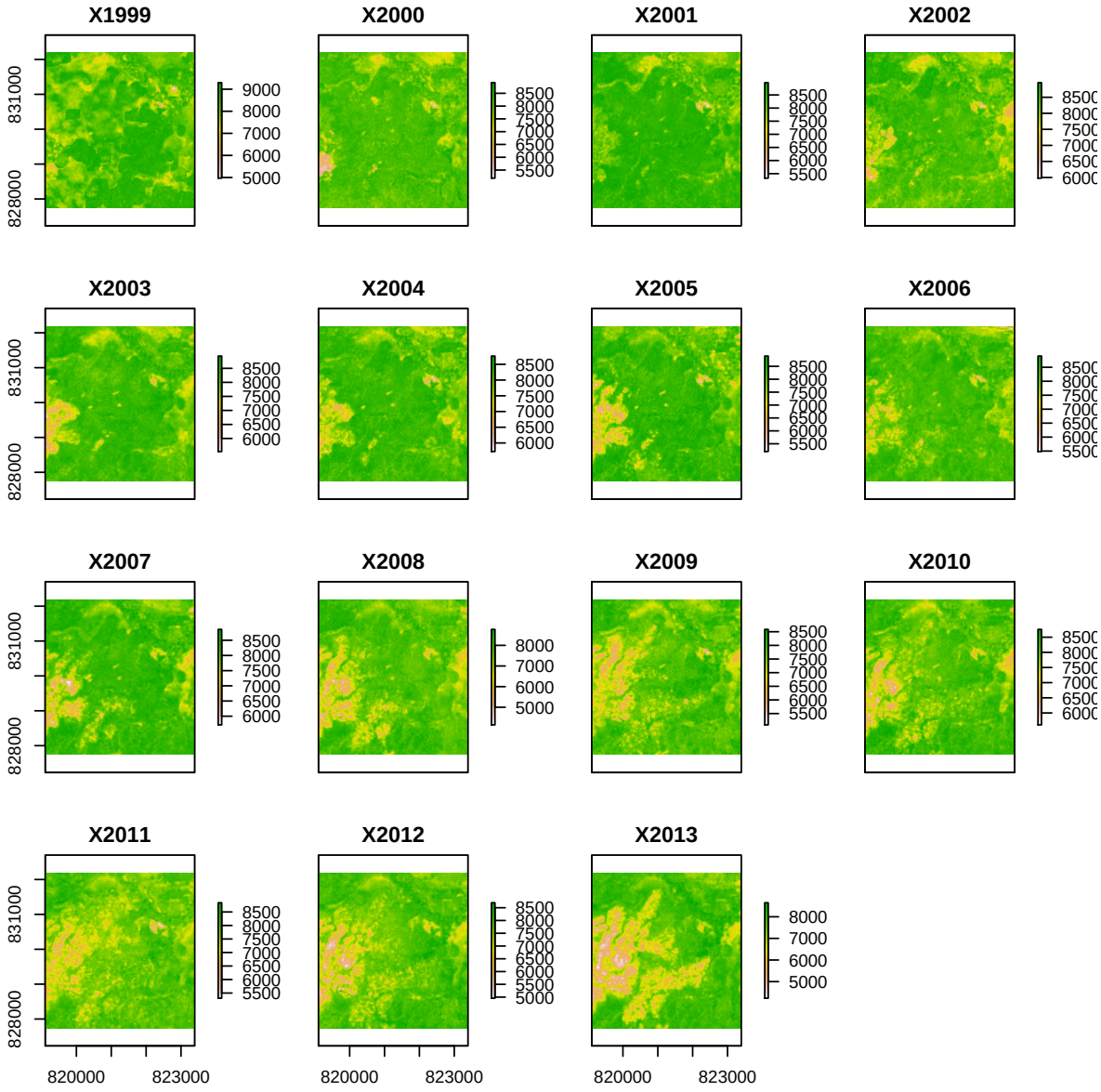
`minDate` and `maxDate` can be supplied in one of two ways: (1) as a character or date object in the format `"%Y-%m-%d"` (eg. "2013-01-31") or (2) as a numeric vector of length 2 representing the year and Julian day (e.g. `c(2013, 31)`).

Note that the Tura brick already has scene information included as layer names (see `names(tura)`). To run `annualSummary` on RasterBricks where no scene information is available, a dates vector *must* be supplied to be able to divide the layers into years. This vector is also required for `summaryBrick` when constraining the calculation by date using `minDate` and `maxDate`.

`annualSummary` works similarly to `summaryBrick`, but provides a statistic for each year of the time series. At this time, this function is only applicable to Landsat data, and the scene IDs must be supplied either as `names(x)` or as a separate character vector to `sceneID`. In

this sample, we will again calculate median NDVI on all ETM+ data, but this time we are interested in seeing the *annual* median NDVI value.

```
annualMed <- annualSummary(tura, fun=median, sensor="ETM+", na.rm=TRUE)
plot(annualMed)
```



Both `summaryBrick()` and `annualSummary()` can pass additional arguments to `mc.calc()`, meaning that the result can be saved to file with a `filename` argument, and multiple cores can be used in the calculation with a `mc.cores` argument.

From the summary RasterBrick above, we can see that this is a very dynamic area, with some of the biggest forest changes occurring from 2009 until 2013. With the help of `bfastmonitor`, we will look at ways to quantify and map these changes at high temporal resolution.

6 Spatial BFASTMonitor

6.1 Working with pixels: `bfmPixel`

Working with `bfastmonitor` (BFM) can be intimidating at first, especially given the number of parameters that need to be set (or at least considered) before running the algorithm. It is important to understand these parameters, as not all raster time series are equal. Some time series may have frequent and large temporal gaps (as can be expected in some tropical forest), while others are temporally dense. These differences may call for slightly different approaches in implementing BFM.

