Phenopix

Gianluca Filippa, Edoardo Cremonese, Mirco Migliavacca, . . .

2015-01-26

This vignette aims at illustrating the main features of the package `phenopix`. This package was designed for processing digital images of the vegetation cover in order to compute vegetation indexes that can be in turn used to track the seasonal development of the vegetation. The analysis can be run on one or more portions of the image (so called regions of interest, ROIs). Regions of interest can be of any polygonal shape. For data processing, two approaches are available: ROI-averaged analysis or pixel based analysis. ROI-averaged analysis is based on the computation of vegetation indexes as the average of the entire ROI, whereas pixel based analysis allows to treat separately each pixel of the image. Data used to show phenopix package are from imagery archive of Torgnon Grassland and Larch sites, belonging to the phenocam network. The rationale and the objectives that motivate the processing chain that will be described here are established in the scientific literature since a decade or so

here a paragraph with some key references could be added, Andrew's papers, ICOS protocol, etc..

The steps

The first step is to give a well defined structure to a folder with the function `structureFolder()`.

The second step of the analysis is to choose a region of interest in an image. The functions useful for this step include:

- `DrawROI()` to draw a region of interest in your pictures
- `PrintROI()` to plot your ROI into an image
- `updateROI()` to apply ROI coordinates to an image of different size

Once the roi is chosen, drawn and the underlying coordinates properly saved, colour digital numbers are extracted and vegetation indexes (VIs) are calculated, using one main function `extractVIs()`.

Afterwards, raw VIs must be filtered out to get a reliable seasonal trajectory. This is the job of the function `autoFilter()`.

Then, several options are available to process the resulting data, ranging from fitting a curve to extracting break points on a seasonal trajectory, including several methods to extract relevant moments in the season (aka thresholds). Functions useful for this step include:

- `greenProcess()` to fit a curve to the data (ROI-averaged approach)
- `greenExplore()` to fit all curves and thresholds with no uncertainty estimation, this function is coupled with
- `plotExplore()`, which plots all fittings and thresholds in the object in output from `greenExplore()`
- `spatialGreen()` to fit a curve to the data (pixel-based approach)
- `PhenoBP()` to extract break points on a seasonal trajectory of data

A number of facilities are then built to plot, summarise, post process and render the results. These include:

- generic `plot()`, `print()`, `update()` and `summary()` functions with dedicated methods
- `plotSpatial()` to plot results from the pixel-based analysis
- `extractParameters()` to extract thresholds and curve parameters after the pixel-based analysis.

In the following paragraphs each step will be discussed and illustrated in detail.

Structuring a folder tree useful for the analysis

Giving a good structure to your analysis can make all subsequent steps simple and straightforward. If you are running a site that records images you will be dealing with quite heavy folders (with likely multiple years of data, hence some thousand files of images) that you need to handle with care. We suggest separate folders for each site (of course) but also year of analysis. Each year folder should contain a subfolder with all images to be processed (`/IMG`), one folder containing the reference image, i.e. the image you will use to draw your ROI (`/REF`), one folder containing data for the region of interest (`/ROI`) and one folder containing extracted vegetation indexes (`/VI`).
The function `structureFolder()` provides a facility to create appropriate subfolders:

```
my.path <- structureFolder(path = getwd(), showWarnings = FALSE)

## Put all your images in /home/gian/phenopix_vignette/vignettes/IMG/
## Put your reference image in /home/gian/phenopix_vignette/vignettes/REF/
## Draw your ROI with DrawROI():
##  set path_img_ref to  /home/gian/phenopix_vignette/vignettes/REF/
##  set path_ROIs to /home/gian/phenopix_vignette/vignettes/ROI/
## Then you can extractVIs():
##  set img.path as /home/gian/phenopix_vignette/vignettes/IMG/
##  set roi.path as /home/gian/phenopix_vignette/vignettes/ROI/
##  set vi.path to /home/gian/phenopix_vignette/vignettes/VI/
## ------------------------
## Alternatively, assign this function to an object and use named elements of the returned l

str(my.path)

## List of 4
##  $ img: chr "/home/gian/phenopix_vignette/vignettes/IMG/"
##  $ ref: chr "/home/gian/phenopix_vignette/vignettes/REF/"
##  $ roi: chr "/home/gian/phenopix_vignette/vignettes/ROI/"
##  $ VI : chr "/home/gian/phenopix_vignette/vignettes/VI/"
```

`structureFolder()` creates subfolder at a given path (in this example, the working directory) and stores all path in a named list. You can easily access all needed paths by simply pointing to the right object in your path object. Note that if one folder already exists the function do not overwrite existing folders, but gives a warning. Note that the suggested structure is absolutely not mandatory. It is just a suggestion that can make easier the next steps. Once the folder structure is done, you have to:

- manually put your series of images to be processed into the /IMG folder
- manually put one of such images in the /REF folder, this is the image that will be printed on screen to draw your ROI.

Drawing a region of interest (ROI)

Apart from structuring folders, drawing a roi is the first, hence most important step of the analysis. The procedure is based on two steps: first, a reference image (chosen by the user) is plotted by calling function `readJPEG()` from package `jpeg` and `rasterImage()`. In Fig.1 is the reference image from one of our sites, Torgnon (NW Italy, 2100 m of elevation) and the code used to plot the image.

```
img <- jpeg::readJPEG('REF/20130630T1000.jpg')
ratio <- dim(img)[1]/dim(img)[2]
```

```
par(mar=c(rep(1,4)))
plot(0, type = "n", xlim = c(0, 1),
ylim = c(0, 1), axes = FALSE,
ylab='', xlab='')
rasterImage(img, xleft = 0, ybottom = 0,
        xright = 1,
                ytop = ratio)
```

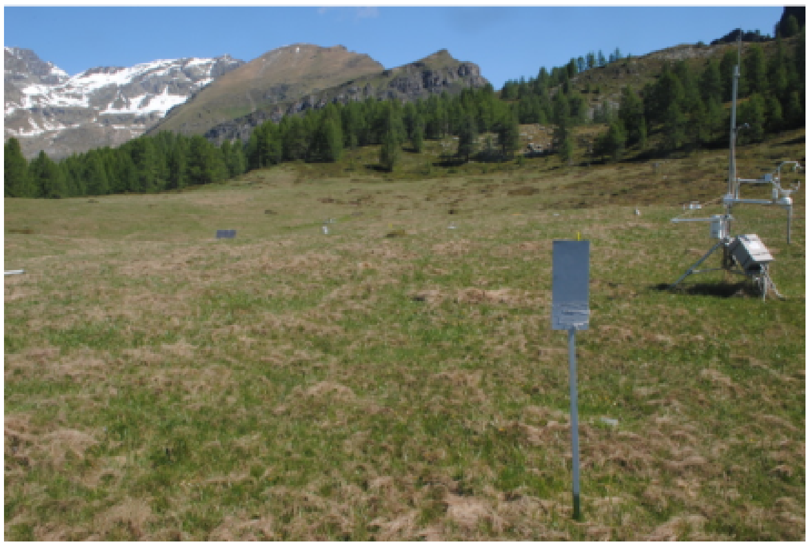

Figure 1: A jpeg image printed on a graphic device using `readJPEG` and `rasterImage`

This chunk of code is automatically included in the `DrawROI()` function. The usage is:

```
DrawROI(path_img_ref, path_ROIs, nroi = 2, roi.names=c('fg',
'bg'))
```

where path_img_ref is the folder of your reference image, path_ROIs is the path in your computer where to store RData with ROI properties, number of ROIs and their names. A call to the function opens a graphic device and allows the use of `locator()` to define your ROI(s). Note that the use of `locator` is somewhat system specific. Check out the help file `?locator` for more details. Locator allows to draw a polygon by left-clicking vertices and then right-clicking (or press ESC on MacOS) to close the polygon. If you have chosen more than one ROI, after closing your first polygon, the image will apper again unmodified to draw the second ROI, and so on. Note that the plot title helps you in remebering which of your ROIs you are actually drawing. When you are done, in your `path_ROIs` an RData called `roi.data.RData` will be stored. This is actually a list with the following structure:

```
load('ROI/roi.data.Rdata')
str(roi.data)
```

```
## List of 2
##  $ fg:List of 2
##   ..$ pixels.in.roi:'data.frame':    273920 obs. of  3 variables:
##   .. ..$ rowpos: num [1:273920] 0.00156 0.00313 0.00469 0.00625 0.00781 ...
##   .. ..$ colpos: num [1:273920] 0.00156 0.00156 0.00156 0.00156 0.00156 ...
##   .. ..$ pip   : int [1:273920] 0 0 0 0 0 0 0 0 0 0 ...
##   ..$ vertices     :List of 2
##   .. ..$ x: num [1:9] 0.0176 0.0193 0.2443 0.5051 0.6551 ...
##   .. ..$ y: num [1:9] 0.2666 0.0288 0.0194 0.0138 0.0232 ...
##  $ bg:List of 2
##   ..$ pixels.in.roi:'data.frame':    273920 obs. of  3 variables:
##   .. ..$ rowpos: num [1:273920] 0.00156 0.00313 0.00469 0.00625 0.00781 ...
##   .. ..$ colpos: num [1:273920] 0.00156 0.00156 0.00156 0.00156 0.00156 ...
##   .. ..$ pip   : int [1:273920] 0 0 0 0 0 0 0 0 0 0 ...
##   ..$ vertices     :List of 2
##   .. ..$ x: num [1:8] 0.0278 0.0244 0.3346 0.5699 0.633 ...
##   .. ..$ y: num [1:8] 0.416 0.364 0.332 0.327 0.388 ...
```

A two elements list (one for each ROI) with ROI names. Each element is again a list containing two elements. One is a data.frame containing coordinates of all image pixels, together with a code indicating whether the given pixel belongs to the ROI or not. The second is a list with the coordinates of ROI margins as in output from `locator()`.

