EBImage - Image Processing Toolkit For R

Oleg Sklyar

April 3, 2006

1 Foreword

EBImage is image processing and analysis package for *R*. The primary goal of the package is to enable automated or semiautomated analysis of large sets of images, e.g. results of automated microscopy screens.

Many of *EBImage* routines are based on *ImageMagick* C++ library (*Magick++*) that provides all I/O operations as well as some filter routines. On top of *ImageMagick* further routines and algorithms are implemented to support *distance maps*, *thresholding*, *object counting* and other.

EBImage is not a wrapper for *ImageMagick*. First, in contrast to *ImageMagick* its purpose is to enable straightforward way to manipulate and analyse large image datasets. Images are represented as *R* objects that are directly derived from *R* arrays. This on one hand enables flexibility in data manipulation and analysis and on the other provides a way to apply different image processing routines to the data. Second, *EBImage* does not cover all the routines of *ImageMagick* and is limited to those that are essential and usefull for data analysis. In this way, it is not intended for image enhancing, restoration and similar. Furthermore, further in the development many *ImageMagick* routines will be substituted with native C++ code working directly with *R* data structures to reduce performance losses that occur during conversions between *R* and *ImageMagick* data structures.

2 System Requirements

EBImage like any image processing software is memory and CPU intensive. There are no formal limitations on system parameters, but it works with unpacked images, therefore sufficient memory is required to keep and process them. Some operations in *EBImage* require copies of loaded image to be produced and *R* automatically creates copies of its objects during function calls etc. As an example for memory usage, a single image of 800x600 pixel will need about 3.8Mb of memory just to store it and double as much will be needed to load it because there are two copies of the same unpacked image during loading, saving and displaying.

At the current developmen stage *EBImage* was designed and tested on Linux and Unix machines only. Both 32-bit and 64-bit platforms are supported. Installation of *EBImage* requires that development versions of *ImageMagick* and *Magick++* are installed on the system. Additional requirements include POSIX threads and Standard Template Library (STL), which are normally parts of any standard Linux installation.

```
> library("EBImage")

* EBImage of Bioconductor.org: help(EBImage) to get started...
```

3 Data Structures and Image I/O

Images in *EBImage* are stored in objects of class `Image2D` for 2D images and `Image3D` for image stacks (like TIFF files) or 3D images. `Image2D` class is derived directly from *R* `array`, thus it supports all operations that are defined for arrays. Most of standard operations were redefined in a way that the result of the operation is again `Image2D`. This inheritance also ensures that subscript operations applied to images are optimized for performance because those for `array` class are implemented in native C. Several additional methods are defined for `Image2D` over `array` to enable image IO, color mode conversions, data normalization etc.

Information about the color mode of the image can be accessed via its slot `rgb`.

It is advisable to use methods defined for `Image2D` to operate with image data, however using the inherited slot `.Data` will give a direct access to the data array!

Image data are stored as double values that should be normalized to the range [0..1] for grayscale or binary images. Colored (RGB) images are

stored as integer values, in byte-per-color representation. The highest byte must always be kept 0 in order to enable correct opacity transformations between *R* and *ImageMagick*. Whereas all mathematical operations are perfectly correct on grayscale images, RGB data is meaningless for many such operations. In some cases arithmetic operations of + and - are useful though also for RGB images as well.

EBImage can work with both locally stored images and those located on remote servers. Protocols supported include FTP, HTTP and all other supported by *ImageMagick*.

EBImage command `ping.image` provides an easy way to find out about attributes of images to be loaded without actually loading them:

```
> server = "http://www.ebi.ac.uk"
> file = "/~osklyar/projects/EBImage/examples/example.tif"
```