`bfmPixel` is a function which queries individual pixels in a raster time series and runs bfm over that pixel's time series. To run `bfmPixel` in interactive mode, one of the layers of the time series needs to first be plotted.

```
# load tura RasterBrick
data(tura)
# plot the 6h layer
plot(tura, 6)
# run bfmPixel() in interactive mode with a monitoring period
# starting @ the 1st day in 2009
bfm <- bfmPixel(tura, start=c(2009, 1), interactive=TRUE)
```

Inspecting the output of this function, you will see that a list of length 2 has been outputted. First, a list of class `bfastmonitor` is output under `$bfm`. Second, an integer indicating the cell number that has been clicked is output under `$cell`. The output is formatted this way so that if `bfmPixel` is run in interactive mode, a follow-up analysis can be done on that same cell without having to guess which cell was clicked before. In the above example, the cell number can be retrieved simply by typing `bfm$cell` in the console.

Let's run `bfmPixel` again, but this time indicating a specific cell number instead of using interactive mode. Note that by default `interactive=FALSE`, so there is no need to specify this if a value for `cell` is given.

```
targcell <- 3492
bfm <- bfmPixel(tura, cell=targcell, start=c(2009, 1))
# inspect and plot the £bfm output
bfm$bfm

##
## BFAST monitoring
##
## 1. History period
## Stable period selected: 1984(285)--2008(359)
## Length (in years): 24.202740
## Model fit:
## (Intercept)       trend  harmoncos1  harmoncos2  harmoncos3  harmonsin1
##  7577.91041     0.09905  -297.47432    71.66341    52.05172  -393.96794
##  harmonsin2  harmonsin3
##     5.38914   -36.69532
```

```
## R-squared: 0.360894
##
##
## 2. Monitoring period
## Monitoring period assessed: 2009(1)--2013(101)
## Length (in years): 4.273973
## Break detected at: 2009(298)

plot(bfm$bfm)
```

Specifying a cell number gives the advantage of being able to test different parameters on the same pixel time series to see what impact these parameters have on the sensitivity of the algorithm. Note that a plot is immediately produced if `plot = TRUE`.

```
# use a harmonic model only
bfm1 <- bfmPixel(tura, cell=targcell, start=c(2009, 1),
                 formula=response~harmon, plot=TRUE)
# same, but with an order of 1
bfm2 <- bfmPixel(tura, cell=targcell, start=c(2009, 1),
                 formula=response~harmon, order=1, plot=TRUE)
# only trend
bfm3 <- bfmPixel(tura, cell=targcell, start=c(2009, 1),
                 formula=response~trend, plot=TRUE)
```

Check out `?bfastmonitor` for more information on the parameters that are included in the method.

A number of additional options are provided in `bfmPixel`. Specifying a value for `monend` will trim the time series after that date (thus limiting the monitoring period to a given period). This simple modification can have an impact on the resulting change magnitude, which is computed as the median residual between the predicted and observed values within the monitoring period.

```
# bfmPixel using a 1-year monitoring period
bfm4 <- bfmPixel(tura, cell=targcell, start=c(2009, 1),
                 monend=c(2010, 1), plot=TRUE)
```

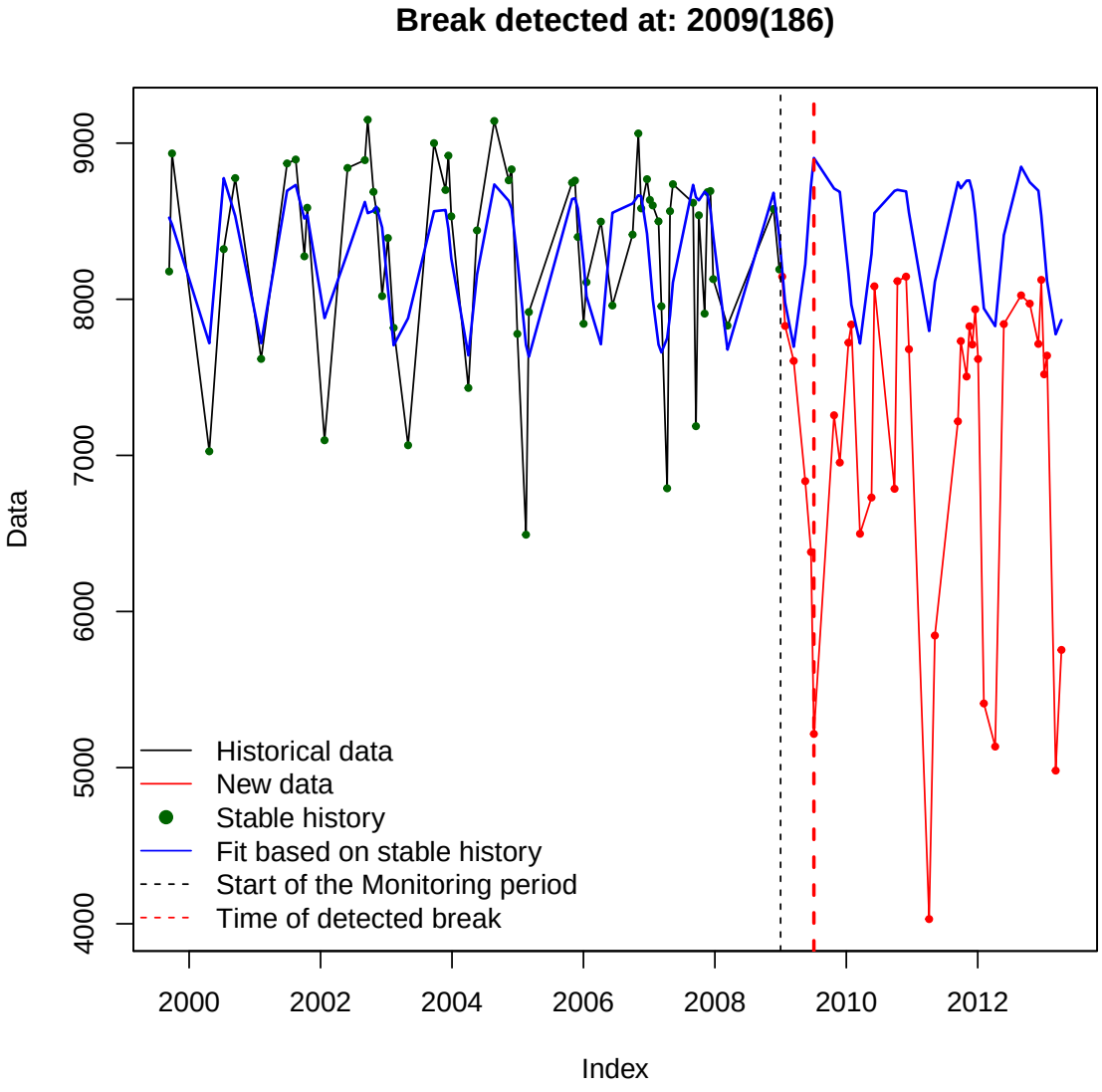
Finally, for Landsat time series the analysis can be limited to data of a specific sensor. The `sensor` parameter currently accepts the following possible character values:

- `"ETM+"`
- `"ETM+ SLC-on"`
- `"ETM+ SLC-off"`
- `"TM"`

Note that in our previous runs of `bfmPixel`, there is what seems to be a bias between TM data from the 80's and 90's (which can be found throughout the scene). This has a large

effect on our model, especially when a trend is included. A very quick way to avoid this bias is to include only data from the ETM+ sensor, which was launched in 1999.

```
# apply bfmPixel only on ETM+ data
bfm5 <- bfmPixel(tura, cell=targcell, start=c(2009, 1),
                 sensor="ETM+", plot=TRUE)
```



6.2 Working with raster time series: `bfmSpatial`

Now that we have done some pixel-based testing, it is time to apply `bfastmonitor` over an entire raster time series. This will allow us to pinpoint the location and timing of changes in our study area. In the previous section, we saw that there was a large gap in the 1990's in the tura raster time series, and given the acceptable data density, we could just limit the analysis to ETM+ data. We also saw that a harmonic-trend model with an order of 1 seemed

to be the most reasonable time series model for the tura dataset. We will pass these same arguments to `bfmSpatial`. At the same time, we will calculate how long the process takes by wrapping it in a `system.time` call.

```
# run bfmSpatial over tura for a monitoring period of 2009 -
t1 <- system.time(
    bfm <- bfmSpatial(tura, start=c(2009, 1), order=1)
    )
# plot the result
plot(bfm)
```

Luckily for us, tura is a very small RasterBrick, and processing is not incredibly time consuming. Realistically, analyses are done over significantly larger areas (possibly extending beyond a single scene), so running `bfmSpatial` on such large areas quickly becomes a very time-consuming task. To assist in this, multi-core support has been built into `bfmSpatial`, using the `parallel` package. Let's run the same line of code again, but this time specify a value for `mc.cores`, which will distribute the process over that many cpus. Again, we will wrap the code in a `system.time` call to test the speed of the processing.

```
t2 <- system.time(
    bfm <- bfmSpatial(tura, start=c(2009, 1), order=1, mc.cores=8)
    )
# compare processing time with previous run
t1 - t2
```

7 Post-Processing of BFM output

Three raster layers are output by `bfmSpatial`:

1. breakpoint timing - in decimal year format
2. change magnitude - the medial residual between expected and observed values in the monitoring period
3. error flag - possible values of 1 if an error has been encountered in a particular pixel, and NA where the algorithm was successful (or a mask had already been applied previously)