Additionally, in `path_ROIs` separate jpeg files for each of your regions of interest are stored. A call to the function `printROI()` allows to plot in the same graph all existing ROIs for a picture. In the example from Torgnon, two ROIs were drawn, one corresponding to the foreground of the image and one to the background (`fg` and `bg` respectively. Here is the code to generate the plot in fig.2:

```
PrintROI(path_img_ref = 'REF/20130630T1000.jpg',
              path_ROIs = 'ROI/',
              which = 'all',
              col = palette()
)
```

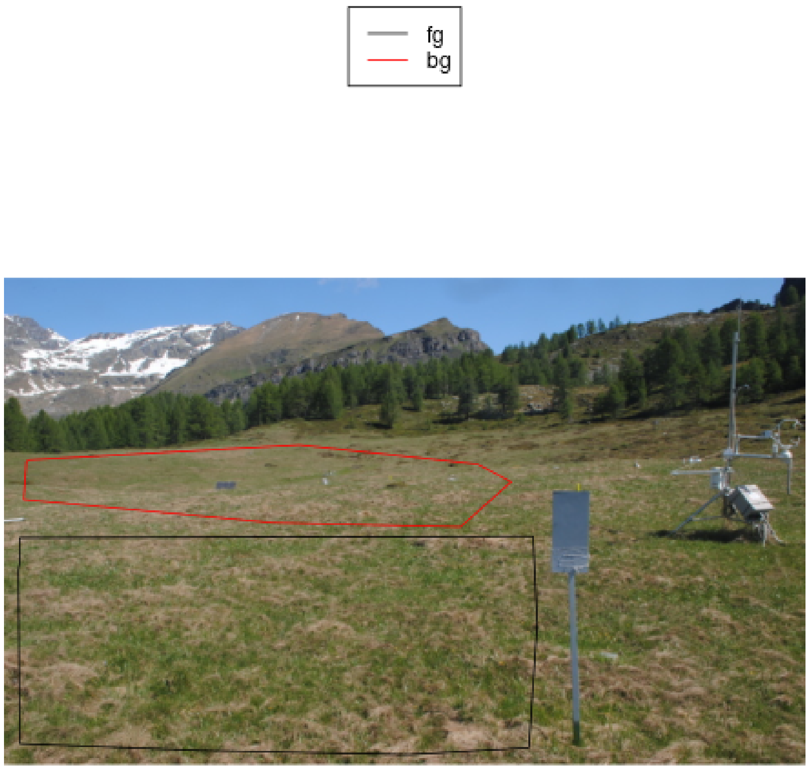

Figure 2: A plot of your regions of interest (ROIs), in output from PrintROI()

When you draw a ROI on your best quality image (say 640 x 428 pixels, as the REF image for Torgnon) you will probably need to identify the same ROI in smaller size images. This will be the case, for example, if you want to conduct a pixel-based analysis, illustrated later on. Pixel based analysis is computationally intense and therefore it is suggested to run it on rather small size images. The function `updateROI` allows to recalculate pixels falling within a given ROI in images of different size compared to the one where the ROI was first drawn. Usage is:

```
updateROI(old.roi.path, new.path.img.ref, new.roi.path)
```

`old.roi.path` is the path to the folder containing your old `roi.data.Rdata`, `new.path.img.ref` is the path to the folder containing your re-sized image, and `new.roi.path` is the folder where your new `roi.data.Rdata` will be stored.

Extraction of vegetation indexes

At this point, you have an r object stored as `roi.data.Rdata` in your ROI path that defines which pixels fall into one or more ROIs. The next step will be to extract information on those pixels from each of your images. The function that performs this task is `extractVIs` and the usage is as follows:

```
extractVIs(img.path, roi.path, vi.path = NULL, roi.name = NULL,
plot = TRUE, begin = NULL, spatial = FALSE)
```

`img.path` is the path where one year of images are stored. It is not mandatory to have only one year of images in your folder. However it is suggested to structure your data into separate folders for each year because nearly all the functions we will see later are designed to work an a single season of data. `roi.path` is the path to your `roi.data.Rdata`, `VI.path` is the path where extracted vegetation indexes will be saved. Hence, this function can be assigned to an object to have your vegetation indexes returned into R, or alternatively `load`ed later if not assigned. The argument `begin` allows to set a beginning date to update an existing time series without reprocessing the whole year of data. For example, if you run extractVIs in mid june to have a first look at your time series, once your season will be completed you do not want to re-run the analysis on the already processed images. Hence, you set the argument `begin` to the first unprocessed date. A new `roi.data.Rdata` will be saved in your path, with the beginning date incorporated in the filename. It is up to the user to bind the old and new time series.

The argument `spatial` allows to perform pixel-based analysis. This is a topic that will be discussed later.

The construction of the time series impies that R recognizes a time vector, tipically retrieved from the file name of each picture. The function responsible for this conversion is `ExtractDateFilename()`. It is a rather internal function but it is worth to look how it works to properly set the filenames of your imagery archive. Arguments to the function are `filename` and `date.code`. Filename must be a character string with an underscore “`_"` that separates site name and date (e.g. ‘torgnon_20140728.jpg’). One and only one underscore must appear before the date stamp. The format of your date must be provided in `date.code`. In the example above, `date.code` will be: ‘yyyymmdd’. Let’s look at some examples, but before doing so, it is worth to remember the the file naming system is under your responsibility when you set up the storage process for your images, or by some renaming routines set up later.

```r
filename <- 'torgnon_20140728.jpg' ## correct, with no hour
## if hour is missing it is defaulted to 12 pm
extractDateFilename(filename, date.code='yyyymmdd')
```

```
## [1] "2014-07-28 12:00:00 CEST"
```

```r
filename <- 'torgnon_201407281100.jpg' ## correct, with hour
## hours and minutes to upper letters, in R POSIX style
extractDateFilename(filename, date.code='yyyymmddHHMM')
```

```
## [1] "2014-07-28 11:00:00 CEST"
```

```r
filename <- 'torgnon_ND_201407281100.jpg' ## wrong, with two
## underscores before date, the function returns NA
extractDateFilename(filename, date.code='yyyymmddHHMM')
```

```
## [1] NA
```

```r
filename <- 'torgnon_1407281100.jpg' ## correct, with 2 numbers for the year
extractDateFilename(filename, date.code='yymmddHHMM')
```

```
## [1] "2014-07-28 11:00:00 CEST"
```

```r
## any separator for date elements is allowed
## including underscore
filename <- 'torgnon_2014.07_28-11.00.jpg'
extractDateFilename(filename, date.code='yyyy.mm_dd-HH.MM')
```

```
## [1] "2014-07-28 11:00:00 CEST"
```

Now let's look from closer at the structure of the object `VI.data` saved in your `/VI` directory.

```r
load('VI/VI.data.Rdata')
summary(VI.data) ## a list with two data.frames, one for each ROI
```

```
##    Length Class      Mode
## fg 18     data.frame list
## bg 18     data.frame list
```

```r
names(VI.data[[1]]) ## check which vegetation indexes are extracted
```

```
##  [1] "date"   "doy"    "r.av"   "g.av"   "b.av"   "r.sd"   "g.sd"
##  [8] "b.sd"   "bri.av" "bri.sd" "gi.av"  "gi.sd"  "gei.av" "gei.sd"
## [15] "ri.av"  "ri.sd"  "bi.av"  "bi.sd"
```

The processing of each ROI produces a data.frame object with date in POSIX format, numeric day of year (doy), and the vegetation indexes. Green, red and blue digital numbers (range [0,255]) averaged over the ROI (g.av, r.av and b.av, respectively), their standard deviations (g.sd, r.sd and b.sd). bri.av is the ROI averaged brightness, calculated as the sum of red green and blue digital numbers for each pixel and then averaged. From the digital numbers (dn) of each colour, relative indices (rel.i) are calculated as follows:

$\text{rel.i} = \text{dn}_{\text{color}}/\ (\text{dn}_{\text{red}} + \text{dn}_{\text{green}} + \text{dn}_{\text{blue}})$

These values are calculated for each pixel and then averaged over the entire ROI (columns gi.av, ri.av, bi.av), and the standard deviation is calculated as well. In fig.3 you can see how a seasonal course of raw colour digital numbers of a subalpine grassland site looks like:

```
with(VI.data$fg, plot(date, r.av, pch=20, col='red',
  ylim=c(0,255), ylab='DN [0,255]'))
with(VI.data$fg, points(date, g.av, col='green', pch=20))
with(VI.data$fg, points(date, b.av, col='blue', pch=20))
```

More interesting is the plot of relative indices (fig.4):

```
with(VI.data$fg, plot(date, ri.av, pch=20, col='red',
  ylim=c(0.1,0.6), ylab='Relative indices'))
with(VI.data$fg, points(date, gi.av, col='green', pch=20))
with(VI.data$fg, points(date, bi.av, col='blue', pch=20))
```

Several patterns are interesting in the seasonal course of fig.4:

- Snow disappearance (mid May) lead to an increase in relative red and a sharp decrease in relative blue
- The green signal follows a bell shaped pattern throughout the growing season, with a maximum in late July. This signal is somewhat mirrored by an inverse behavior of relative blue, whereas relative red gradually increases througout the season.