Images can be loaded with `read.image`. The function automatically discovers image types for every individual file in the `files` argument, calculates number of images and returns an `Image2D` or `Image3D` object as result. *EBImage* is mainly designed to work with grayscale images, therefore, if not specified otherwise `read.image` converts all data to grayscale on load. For instance, to load an example image from *EBImage* project page one can issue

```
> im = read.image(paste(server, file, sep = ""))
> im[1:5, 1:5, 1:2]

Image3D: 2 images of 5x5
        Type: grayscale, doubles in the range [0..1]
, , 1

          [,1]      [,2]      [,3]       [,4]       [,5]
[1,] 0.1488670 0.1376364 0.1067369 0.15167468 0.10392920
[2,] 0.1572900 0.1292134 0.1151598 0.08987564 0.07302968
[3,] 0.1263905 0.1292134 0.1207752 0.12358282 0.11515984
[4,] 0.1207752 0.1292134 0.1207752 0.10392920 0.10673686
[5,] 0.1432517 0.1292134 0.1292134 0.11796750 0.09268330

, , 2

          [,1]      [,2]      [,3]      [,4]      [,5]
[1,] 0.2019532 0.2150607 0.2221561 0.2090486 0.1981384
```

```
[2,] 0.2265202 0.2265202 0.2221561 0.2095979 0.2074159
[3,] 0.2494392 0.2298009 0.2188754 0.2057832 0.2166934
[4,] 0.2521706 0.2270695 0.2145113 0.2172427 0.2248875
[5,] 0.2483558 0.2385290 0.2270695 0.2172427 0.2232395
```

If the same image is loaded as RGB, the following output will be generated

```
> imRGB = read.image(paste(server, file, sep = ""), rgb = TRUE)
> imRGB[1:5, 1:5, 1:2]

Image3D: 2 images of 5x5
        Type: RGB, 8-bit per color
, , 1

        [,1]    [,2]    [,3]    [,4]    [,5]
[1,] 2434341 2302755 1776411 2500134 1710618
[2,] 2631720 2105376 1907997 1447446 1184274
[3,] 2105376 2105376 1973790 2039583 1907997
[4,] 1973790 2105376 1973790 1710618 1776411
[5,] 2368548 2105376 2105376 1973790 1513239

, , 2

        [,1]    [,2]    [,3]    [,4]    [,5]
[1,] 3355443 3552822 3684408 3487029 3289650
[2,] 3750201 3750201 3684408 3487029 3421236
[3,] 4144959 3815994 3618615 3421236 3618615
[4,] 4210752 3750201 3552822 3618615 3750201
[5,] 4144959 3947580 3750201 3618615 3684408
```

Images can be saved using `write.image` function of *EBImage*. If the source image is of class `Image3D`, the number of files specified can be either one if the target format supports image stacks or must exactly correspond to the number of images. For example, to save a small section of the image (printed above) into a single TIFF file, one needs to issue

```
> write.image(im[1:5, 1:5, 1:2], "testOutput.tif")

[1] "testOutput.tif"
```

Alternatively, to save two images into separate JPG files (or any other format)

```
> outputFiles = c("testOutput01.jpg", "testOutput02.jpg")
> write.image(im[1:5, 1:5, 1:2], outputFiles)
```

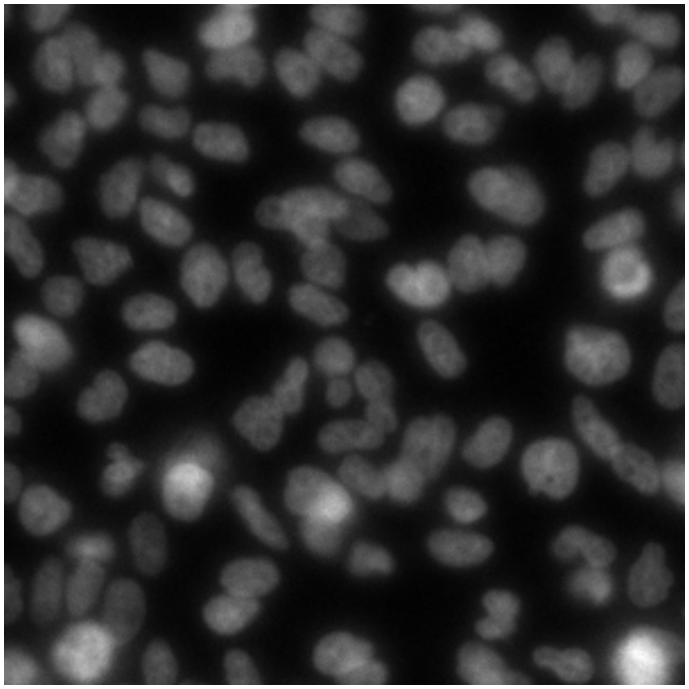
```
[1] "logo.jpg"          "testOutput01.jpg" "testOutput02.jpg"
```

There is no difference if the original imahe is RGB or grayscale, output file format is automatically adjusted.