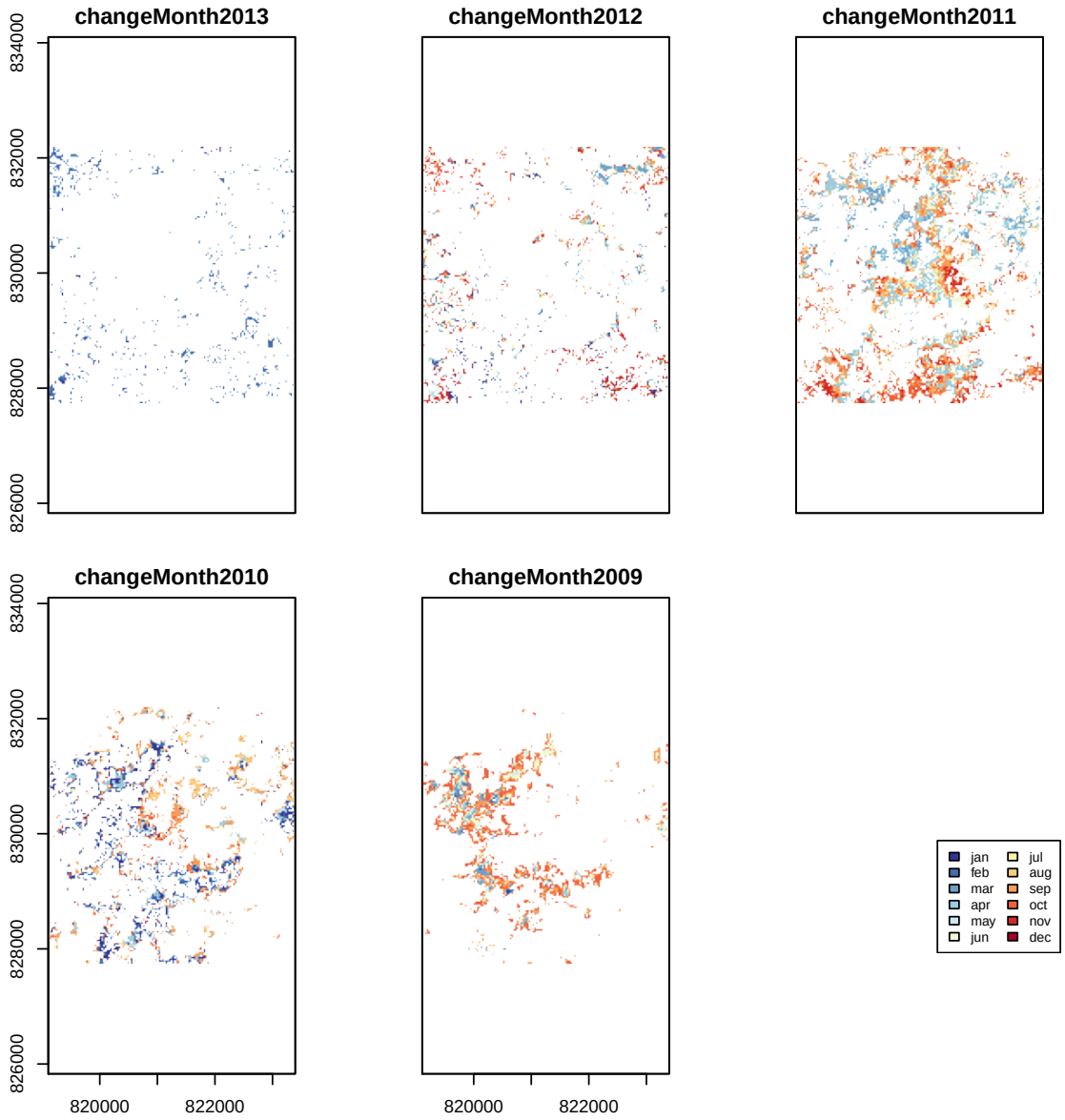
The `bfastSpatial` package has some simple functions for 'unpacking' the `bfmSpatial` results. `bfmChange` simply extracts the first layer and saves it to a separate object.

```
# extract change raster
change <- bfmChange(bfm)
```

The change raster gives breakpoint timing values in decimal years. A better way to map change timing from these results may be to show the month of change (the validity of this depends on the temporal density of there input time series). `changeMonth` is a convenient way to convert breakpoint values to change months. The result is one or more raster layers with

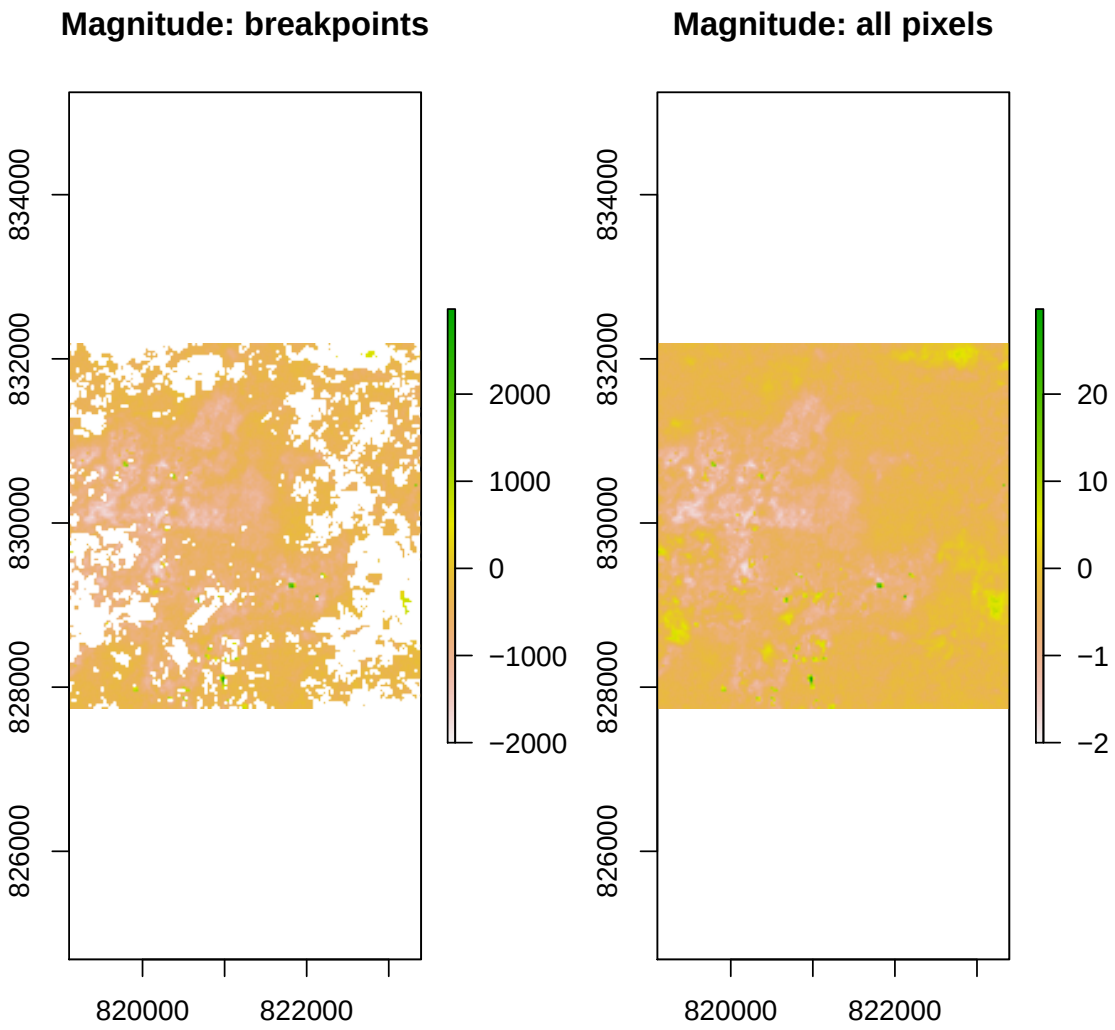
integer values between 1 and 12, representing the months of the year. In cases where multiple years are represented in the change raster, these are split into raster layers representing one layer per year.

```
months <- changeMonth(change)
# set up labels and colourmap for months
monthlabs <- c("jan", "feb", "mar", "apr", "may", "jun",
               "jul", "aug", "sep", "oct", "nov", "dec")
library(RColorBrewer)
cols <- rev(brewer.pal(11, "RdYlBu"))
# interpolate colormap (since max # of colours for "RdYlBu" is 11)
cols <- colorRampPalette(cols)(12)
plot(months, col=cols, breaks=c(1:12), legend=FALSE)
# insert custom legend
legend("bottomright", legend=monthlabs, cex=0.5, fill=cols, ncol=2)
```