Filter out data

Data retrieved from images often need robust methods for polishing the time series. Bad weather conditions, low illumination, dirty lenses are among the most

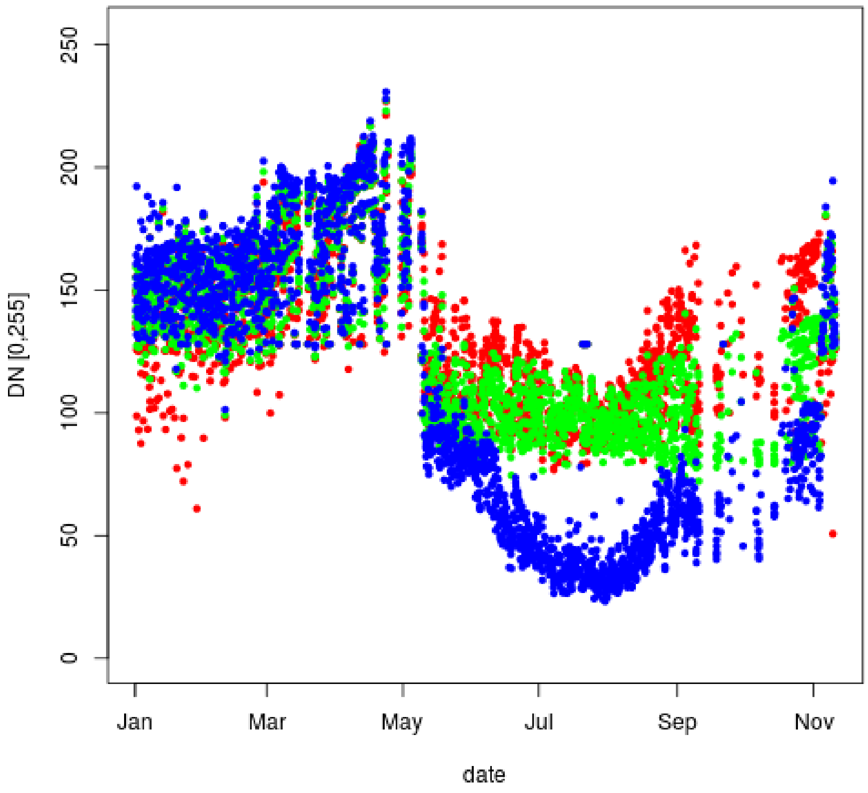

Figure 3: Seasonal course of raw digital numbers, Torgnon, year 2012

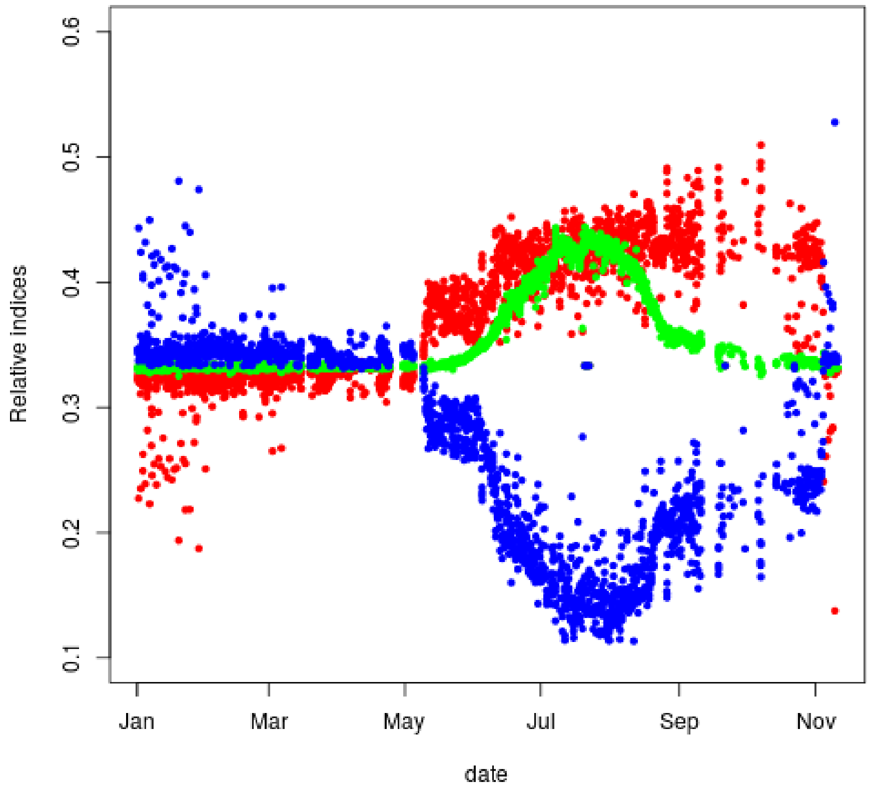

Figure 4: Seasonal course of relative green red and blue indices, Torgnon grassland, year 2012

common issues that determine noise in the time series of vegetation indices. Accordingly we designed a function `autoFilter()` based on 4 different approaches, see the examples in `?autoFilter` for details in the filtering procedure. The function is designed to receive in input a data.frame structured as in output from `extractVIs`, hence its default expression may appear rather complicate:

```
autoFilter(data, dn=c('ri.av', 'gi.av', 'bi.av'), raw.dn = FALSE,
brt = 'bri.av', na.fill = TRUE, filter = c("night", "spline",
"max"), filter.options = NULL, plot = TRUE, ...)
```

But when applied to the `VI.data` object generated before it is quite straightforward as you see in the code below. Note also that `autoFilter()` returns by default a diagnostic plot shown in fig.5:

```
filtered.data <- autoFilter(VI.data$fg)
```

```
str(filtered.data)
```

```
## 'zoo' series from 1 to 314
##   Data: num [1:277, 1:7] 0.323 0.324 0.318 0.324 0.316 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:7] "rcc" "gcc" "bcc" "brt" ...
##   Index:  num [1:277] 1 2 3 4 5 6 7 8 9 10 ...
```

```
names(filtered.data)
```

```
## [1] "rcc"             "gcc"             "bcc"             "brt"
## [5] "night.filtered"  "spline.filtered" "max.filtered"
```

In the structure of the output data.frame there are two important points:

- We introduce here a new class of R objects (`zoo`). From here on all further analyses are based on `zoo` (or, to a lesser extent `ts`) time series. The time index of the data is numeric day of year (doy). As a consequence, the attribute year is lost at this step of the analysis (i.e. we suggest to include it in the object name);

- The function `autoFilter` aggregates the data at a daily time step by default. The returned data.frame contains unfiltered (but still daily aggregated) colour indices (here called gcc, rcc and bcc, cc standing for chromatic coordinate) and a column of data for each filtering step. The name of the filter applied is reported in the column name.

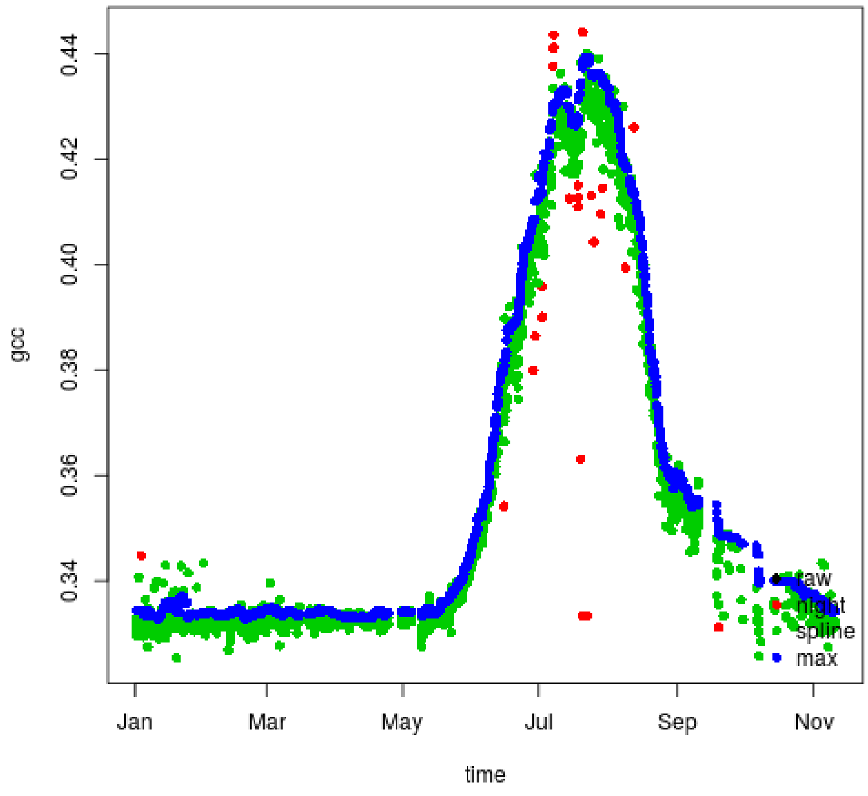

Figure 5: Raw and filtered relative greenness index, default plot of function `autoFilter()`

For those unfamiliar with the `zoo` structure we have created a function 'convert' to convert from zoo to a normal data.frame

```
dataframed <- convert(filtered.data, year='2012')
str(dataframed)
```

```
## 'data.frame':    277 obs. of  9 variables:
##  $ rcc            : num  0.323 0.324 0.318 0.324 0.316 ...
##  $ gcc            : num  0.331 0.332 0.334 0.331 0.332 ...
##  $ bcc            : num  0.346 0.344 0.35 0.343 0.354 ...
##  $ brt            : num  442 409 447 409 435 ...
##  $ night.filtered : num  0.331 0.332 0.334 0.331 0.332 ...
##  $ spline.filtered: num  0.331 0.332 0.333 0.331 0.332 ...
##  $ max.filtered   : num  0.334 0.334 0.334 0.334 0.334 ...
##  $ doy            : num  1 2 3 4 5 6 7 8 9 10 ...
##  $ time           : POSIXct, format: "2012-01-01" "2012-01-02" ...
```

However, we strongly recommend to get familiar with the `zoo` package since it has wonderful facilities for plotting, aggregating and filling time series.

Fit a curve to the data

The seasonal trajectory of greenness index of a vegetation canopy provides per se important information, but to turn qualitative information into quantitative data we need to make some more computation. Traditionally, data similar to these (e.g. satellite-based NDVI trajectories) are processed in two main ways:

- extract time thresholds based on a percentage of development (e.g. the day when half of the maximum value of the index is reached);
- fit a curve and extract relevant thresholds based on curve properties.

In the package `phenopix` both possibilities are available. The core function for data fitting and threshold extraction is `greenProcess()`. This function calls and is related to several rather internal functions that perform the different fittings. Available fittings include:

- the fit of a cubic spline
- the fit of an equation proposed by Beck et al. (2006)
- the fit of an equation proposed by Elmore et al. (2012)
- the fit of an equation proposed by Klosterman et al. (2014) with two implementations

- the fit of an equation proposed by Gu et al. (2009)

All fits are based on a double - logistic function with a different number of parameters.