4 Simple Image Data Manipulation and Image Arithmetics

As indicated in the previous section, `Image2D` and `Image3D` support all operations defined for arrays. Object `im` that was created in the previous section contains in fact 9 images, but for demonstration purposes we only want to work with one of those showing cell nuclei. One can easily obtain `Image2D` of interest from this `Image3D` in exactly the same manner as one would subscript arrays

```
> im1 = im[, , 3]
```



Let us consider we want manually treshold this image without turning to complex filters. First, one might need to determine the range of image data if they were not normalized beforehand

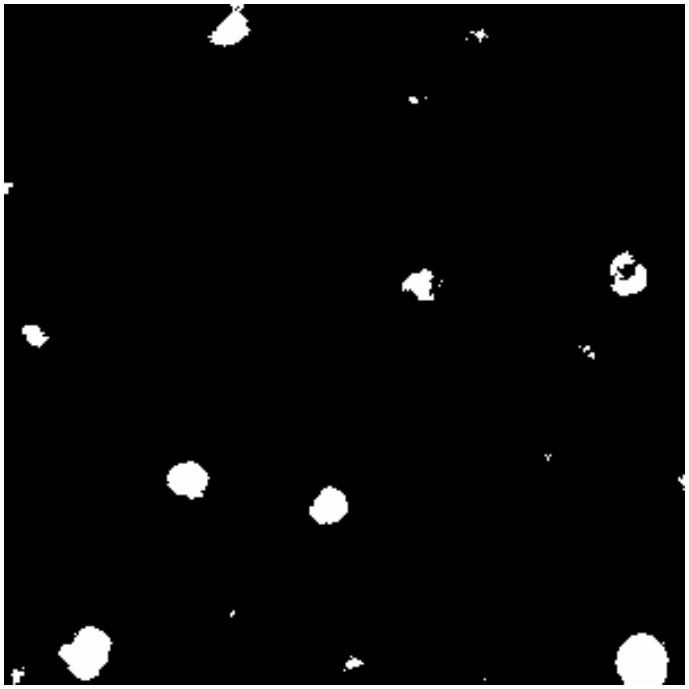
```
> range = minMax(im1)
> range

[1] 0 1

> treshold = (range[2] + range[1])/2
```

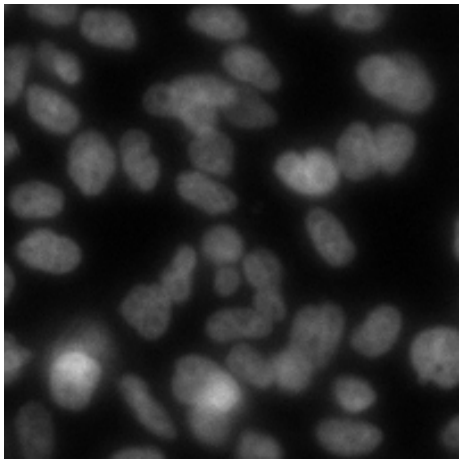
Simple tresholding can be realized simply by substituting values smaller than the treshold by 0 and larger than the treshold by 0.99 (there is a know bug that inverts binary images if they are defined with 0 and 1 only)

```
> im2 = im1
> im2[im2 < treshold] = 0
> im2[im2 >= treshold] = 0.99
```



In a similar manner images can be easily cropped to the required size. Suppose we only need a central part of `im1` of the size 200x200

```
> im2 = im1[51:250, 51:250]
```



... to be continued...

5 Simple Image Data Manipulation and Image Arithmetics

Vignette is under development! Please check the package web page for updates of the manual: `http://www.ebi.ac.uk/~osklyar/projects/EBImage`