`bfmMagn` does the same as `bfmChange` to the second layer, but adds some other options (masking by breakpoint, applying a threshold). Magnitude is computed as the median residual between expected and observed values throughout the *entire* monitoring period, meaning that all pixels are assigned a value regardless of whether or not a breakpoint has been detected. If we are only interested in looking at pixels where a breakpoint has been detected, we can use the `change` argument in the `bfmMagn`, which takes a raster of breakpoints to use as a filter.

```
# extract magn raster, and include only change pixels
magn <- bfmMagn(bfm, change=change)
# compare with 'original' magn raster
op <- par(mfrow=c(1, 2))
plot(magn, main="Magnitude: breakpoints")
plot(bfm[[2]], main="Magnitude: all pixels")
par(op)
```



Note that the magnitude values are in the same scale as the input data, which had been scaled by 10000. Scaling back can be done by simple raster algebra, or with the `raster::calc` function to enable writing to file.

```
magn <- bfmMagn(bfm, change=change) / 10000
```

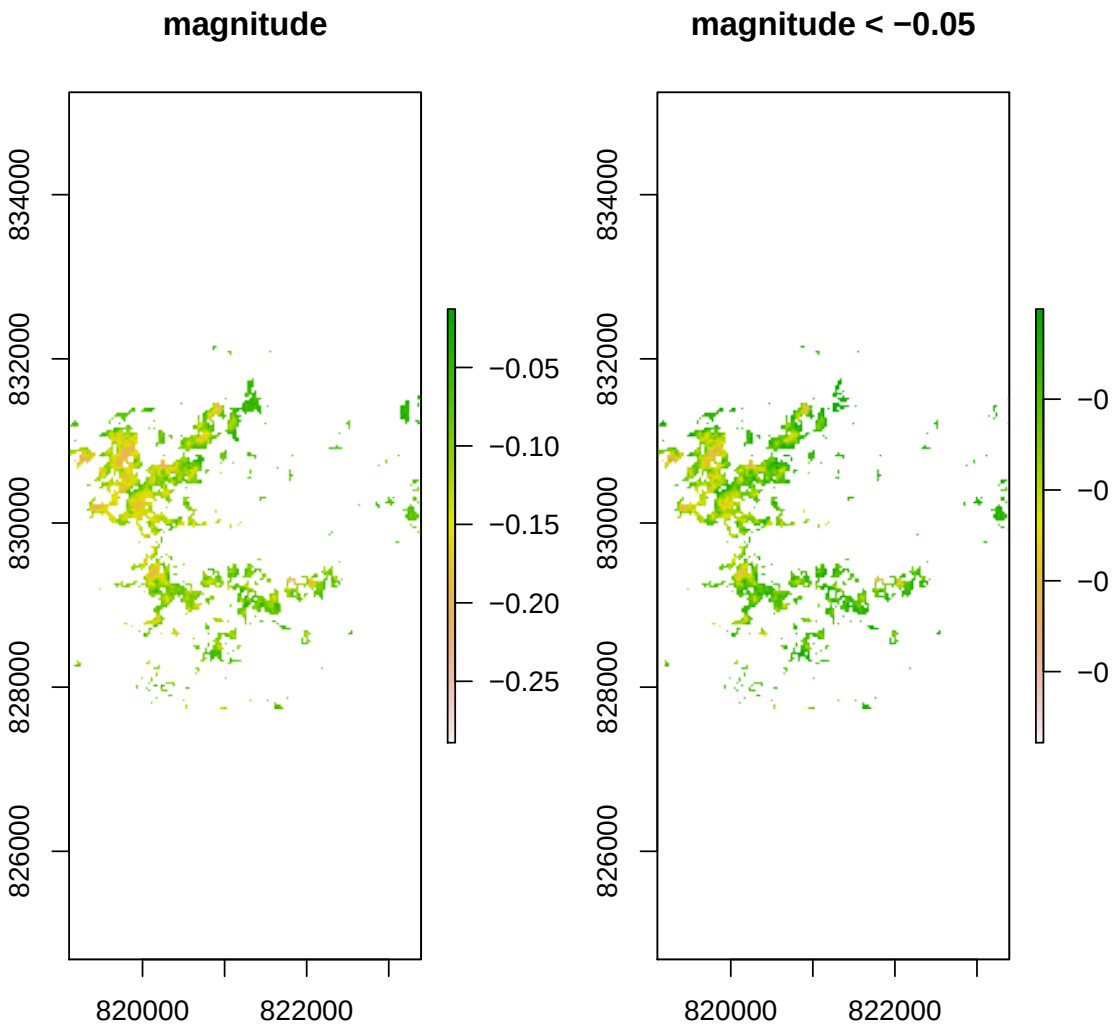
The magnitude could be used to discriminate between actual deforestation and other factors that might give rise to breakpoints (such as noisy data). However, interpreting magnitude is complicated by the fact that follow-up dynamics can affect the magnitude value, since it is computed throughout the entire monitoring period. To circumvent this (not entirely), we will re-run `bfmSpatial` using a 1-year monitoring period, under the assumption that this monitoring window is too short for post-change dynamics to significantly affect the magnitude value. To do this, we will include the `monend` argument of `bfmSpatial`, which will limit the analysis to a specific time period. In this example, we will use a 2009-2010 monitoring period