After curve fitting, relevant dates in the seasonal trajectory (aka thresholds) are extracted with different methods:

- A method called `spline` (the name will soon change) which splits the seasonal course into increasing and decreasing trajectory based on the sign of the first derivative and then identifies the 50% of both the increasing and decreasing trajectory. It allows to determine start of season (sos), end of season (eos) and length of season (los) as the difference between the two.
- A method called `derivatives` which extends `spline` in that it also calculates maximum growing and decreasing rates
- A method based on Klosterman approach which individuates 4 moments in the seasonal trajectory. Greenup represents the beginning of growth, maturity represents the reaching of some summer plateau, senescence represents the beginning of green decrease (or yellowing increase) and dormancy represents the end of the growing season.
- A method based on Gu approach which individuates 4 moments and some other curve parameters. The 4 relevant moments do not differ in their meaning compared to Klosterman phases, and are called upturn date (UD), stabilization date (SD), downturn date (DD) and recession date (RD).

Detail on curve fitting and threshold extraction is provided in the help function of ?greenProcess as well as in the help files of other more internal functions such as `?KlostermanFit`, `?GuFit`, `?PhenoExtract`. In fig.6 weshow 4 different fitting methods applied to the same data (Torgnon grassland).

```
## spline curve + spline thresholds
## fit1 <- greenProcess(filtered.data$max.filtered,
##   'spline',
##   'spline',
##   plot=FALSE
## )
summary(fit1)
```

```
##
## Data
##      Index        object$data
##  Min.   :  1.0   Min.   :0.3328
##  1st Qu.: 71.0   1st Qu.:0.3340
##  Median :152.0   Median :0.3374
##  Mean   :150.6   Mean   :0.3582
##  3rd Qu.:223.0   3rd Qu.:0.3755
##  Max.   :314.0   Max.   :0.4394
##
## Predicted
##      Index       object$fit$fit$predicted
##  Min.   :  1.0   Min.   :0.3330
##  1st Qu.: 71.0   1st Qu.:0.3340
##  Median :152.0   Median :0.3370
##  Mean   :150.6   Mean   :0.3582
##  3rd Qu.:223.0   3rd Qu.:0.3766
##  Max.   :314.0   Max.   :0.4358
##
## Formula
## NULL
##
## Thresholds
##         sos         eos         los         pop         mgs         rsp
## 168.0000000 232.0000000  64.0000000 203.0000000   0.4177261          NA
##         rau        peak         msp         mau
##          NA   0.4358051   0.3843107   0.3875324

## plot(fit1, type='p', pch=20, col='grey')

## Beck fitting + derivatives
## fit2 <- greenProcess(filtered.data$max.filtered,
## 'beck',
## 'derivatives',
## plot=FALSE)

summary(fit2)

##
## Data
##      Index        object$data
##  Min.   :  1.0   Min.   :0.3328
##  1st Qu.: 71.0   1st Qu.:0.3340
##  Median :152.0   Median :0.3374
##  Mean   :150.6   Mean   :0.3582
##  3rd Qu.:223.0   3rd Qu.:0.3755
##  Max.   :314.0   Max.   :0.4394
##
## Predicted
##      Index       object$fit$fit$predicted
##  Min.   :  1.0   Min.   :0.3358
##  1st Qu.: 71.0   1st Qu.:0.3359
##  Median :152.0   Median :0.3371
##  Mean   :150.6   Mean   :0.3600
##  3rd Qu.:223.0   3rd Qu.:0.3781
##  Max.   :314.0   Max.   :0.4387
##
## Formula
## expression(mn + (mx - mn) * (1/(1 + exp(-rsp * (t - sos))) +
##     1/(1 + exp(rau * (t - eos)))))
##
## Thresholds
##          sos          eos          los          pop          mgs
## 186.000000000 261.000000000  75.000000000 200.000000000   0.406786541
##          rsp          rau         peak          msp          mau
##   0.003639004  -0.007351217   0.438650638   0.424337784   0.346366590
```

```
## plot(fit2, type='p', pch=20, col='grey')

## klosterman fitting + klosterman thresholds
## fit3 <- greenProcess(filtered.data$max.filtered,
## 'klosterman',
## 'klosterman',
## plot=FALSE)
summary(fit3)
```

```
##
## Data
##      Index        object$data
##  Min.   :  1.0   Min.   :0.3328
##  1st Qu.: 71.0   1st Qu.:0.3340
##  Median :152.0   Median :0.3374
##  Mean   :150.6   Mean   :0.3582
##  3rd Qu.:223.0   3rd Qu.:0.3755
##  Max.   :314.0   Max.   :0.4394
##
## Predicted
##      Index       object$fit$fit$predicted
##  Min.   :  1.0   Min.   :0.3329
##  1st Qu.: 71.0   1st Qu.:0.3342
##  Median :152.0   Median :0.3388
##  Mean   :150.6   Mean   :0.3582
##  3rd Qu.:223.0   3rd Qu.:0.3758
##  Max.   :314.0   Max.   :0.4375
##
```

```
## Formula
## expression((a1 * t + b1) + (a2 * t^2 + b2 * t + c) * (1/(1 +
##     q1 * exp(-B1 * (t - m1)))^v1 - 1/(1 + q2 * exp(-B2 * (t -
##     m2)))^v2))
##
## Thresholds
##    Greenup   Maturity Senescence   Dormancy
##        142        224        225        253

## plot(fit3, type='p', pch=20, col='grey')

## gu fitting and thresholds
## fit4 <- greenProcess(filtered.data$max.filtered,
## 'gu',
## 'gu',
## plot=FALSE)
summary(fit4)

##
## Data
##      Index        object$data
##  Min.   :  1.0   Min.   :0.3328
##  1st Qu.: 71.0   1st Qu.:0.3340
##  Median :152.0   Median :0.3374
##  Mean   :150.6   Mean   :0.3582
##  3rd Qu.:223.0   3rd Qu.:0.3755
##  Max.   :314.0   Max.   :0.4394
##
## Predicted
##      Index       object$fit$fit$predicted
##  Min.   :  1.0   Min.   :0.3340
##  1st Qu.: 71.0   1st Qu.:0.3340
##  Median :152.0   Median :0.3397
##  Mean   :150.6   Mean   :0.3582
##  3rd Qu.:223.0   3rd Qu.:0.3750
##  Max.   :314.0   Max.   :0.4368
##
## Formula
## expression(y0 + (a1/(1 + exp(-(t - t01)/b1))^c1) - (a2/(1 + exp(-(t -
##     t02)/b2))^c2))
##
## Thresholds
##            UD            SD            DD            RD       maxline
## 148.661704355 187.342880427 214.467153725 248.926022679   0.436761779
##      baseline           prr           psr plateau.slope
##   0.333978802   0.002657183  -0.003004145   0.000327951
## plot(fit4, type='p', pch=20, col='grey')

## show all together
par(lwd=3)
plot(filtered.data$max.filtered, type='p', pch=20,
  ylab='Green chromatic coordinate', xlab='DOYs')
lines(fitted(fit1), col='blue')
lines(fitted(fit2), col='red')
lines(fitted(fit3), col='green')
lines(fitted(fit4), col='violet')
legend('topleft', col=c('blue', 'red', 'green', 'violet'),
  lty=1, legend=c('Spline', 'Beck', 'Klosterman', 'Gu'),
  bty='n')
```

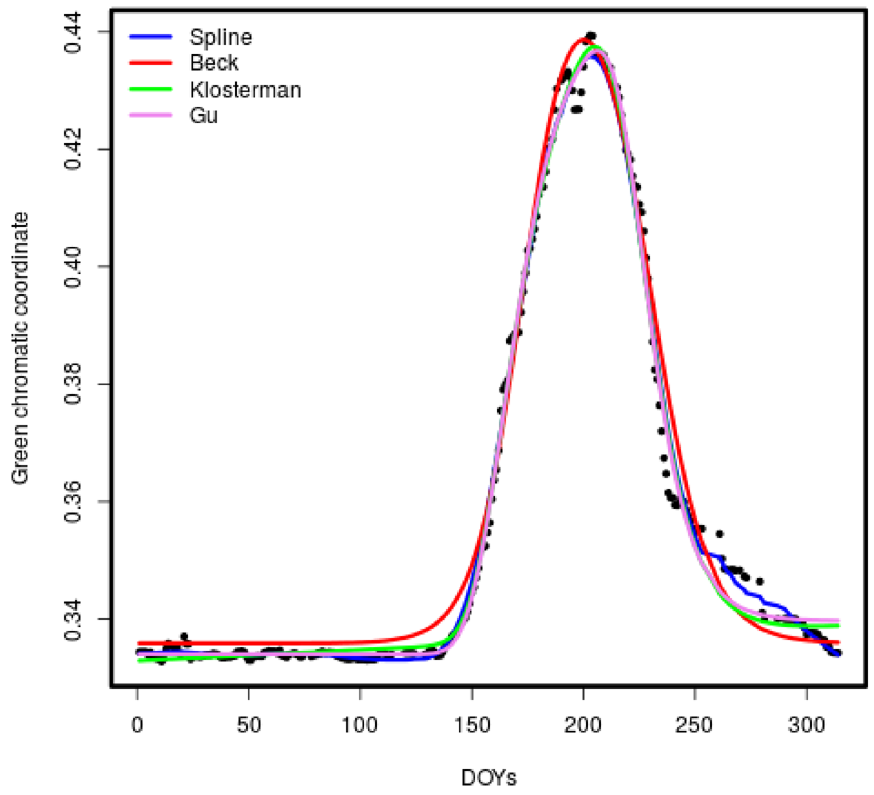

Figure 6: Comparison of 4 different fittings from phenopix package

The function greenProcess creates an object of class phenopix with dedicated methods. The summary function displays a summary of the input data and of the predicted points. It then reports the formula of the fitting equation, if pertinent, see e.g. summary of fit1 which is not based on an equation. Thresholds are printed as well. Note also the fitted function applied to phenopix object that returns a zoo time series of fitted values that can be directly lined to the plot.

To complete the overview on display generic methods applied to the objects of class phenopix here is the application of generic plot (fig.7) and print functions:

```
plot(fit4, pch=20, col='grey', type='p',
  xlab='DOYs', ylab='Green chromatic coordinates')
print(fit4)
```

```
##
##  #### phenopix time series processing ####
##
## FITTING: GU
##
## PREDICTED VALUES:
##      Index       x$fit$fit$predicted
##  Min.   :  1.0   Min.   :0.3340
##  1st Qu.: 71.0   1st Qu.:0.3340
##  Median :152.0   Median :0.3397
##  Mean   :150.6   Mean   :0.3582
##  3rd Qu.:223.0   3rd Qu.:0.3750
##  Max.   :314.0   Max.   :0.4368
##
## FITTING EQUATION:
## expression(y0 + (a1/(1 + exp(-(t - t01)/b1))^c1) - (a2/(1 + exp(-(t -
##     t02)/b2))^c2))
##
## FITTING PARAMETERS:
##          y0          a1          a2         t01         t02          b1
##   0.3339788   0.1104471   0.1047788 129.3082257 200.3364257  14.6352282
##          b2          c1          c2
##  11.7331508  11.1601187   9.0723878
##
## THRESHOLDS: GU
##            UD            SD            DD            RD       maxline
## 148.661704355 187.342880427 214.467153725 248.926022679   0.436761779
##      baseline           prr           psr plateau.slope
##   0.333978802   0.002657183  -0.003004145   0.000327951
##
## UNCERTAINTY: FALSE
##  N of replications = 0
```

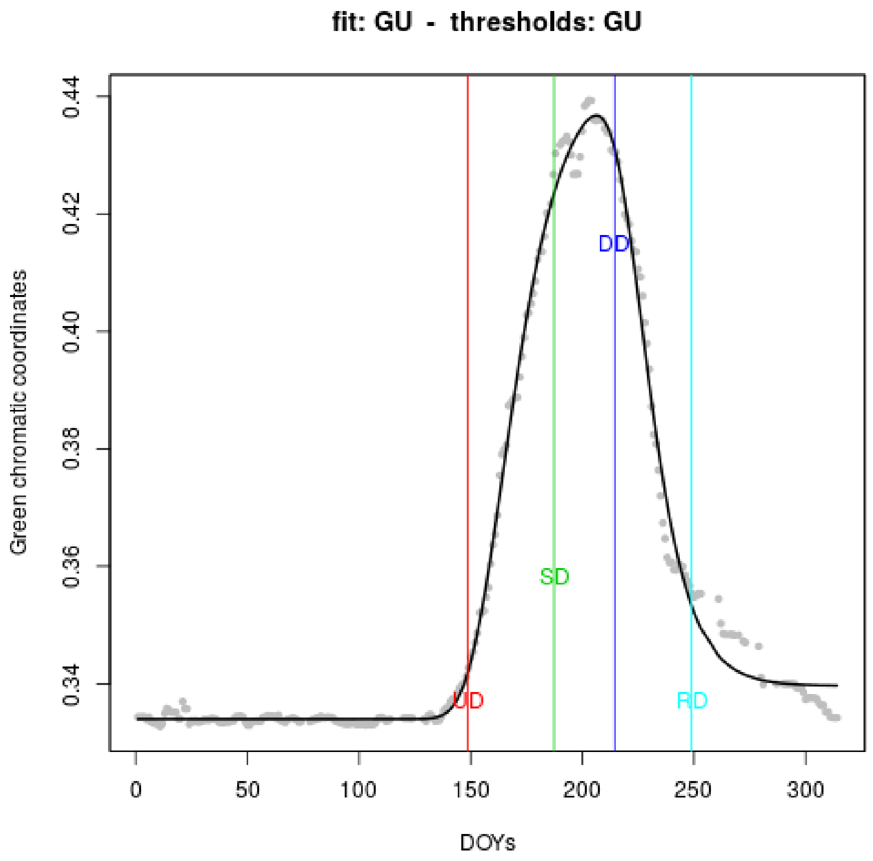

Figure 7: Generic `plot` function applied to `phenopix` objects

The `print` function returns information similar to `summary` but it also reports which fitting and threshold methods were used, and if the uncertainty was estimated. The `plot` function returns a plot similar to the one constructed above, except that extracted thresholds are also shown the as vertical coloured lines. Figure 5 shows that different fitting equation lead to very similar fitted values on the example from Torgnon data. For the sake of robustness, in such situation it is preferable to choose a fitted equation rather than a spline fit. Let's decide to choose the fitting from Gu. Now let's look from closer how do the different threshold extraction methods impact when applied to the same fitted curve in fig.8 (and note the use of `update` generic function with method `phenopix`):

```
fit4.spline <- update(fit4, 'spline', plot=FALSE)
fit4.klosterman <- update(fit4, 'klosterman', plot=FALSE)
fit4.gu <- update(fit4, 'gu', plot=FALSE)
par(mfrow=c(2,2), oma=rep(5,4,4,2), mar=rep(0,4))
plot(fit4.spline, type='n', main='', xaxt='n')
mtext('spline', 3, adj=0.1, line=-2)
plot(fit4.klosterman, type='n', main='', xaxt='n', yaxt='n')
mtext('klosterman', 3, adj=0.1, line=-2)
plot(0, type='n', axes=FALSE, xlab='', ylab='')
plot(fit4.gu, type='n', main='', yaxt='n')
axis(4)
mtext('gu', 3, adj=0.1, line=-2)
```

The `spline` thresholds (50% of increasing and decreasing trajectory) hold a different meaning compared to Klosterman and Gu thresholds. The latter two show good correspondence except that the Klosterman s beginning of senescence occurs later compared to correspondent phase in Gu thresholds (i.e DD, downturn date).

In this paragraph we have shown 4 different approaches to matematically describe the seasonal trajectory of greenness, with additionally 5 methods to extract thresholds on the obtained curves. The combination of curves and threshold methods leads to as many as 20 possible approaches to describe a seasonal trajectory. Sometimes it could be useful to make a decision on which curves and thresholds to use, without computing the uncertainty on all of them. To do so we have designed two functions that provide a quick overview on what would be the best fit and threshod method for your actual trajectory. Here is how to compute the 20 combinations of fit and uncertainty in a single function:

```
explored <- greenExplore(filtered.data$max.filtered)
```

explored is a list with 20 + 1 elements, i.e. the 20 combinations + a vector containing the RMSEs from each of the 4 fittings. This object will only be used as argument of the `plotExplore()` function (fig.9):

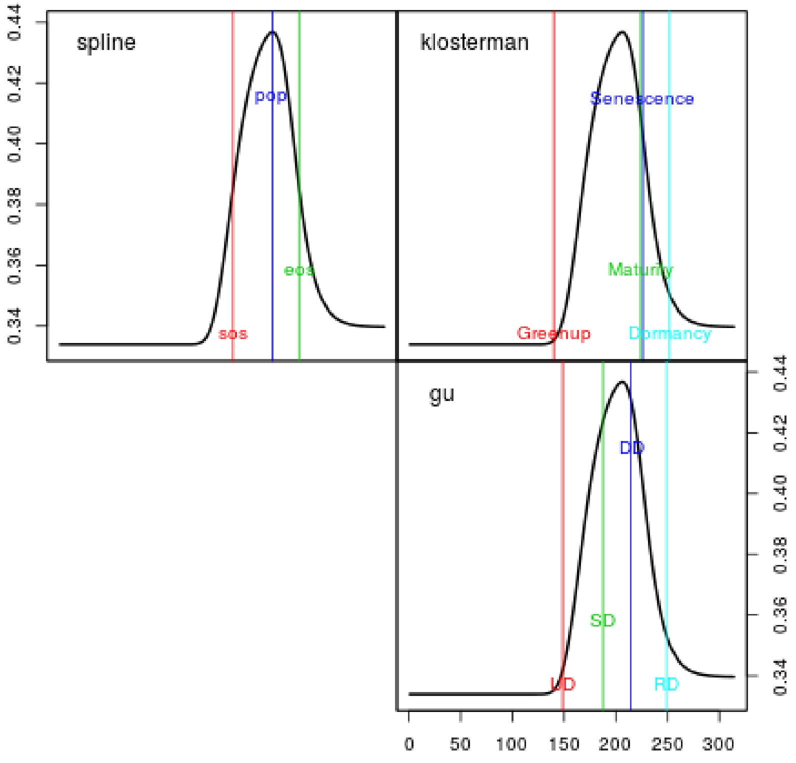

Figure 8: Three threshold methods applied to the Gu fitting

```r
plotExplore(explored)
```

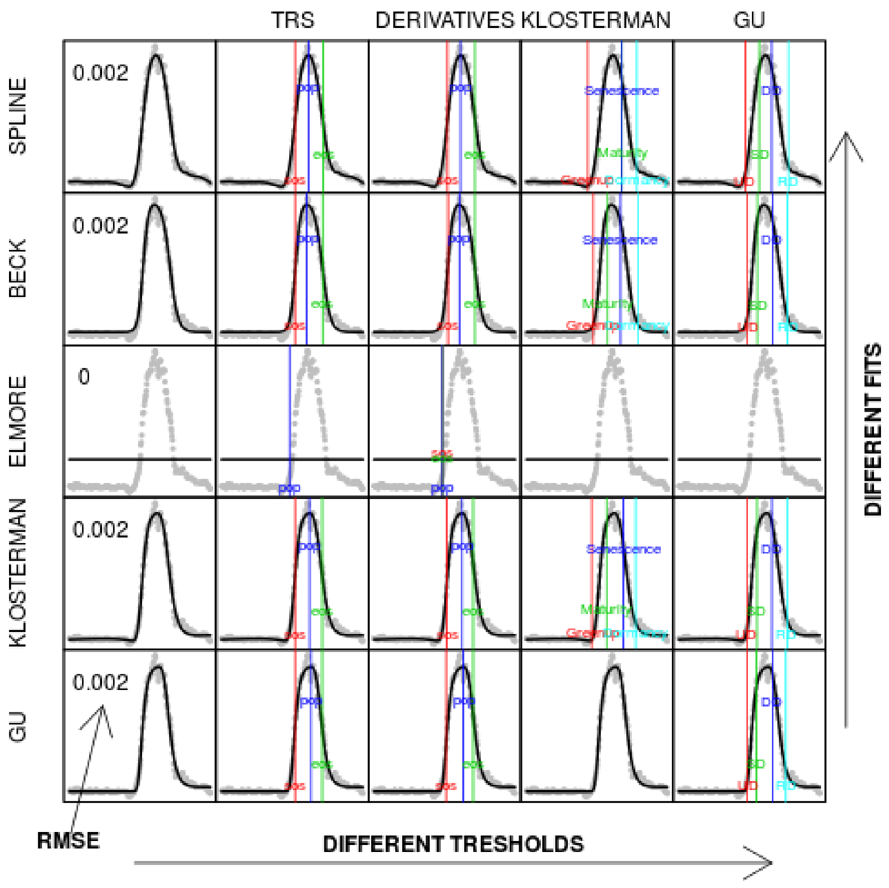

Figure 9: Overview of all combinations of curves and fits as obtained by the `plotExplore` function

The plot in fig.9 shows the impact of different fittings (moving up-downwars) and different thresholds (from left to right) on the same data (Torgnon grassland). The RMSE for each of the four fitting methods is also annotated in the first column. This plot might be useful to choose the most appropriate fitting and thresholding methods on your data.