with all other parameters the same as above.

```
bfm09 <- bfmSpatial(tura, start=c(2009, 1), monend=c(2010, 1), order=1)
# extract change
change09 <- bfmChange(bfm09)
# extract and rescale magnitude
magn09 <- bfmMagn(bfm09, change=change09) / 10000
```

Suppose we know from field observations or other reference data that breakpoints with NDVI magnitudes of -0.05 are associated with high probability of detected actual deforestation. We can easily apply this threshold to our magnitude raster, or even pass this threshold as an additional argument to `bfmMagn`. In the later case, take care with the scaling of the data - the original `bfmSpatial` results are still in their "raw" format. In this case, our threshold should be accordingly scaled. In this example, we will scale the final magnitude back, as above.

```
# extract and rescale magnitude and apply a -500 threshold
magn09thresh <- bfmMagn(bfm09, change=change09, thresh = -500) / 10000
# compare
op <- par(mfrow=c(1, 2))
plot(magn09, main="magnitude")
plot(magn09thresh, main="magnitude < -0.05")
par(op)
```



A careful inspection of these results will show that applying the magnitude filter has changed the results somewhat, but since there isn't a great deal of noise in this example, the results were not dramatic. Another way we can clean up our results is to get rid of tiny clumps of change pixels that are not likely to represent actual change. For example, a simple raster sieve can eliminate single-pixel change clumps, which is justifiable given the fact that a Landsat pixel represents $900m^2$, and detectable forest changes are expected to occur at larger scales. Note that the threshold has to be set on a "less-than" basis, so we can use the area of 2 pixels as our cut-off.

```
magn09_sieve <- areaSieve(magn09thresh, thresh=1800)
```

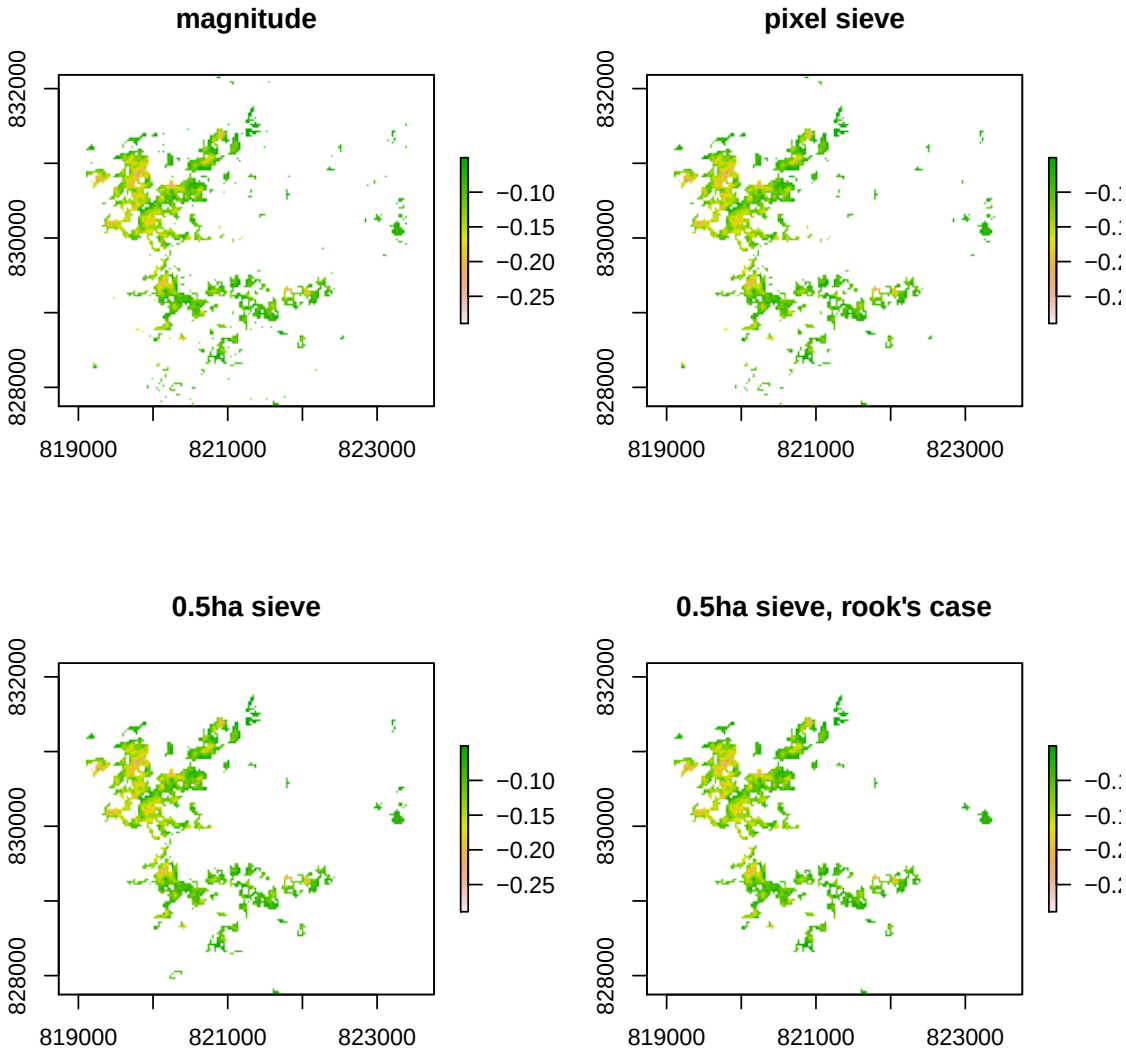
With `areaSieve`, this can be taken a step further, and an area threshold can be supplied to apply to the raster. `areaSieve` only supports thresholds in m^2, so be sure to do the appopriate conversions before appling the sieve. The default value is 5000, which corresponds

to the 0.5ha forest definition often applied in national assessments.

```
magn09_areasieve <- areaSieve(magn09thresh)# default thresh = 5000
```

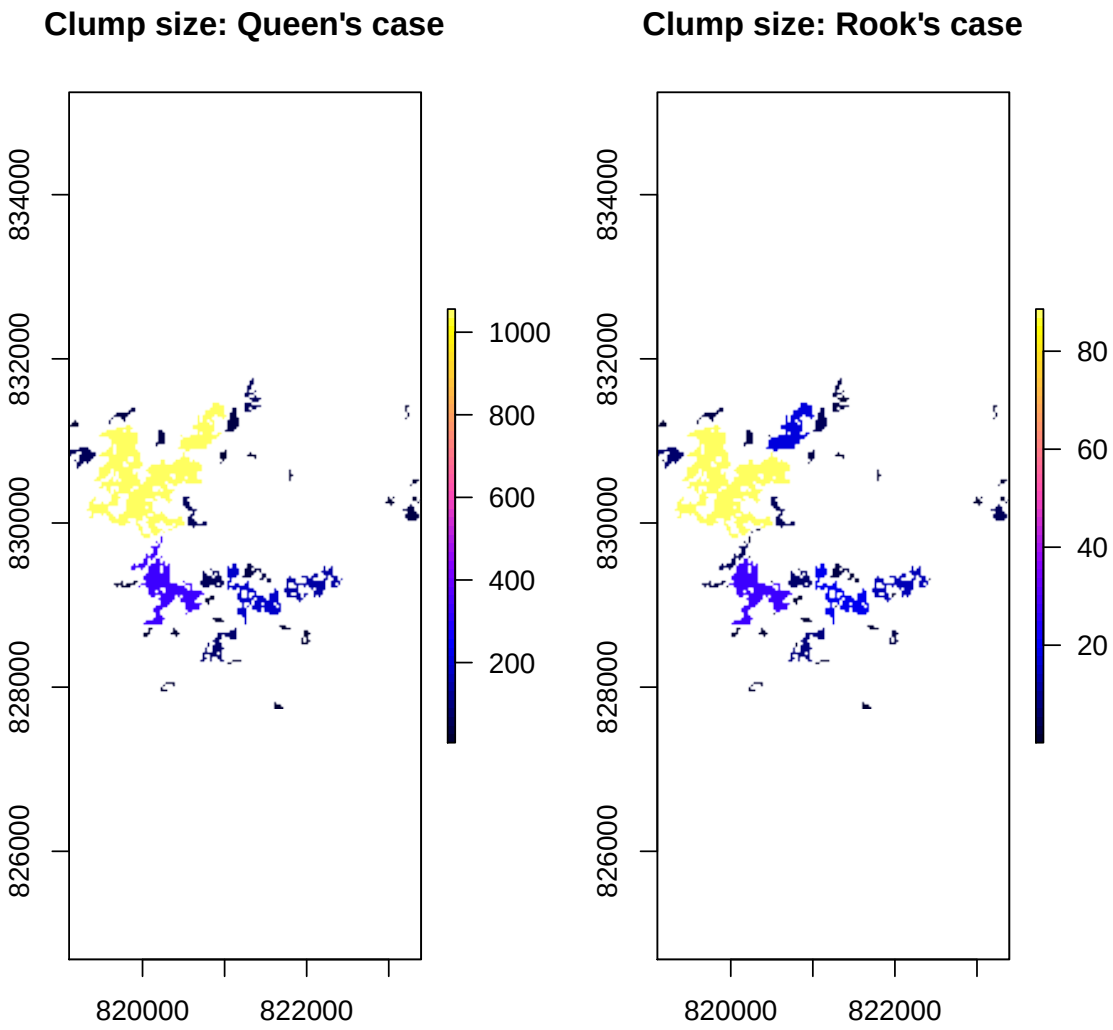
`areaSieve` passes additional arguments to `raster::clump`, which allows for considerations of direction when defining pixel neighbourhoods. `clump` adheres to the principle of "queen's case" versus "rook's case". In the former, pixels in all directions (default: `directions = 8`) are considered when defining clumps, whereas in the latter case, only pixels directly adjacent (excluding diagonals) are considered (`directions = 4`).

```
magn09_as_rook <- areaSieve(magn09thresh, directions=4)
# compare all magn rasters
op <- par(mfrow=c(2, 2))
plot(magn09thresh, main="magnitude")
plot(magn09_sieve, main="pixel sieve")
plot(magn09_areasieve, main="0.5ha sieve")
plot(magn09_as_rook, main="0.5ha sieve, rook's case")
par(op)
```