The uncertainty estimation

One main functionality of the package is the uncertainty estimation. This is performed in different ways depending on the fitting equation. The basic idea behind the uncertainty estimation is how good the smoothing curve fits to the data. The residuals between fitted and observed is therefore used to generate random noise to the data and fitting is applied recursively to randomly - noised original data. This procedure results in an ensemble of curves, curve parameters and extracted thresholds that represent the uncertainty estimate. The uncertainty on curve parameters is automatically propagated to threshold extraction. In the following example the uncertainty estimation is performed on Torgnon grassland data fitted with the approach of Klosterman et al (2014), with 100 replications. Here is the code:

```
fit.complete <- greenProcess(filtered.data$max.filtered, 'gu',
'gu', plot=FALSE, uncert=TRUE, nrep=50)
```

And here is `fit.complete` printed:

```
print(fit.complete)
```

```
##
##  #### phenopix time series processing ####
##
## FITTING: KLOSTERMAN
##
## PREDICTED VALUES:
##      Index       x$fit$fit$predicted
##  Min.   :  1.0   Min.   :0.3329
##  1st Qu.: 71.0   1st Qu.:0.3342
##  Median :152.0   Median :0.3388
##  Mean   :150.6   Mean   :0.3582
##  3rd Qu.:223.0   3rd Qu.:0.3758
##  Max.   :314.0   Max.   :0.4375
##
## FITTING EQUATION:
## expression((a1 * t + b1) + (a2 * t^2 + b2 * t + c) * (1/(1 +
##     q1 * exp(-B1 * (t - m1)))^v1 - 1/(1 + q2 * exp(-B2 * (t -
##     m2)))^v2))
##
## FITTING PARAMETERS:
##            a1            a2            b1            b2             c
##  1.867151e-05  5.090923e-06  3.328690e-01 -1.245649e-03  1.519304e-01
##            B1            B2            m1            m2            q1
##  8.736767e-02  8.705025e-02  1.299899e+02  2.056079e+02  3.967244e+00
##            q2            v1            v2
##  1.995802e+00  4.198100e+00  2.418243e+00
##
## THRESHOLDS: GU  ENVELOPE:QUANTILES
##           UD       SD       DD       RD   maxline  baseline         prr
## 10% 146.9326 188.5265 213.0008 251.0400 0.4374889 0.3328877 0.002446034
## 50% 147.2705 189.1760 213.1443 251.4086 0.4374889 0.3328877 0.002498147
## 90% 147.6557 189.7483 213.3274 251.6832 0.4374889 0.3328877 0.002559312
##               psr plateau.slope
## 10% -0.002779570  0.0003013841
## 50% -0.002737603  0.0003013841
## 90% -0.002707150  0.0003013841
##
## UNCERTAINTY: TRUE
##  N of replications = 100
```

As you can see from the output, the default behaviour of `greenProcess` for the computation of uncertainty is to provide the median, 10th and 90th percentile of the uncertainty ensemble. This may be changed by modifying the `envelope` argument. The other possible option is `min-max` to get minimum mean and maximum. In addition, the quantiles to be chosen with `envelope` = quantiles can be changed by modifying the `quantile` argument. Here is the example:

```
print(update(fit.complete, 'gu', envelope='min-max', plot = FALSE))
```

```
##
##  #### phenopix time series processing ####
##
## FITTING: KLOSTERMAN
##
## PREDICTED VALUES:
##      Index       x$fit$fit$predicted
##  Min.   :  1.0   Min.   :0.3329
##  1st Qu.: 71.0   1st Qu.:0.3342
##  Median :152.0   Median :0.3388
##  Mean   :150.6   Mean   :0.3582
##  3rd Qu.:223.0   3rd Qu.:0.3758
##  Max.   :314.0   Max.   :0.4375
##
## FITTING EQUATION:
## expression((a1 * t + b1) + (a2 * t^2 + b2 * t + c) * (1/(1 +
##     q1 * exp(-B1 * (t - m1)))^v1 - 1/(1 + q2 * exp(-B2 * (t -
##     m2)))^v2))
##
## FITTING PARAMETERS:
##            a1            a2            b1            b2             c
##  1.867151e-05  5.090923e-06  3.328690e-01 -1.245649e-03  1.519304e-01
##            B1            B2            m1            m2            q1
##  8.736767e-02  8.705025e-02  1.299899e+02  2.056079e+02  3.967244e+00
##            q2            v1            v2
##  1.995802e+00  4.198100e+00  2.418243e+00
##
## THRESHOLDS: GU  ENVELOPE:MIN-MAX
##            UD       SD       DD       RD   maxline  baseline         prr
## min  146.1843 186.1598 212.4913 250.4383 0.4374889 0.3328877 0.002368027
## mean 147.2843 189.1266 213.1333 251.3941 0.4374889 0.3328877 0.002501166
## max  149.1719 190.3566 213.5794 252.0516 0.4374889 0.3328877 0.002827987
##               psr plateau.slope
## min  -0.002850601  0.0002539504
## mean -0.002739581  0.0003043358
## max  -0.002667329  0.0004128567
##
## UNCERTAINTY: TRUE
##  N of replications = 100
```

Beside the few options available by default and described above, the uncertainty data.frame is accessible via the `extract` command, by running:

```
extract(fit.complete, 'metrics.uncert') ## get threshold uncertainty
data
```

```
extract(fit.complete, 'params.uncert') ## get parameters of each
fitting curve
```

For example, if you want to use thresholds extracted from the true model and construct uncertainty envelope on them, you can access the uncertainty data.frame by the commands given above.
Note than when the uncertainty is computed, also the `plot` function changes its behaviour, in that it also shows the uncertainty curve ensemble and an error bar on extracted thresholds.

```
plot(fit.complete, type='p', pch=20)
```

The distribution of uncertainty parameters (thresholds + curve parameters) can also be evaluated by means of box-plots with an extra option to the default `plot` method (not shown):

```
plot(fit.complete, what='thresholds')
```

By using the `update` function you can also extract thresholds according to a different method, without refitting the data. Here is the code:

```
update(fit.complete, 'klosterman', plot=FALSE)
```

```
##
##  #### phenopix time series processing ####
##
```

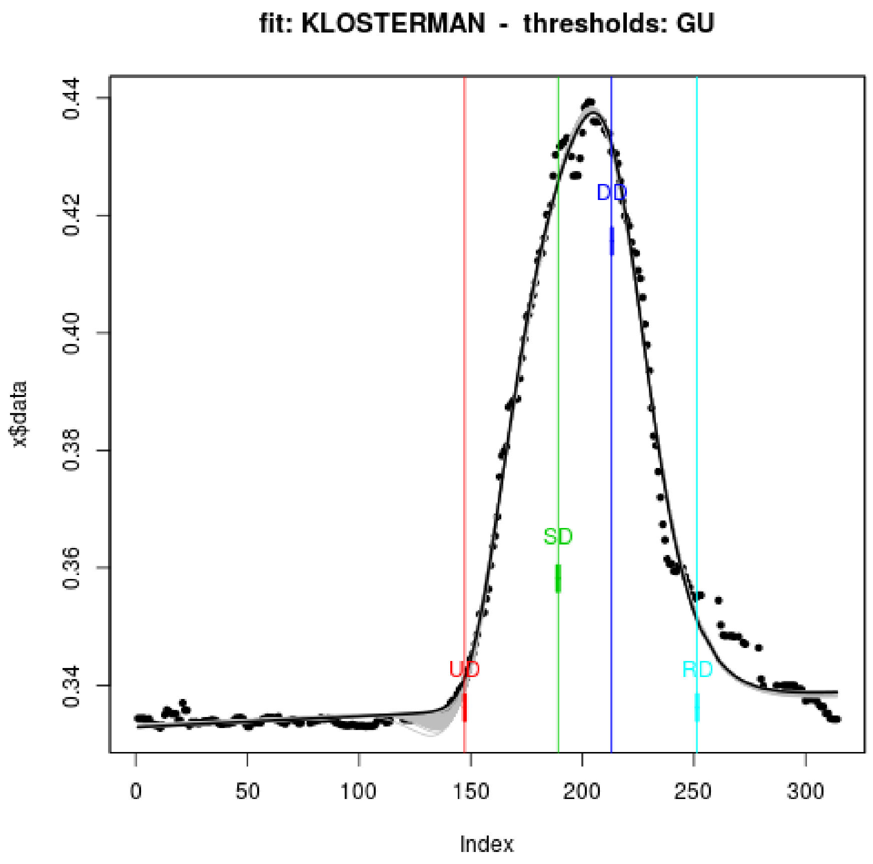

Figure 10: The Uncertainty Estimation (100 rep) on Klosterman fit and Gu thresholds

```
## FITTING: KLOSTERMAN
##
## PREDICTED VALUES:
##      Index        x$fit$fit$predicted
##  Min.   :  1.0   Min.   :0.3329
##  1st Qu.: 71.0   1st Qu.:0.3342
##  Median :152.0   Median :0.3388
##  Mean   :150.6   Mean   :0.3582
##  3rd Qu.:223.0   3rd Qu.:0.3758
##  Max.   :314.0   Max.   :0.4375
##
## FITTING EQUATION:
## expression((a1 * t + b1) + (a2 * t^2 + b2 * t + c) * (1/(1 +
##     q1 * exp(-B1 * (t - m1)))^v1 - 1/(1 + q2 * exp(-B2 * (t -
##     m2)))^v2))
##
## FITTING PARAMETERS:
##            a1            a2            b1            b2             c
##  1.867151e-05  5.090923e-06  3.328690e-01 -1.245649e-03  1.519304e-01
##            B1            B2            m1            m2            q1
##  8.736767e-02  8.705025e-02  1.299899e+02  2.056079e+02  3.967244e+00
##            q2            v1            v2
##  1.995802e+00  4.198100e+00  2.418243e+00
##
## THRESHOLDS: KLOSTERMAN  ENVELOPE:QUANTILES
##     Greenup Maturity Senescence Dormancy
## 10%     136      223        225      253
## 50%     138      224        225      253
## 90%     140      224        225      253
##
## UNCERTAINTY: TRUE
##  N of replications = 100
```

There is a last method to define thresholds on a time series that does not need a fitting. It implements the use of break points from the package `strucchange` and works as follows:

```
print(PhenoBP(x = filtered.data$max.filtered,
        breaks = 3, plot = FALSE,
        confidence= 0.99))
```

```
##       bp1 bp2 bp3
## 0.5%  141 203 244
## mean  142 204 245
## 99.5% 143 205 246
```

The user can set the maximum number of breakpoints to be identified, the confidence interval at which the calculation must be performed and an option or a plot. The output dataframe contains the day of the year for each of the breakpoints and their respective confidence intervals.

Pushing forward the analysis: pixel - based phenology

In order to toughroughly exploit the capabilities of an imagery archive, spatial analysis represents the most promising feature. Hence, specific functions are built to fit curves and extract thresholds on each pixel included in a region of interest instead of averaging the greenness index over the entire ROI. The computation for this analysis may be quite intense, therefore it is suggested to either conduct it in small ROIs or on images with reduced size. For this approach a specific function `updateROI()` was designed. The default of the function was illustrated before. To illustrate the group of new functions for the spatial analysis we present a new dataset from Torgnon Larch Site. At this site we defined 4 ROIs corresponding to individual or small groups of trees which showed a somewhat different phenology by simply looking at the pictures, as shown in the plot below. ROIs distinguish plants where the autumn phenology occurs slightly earlier, or later and an evergreen spruce tree (fig.11).

```
PrintROI('larch/2012/REF/20121004T1400.jpg', 'larch/2012/ROI/')
```

The function that allows to extract VIs in the pixel-based analysis is the same as in the ROI-averaged approach, except that `spatial` is set to TRUE.

```
extractVIs(images.2012, folders640.2012$roi, folders640.2012$VI,
spatial=TRUE)
```

The structure of the object in output is quite different from the one in the ROI-average approach, and is as follows:

- A list with 4 elements, the 4 ROIs
- Within each element of this list is another list with one element for each time step processed
- Within each element of this list is a data.frame with three columns corresponding to the digital number of red green and blue, respectively, and there is a row for each pixel in the ROI.

```
class(VI.data.spatial)

## [1] "list"

names(VI.data.spatial)
```

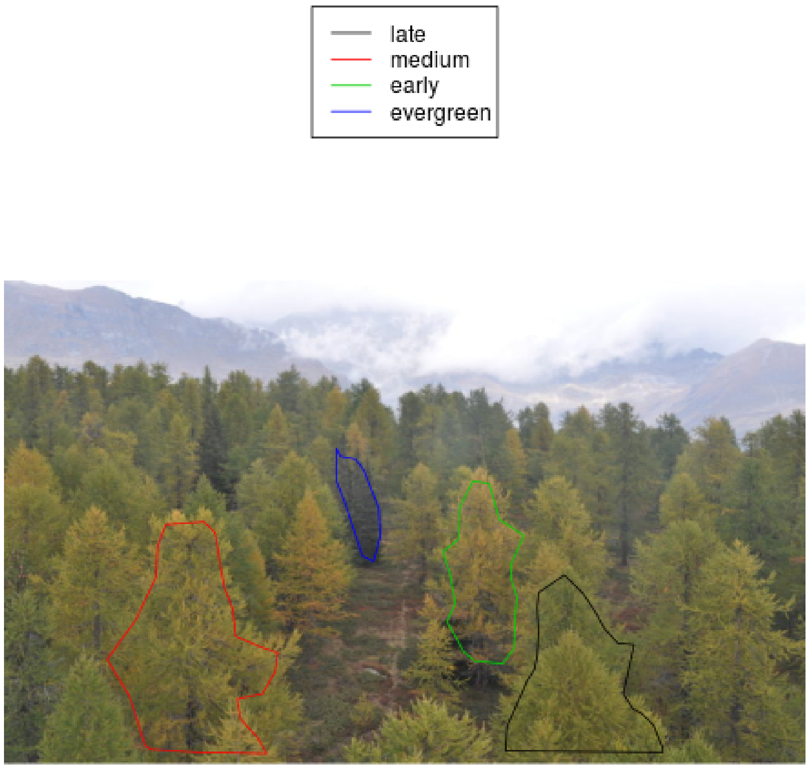

Figure 11: ROIs extracted at the Torgnon Larch site

```
## [1] "late"      "medium"    "early"     "evergreen"
```

```
str(VI.data.spatial$late[[1]])
```

```
## 'data.frame':    11371 obs. of  3 variables:
##  $ red  : num  171 196 207 197 214 213 182 190 157 183 ...
##  $ green: num  178 200 211 200 217 216 185 193 156 187 ...
##  $ blue : num  186 209 220 205 222 223 192 202 172 198 ...
```

Hence, in the ROI named `late` there are 11371 pixels. The function `spatialGreen` which default is shown below, performs the whole computation of relative indices, filtering and fitting for each pixel.

```
spatialGreen(spatial.list, fit, threshold, filters='default',
parallel=TRUE, save=FALSE, path=NULL, assign=TRUE)
```

Arguments of `spatialGreen()` are similar to those of `greenProcess()` and `autoFilter()`, in that a `fit` argument allows to choose a fitting method, a `threshold` argument allows to define a threshold method, the argument `filters` controls number and sequence of filtering steps. `spatialGreen` is an extremely computationally intense function, therefore it required particular care to memory saturation. The argument `parallel` allows to parallelize processes into all but one of your processors. Argument `save` provides the option to save each pixel's fit into a small RData object (saved in `path`). This will prevent your memory from being filled up by the creation of a very large object. If you decide to save your temporary fits than you don't need to assign the product of `spatialGreen`. Instead, if `save` is FALSE assign must be TRUE otherwise the result of your long analysis will be only printed on screen. The main argument `spatial.list` for the function `spatialGreen()` must contain data from one single ROI. Here is how we used `spatialGreen()` to process the ROI named `late` in our Larch site.

```
spatialGreen(spatial.list = VI.data.spatial$late, fit = 'gu',
threshold = 'gu', filters = 'default', parallel = TRUE, save =
TRUE, path = late.path, assign = FALSE)
```

In the code above we decided to save temporary fits from each pixel without constructing a huge object (with save = TRUE and assign = FALSE). Each pixel of the ROI was therefore processed to extract VIs, filter them, fit the curve from Gu et al (2009), extract thresholds with the same approach. After have run the function, the folder `late.path` will contain 11371 `.Rdata` files numbered progressively that represent the fitting results for each pixel.
After the analysis, the function `extractParameters()` allows to extract from the fitting all relevant parameters that can be analysed. Here is an example of extraction of parameters from the ROI named `late` in the larch images shown above:

```
late.spatial <- extractParameters(path = late.path)
```

Arguments of the function `extractParameters()` include `path`, i.e. the path where temporary files are saved, `list`, alternatively to argument `path` if you have assigned the results from `spatialGreen()` in an object in the workspace, and `update` similar to the function `update` shown before, if you want to apply another threshold method to an already existing fit. Let's look at the structure of the object `late.spatial` to illustrate the parameters extracted from each fitted pixel:

```
str(late.spatial)
```

```
## 'data.frame':    11371 obs. of  19 variables:
##  $ UD           : num  NA 142 142 143 142 ...
##  $ SD           : num  NA 154 151 152 156 ...
##  $ DD           : num  NA 285 284 282 279 ...
##  $ RD           : num  NA 303 307 303 305 ...
##  $ maxline      : num  NA 0.46 0.471 0.471 0.474 ...
##  $ baseline     : num  NA 0.335 0.335 0.334 0.334 ...
##  $ prr          : num  NA 0.01055 0.01512 0.01596 0.00998 ...
##  $ psr          : num  NA -0.00627 -0.00498 -0.00588 -0.0044 ...
##  $ plateau.slope: num  NA -0.000124 -0.000194 -0.000158 -0.000241 ...
##  $ y0           : num  0.335 0.335 0.335 0.335 0.334 ...
##  $ a1           : num  0.121 0.125 0.136 0.136 0.141 ...
##  $ a2           : num  0.121 0.125 0.136 0.137 0.141 ...
##  $ t01          : num  139 134 137 138 134 ...
##  $ t02          : num  301 302 306 302 304 ...
##  $ b1           : num  2.25 4.25 3.18 3.03 5 ...
##  $ b2           : num  1.131 0.986 1.058 1.338 1.1 ...
##  $ c1           : num  25.2 19.1 14.5 18.4 13.1 ...
##  $ c2           : num  0.0915 0.0631 0.0469 0.0757 0.041 ...
##  $ RMSE         : num  0.00886 0.00993 0.01472 0.01016 0.00779 ...
```

It is actually a `data.frame` containing one row for each pixel and 19 columns with the extracted thresholds and the parameters of the curve fitting. Additionally, the root mean square error from the fitting is computed and extracted. Note that missing values in this data.frame are not surprising because some fitting may fail to converge, and hence return NA for all parameters and thresholds. Another cause of NA might be that the thresholds do not respect the expected chronology.