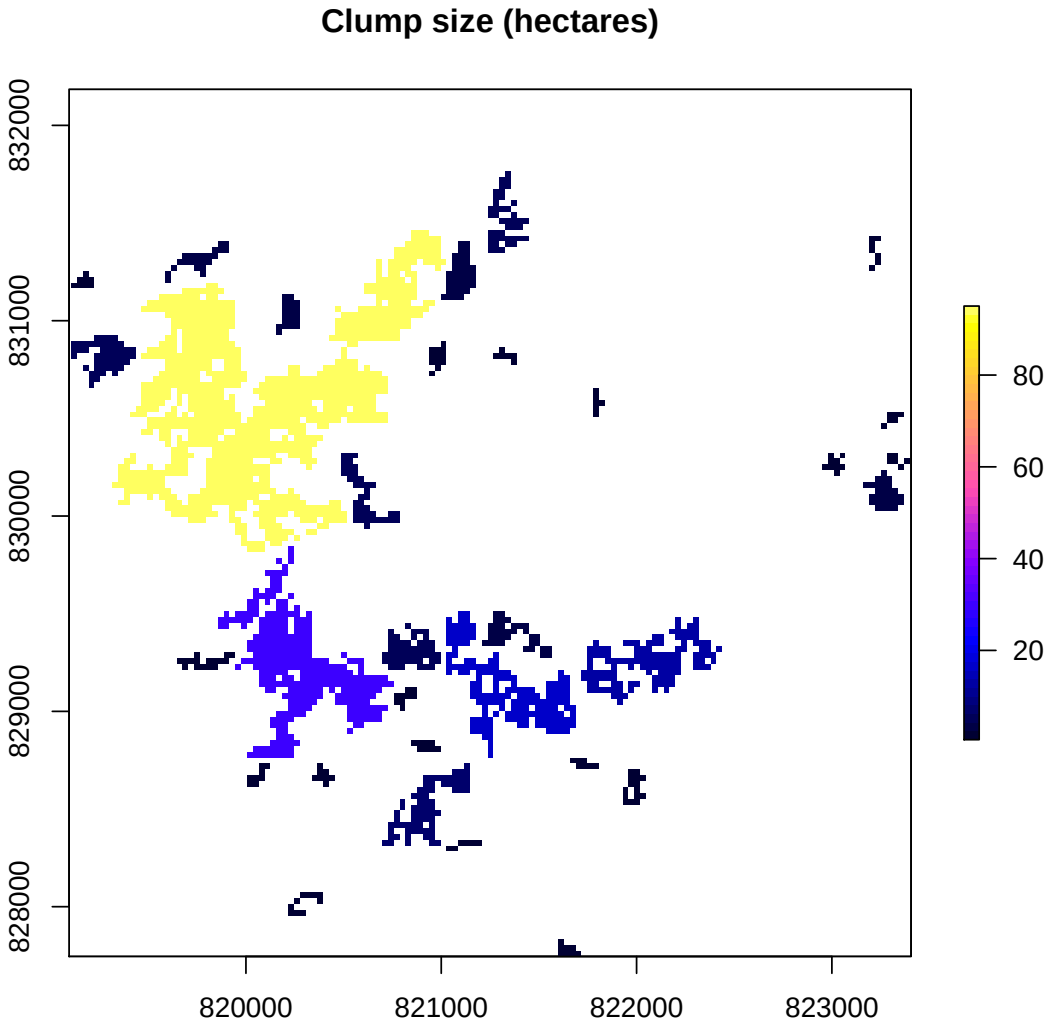
Once we have applied the desired filters and sieves to our bfm magnitude raster, we may want to calculate area statistics. This can be done with another simple function: `clumpSize`. This function is also based on the `raster::clumps` function and computes the size of each pixel clump and assigns that value back to each pixel in that clump. As in `areaSieve`, the queen's and rook's cases can also apply by adjusting the `directions` argument.

```
changeSize_queen <- clumpSize(magn09_areasieve)
changeSize_rook <- clumpSize(magn09_areasieve, directions=4)
# compare
op <- par(mfrow=c(1, 2))
plot(changeSize_queen, col=bpy.colors(50), main="Clump size: Queen's case")
plot(changeSize_rook, col=bpy.colors(50), main="Clump size: Rook's case")
par(op)
```



These clump sizes are in number of pixels, which might not be very useful when communicating our results to other parties. The `f` argument in `clumpSize` allows us to convert these to a useful area measure. In the case of Landsat, we know that each pixel covers $900m^2$, and that one hectare equals $10000m^2$. `clumpSize` can perform this conversion if a value for `f` is supplied.

```
changeSize <- clumpSize(magn09_areasieve, f=900/10000)
plot(changeSize, col=bpy.colors(50), main="Clump size (hectares)")
```



In order to calculate statistics of the changed area, it may be tempting to simply run `raster::summary`. But this function will produce misleading results, as *each pixel* has been assigned the value of its respective clump size (rather than representing each clump as a singular object, as would be the case with a SpatialPolygons object). To circumvent this, a `stats` argument is included in `clumpSize`, which outputs an additional summary table based on clumps rather than on individual pixels. If `stats=TRUE`, rather than outputting a single raster layer, `clumpSize` returns a list with 2 objects: (1) a raster layer as above; and (2) a summary table (matrix) showing mean, min, and all quartiles of the clump sizes.

```
changeSize <- clumpSize(magn09_areasieve, f=900/10000, stats=TRUE)
print(changeSize$stats)

##         clump size
## Mean         6.266
```

```
## Min.          0.540
## 1st Qu.       0.720
## Median        1.215
## 3rd Qu.       3.735
## Max.         95.040
```

We can say that based on our analysis, the average size of change units in Tura in 2009 is estimated to be 6.3 hectares. In this case, the minimum is not suprising, since we had already applied an area sieve using 0.5 hectares as our threshold.