An ad hoc function was designed to display the results from the spatial analysis, `plotSpatial()`. Here is an example:

```
## roi.data.path <- '.../ROI/roi.data.Rdata'
## image.path <- '.../REF/20121004T1400.jpg'
plotSpatial(data = late.spatial, param = 'UD',
roi.data.path = roi.data.path,
roi.name = NULL, image.path = image.path,
probs=c(0.01, 0.99),
plot.density = FALSE, digits = 0)
```

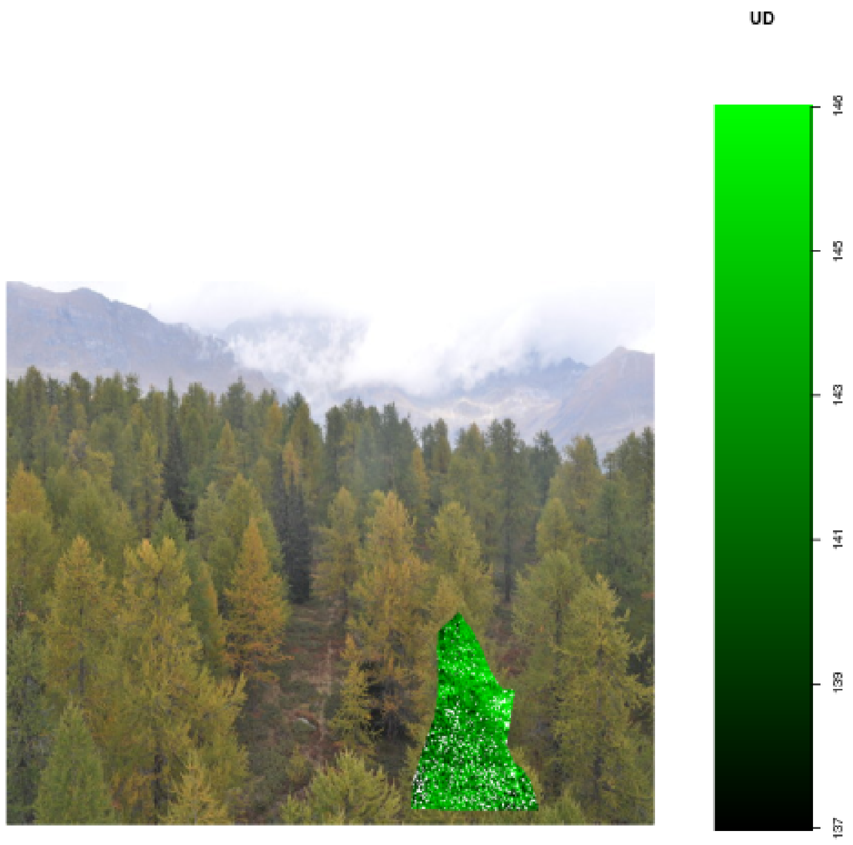

Figure 12: Spatial distribution of Upturn Date in the ROI named late, Torgnon Larch Stand 2012

In this plot the spatial distribution of `param` is displayed in a black to green scale superimposed on an image retrieved in `image.path` using the coordinates in `roi.data.path`. The argument `probs` allows to set limits to the range of `param` for plotting based on quantiles. The default is `c(0.01, 0.99)`, meaning that values lower than 1st percentile and higher than 99th percentile are removed before plotting. The argument `plot.density` allows to plot in the upper area of the plot a density distribution of the plotted parameter.
Upturn date in the ROI called late, which includes almost only one individual

tree can vary by as much as 10 days (fig.11). Now let's look together at the three ROIs where apparent differences in autumn phenology were observed (i.e. late, medium and early). To show results from more than one ROI with `plotSpatial` one has to first put all results in a list:

```
all.spatial <- list(early=early.spatial, late=late.spatial, medium=medium.spatial)
```

Note that it is important that the list be named, with names corresponding to ROI names.
Now we are ready to run:

```
plotSpatial(all.spatial, 'UD', roi.data.path, NULL,
            image.path, plot.density=FALSE, digits=0)
```

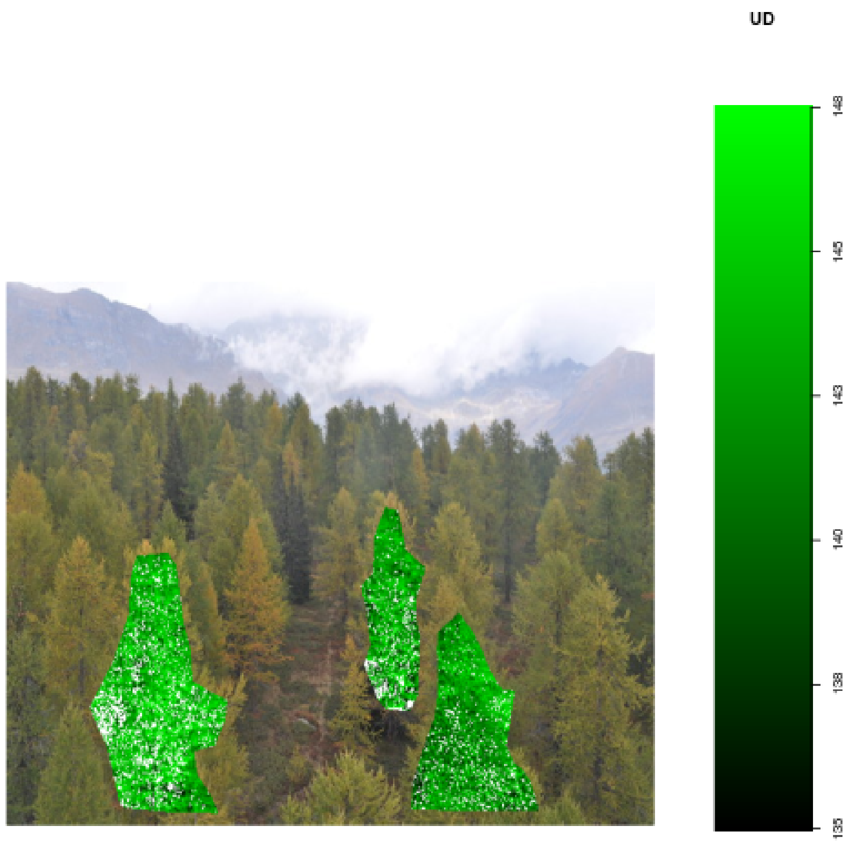

Figure 13: Spatial distribution of Upturn Date in all ROIs, Torgnon Larch Stand 2012

The plot in fig.12 shows no such a big difference in the spring phenology of the selected ROIs, the range of UD being slightly more than 10 days. A different pattern shows up when looking at autumn phenology (fig.13):

```
plotSpatial(all.spatial, 'RD', roi.data.path, NULL,
            image.path, plot.density=FALSE, digits=0)
```

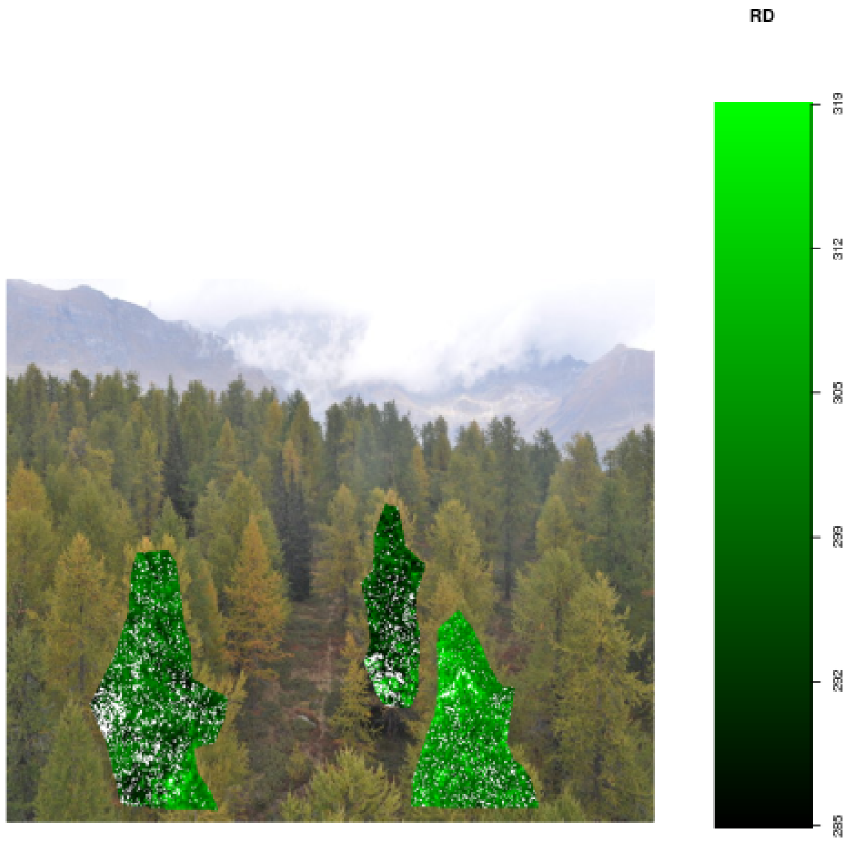

Figure 14: Spatial distribution of Recession Date in all ROIs, Torgnon Larch Stand 2012

Recession date ranges between doy 285 and 320, more than one month difference, clearly showing the expected pattern, with later autumn phenology in the ROI named late (on the right of the plot), earlier in center of the image, and in-between for the ROI on the left (fig.13). The white areas in the figure represent pixels where the fitting failed.

A more eterogeneous autumn than spring phenology in this larch stand was indeed a good tool to show the potential of pixel - based analysis of phenology and its functions. Image resolution is an important issue when deciding to run a pixel-based analysis. To reduce computer time it might be worth decrease image quality. However when the target vegetation is quite far from the camera (as in the larch site) such decrease may drammatically reduce the ability to detect phenological spatial patterns.

Summary and future of the package (to be extended)

`phenopix` package is currently available for download from the R-forge. The package was tested on approx 300 site-years belonging to the phenocam imagery archive, on the camera network of the project e-pheno and will soon be deployed to process images in the European Network of Flux Towers.

References (to be done)