```
$packages
      0.31MB       17.16MB        1.64MB
     "graph"          "GO"    "Biobase"
      1.45MB        1.29MB        0.23MB
  "annotate"        "RBGL"        "KEGG"
      0.23MB        0.22MB         1.2MB
"genefilter"    "Category"    "GOstats"

$total.size
[1] 23.739
```

Wrap Up

We have shown how to generate package dependency graphs and preview package installation using the **pkgDepTools** package. We have described in detail how the underlying code is used and the process of modeling relationships with the **graph** package.

These tools can help identify and understand interdependencies in packages. A very similar approach can be applied to visualizing class hierarchies in R such as those implemented using the S4 (Chambers, 1998) class system or Bengtsson's **R.oo** (Bengtsson, 2006) package.

The **graph**, **RBGL**, and **Rgraphviz** suite of packages provides a very powerful means of manipulating, analyzing, and visualizing relationship data.

Bibliography

H. Bengtsson. *R.oo: R object-oriented programming with or without references*, 2006. URL http://www.braju.com/R/. R package version 1.2.3.

V. Carey and L. Long. *RBGL: Interface to boost C++ graph lib*, 2006. URL http://bioconductor.org. R package version 1.10.0.

J. M. Chambers. *Programming with Data: A Guide to the S Language*. Springer-Verlag New York, 1998.

R. Gentleman, E. Whalen, W. Huber, and S. Falcon. *graph: A package to handle graph data structures*, 2006. R package version 1.12.0.

J. Gentry. *Rgraphviz: Provides plotting capabilities for R graph objects*, 2006. R package version 1.12.0.

E. Hartuv and R. Shamir. A clustering algorithm based on graph connectivity. *Information Processing Letters*, 76(4–6):175–181, 2000. URL citeseer.ist.psu.edu/hartuv99clustering.html.

M. Kanehisa and S. Goto. KEGG: Kyoto encyclopedia of genes and genomes. *Nucleic Acids Res*, 28: 27–30, 2000.

Seth Falcon
Program in Computational Biology
Fred Hutchinson Cancer Research Center
Seattle, WA, USA
emailsfalcon@fhcrc.org

Image Analysis for Microscopy Screens

Image analysis and processing with EBImage

by Oleg Sklyar and Wolfgang Huber

The package **EBImage** provides functionality to perform *image processing* and *image analysis* on large sets of images in a programmatic fashion using the R language.

We use the term *image analysis* to describe the extraction of numeric features (*image descriptors*) from images and image collections. Image descriptors can then be used for statistical analysis, such as classification, clustering and hypothesis testing, using the resources of R and its contributed packages.

Image analysis is not an easy task, and the definition of image descriptors depends on the problem. Analysis algorithms need to be adapted correspondingly. We find it desirable to develop and optimize such algorithms in conjunction with the subsequent statistical analysis, rather than as separate tasks. This is one of our motivations for writing the package.

We use the term *image processing* for operations that turn images into images, with the goals of enhancing, manipulating, sharpening, denoising or similar (Russ, 2002). While some image processing is often needed as a preliminary step for image analysis, image processing is not the primary aim of the package. We focus on methods that do not require interactive user input, such as selecting image regions with a pointer etc. Whereas interactive methods can be extremely effective for small sets of images, they tend to have limited throughput and reproducibility.

EBImage uses the `Magick++` interface to the ImageMagick (2006) image processing library to implement much of its functionality in image processing and input/output operations.

Cell-based assays

Advances in automated microscopy have made it possible to conduct large scale cell-based assays with image-type phenotypic readouts. In such an assay, cells are grown in the wells of a microtitre plate (often a 96- or 384-well format is used) under a condition or stimulus of interest. Each well is treated with one of the reagents from the screening library and the response of the cells is monitored, for which in many cases certain proteins of interest are antibody-stained or labeled with a GFP-tag (Carpenter and Sabatini, 2004; Wiemann et al., 2004; Moffat and Sabatini, 2006; Neumann et al., 2006).

The resulting imaging data can be in the form of two-dimensional (2D) still images, three-dimensional (3D) image stacks or image-based time courses. Such assays can be used to screen compound libraries for the effect of potential drugs on the cellular system of interest. Similarly, RNA interference (RNAi) libraries can be used to screen a set of genes (in many cases the whole genome) for the effect of their loss of function in a certain biological process (Boutros et al., 2004).

Importing and handling images

Images are stored in objects of class `Image` which extends the `array` class. The colour mode is defined by the slot `rgb` in `Image`; the default mode is grayscale.

New images can be created with the standard constructor `new`, or using the wrapper function `Image`. The following example code produces a 100x100 pixel grayscale image of black and white vertical stripes:

```
> im <- Image(0, c(100,100))
> im[c(1:20, 40:60, 80:100),,] = 1
```

By using ImageMagick, the package supports reading and writing of more than 95 image formats including JPEG, TIFF and PNG. The package can process multi-frame images (image stacks, 3D images) or multiple files simultaneously. For example, the following code reads all colour PNG files in the working directory into a single object of class `Image`, converts them to grayscale and saves the output as a single multi-frame TIFF file:

```
> files <- dir(pattern=".png")
> im <- read.image(files, rgb=TRUE)
> img <- toGray(im)
> write.image(img, "single_multipage.tif")
```

Besides operations on local files, the package can read from anonymous HTTP and FTP sources, and it can write to anonymous FTP locations. These protocols are supported internally by ImageMagick and do not use R-connections.

The storage mode of grayscale images is `double`, and all R-functions that work with arrays can be directly applied to grayscale images. This includes the arithmetic functions, subsetting, histograms, Fourier transformation, (local) regression, etc. For example, the sharpened image in Figure 1**c** can be obtained by subtracting the slightly blurred, scaled in colour version of the original image (Figure 1**b**) from its source in Figure 1**a**. All pixels that become negative after subtraction are then re-set to background. The source image is a subset of the original microscopic image. Hereafter, variables in the code are given the same literal names as the corresponding image labels (e.g. data of variable `a` are shown in Figure 1 **a**, `b` – in **b**, and `C` – in **c**, etc).

```
> orig <- read.image("ch2.png")
> a <- orig[150:550, 120:520,]
> b <- blur(0.5 * a, 80, 5)
> C <- a - b
> C[C < 0] = 0
> C <- normalize(C)
```

One can think of this code as of a naive, but fast and effective, version of the *unsharp mask* filter; a more sophisticated implementation from the ImageMagick library is provided by the function `unsharpMask` in the package.

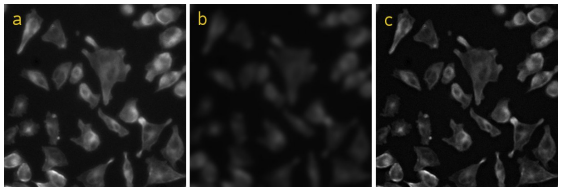

Figure 1: Implementation of a simple *unsharp mask* filter: (a) source image, (b) blurred colour-scaled image, (c) sharpened image after normalization

Some of the image analysis routines assume grayscale data in the interval $[0,1]$, but formally there are no restrictions on the range.

The storage mode of RGB-images is `integer`, and we use the three lowest bytes to store red (R), green (G) and blue (B) values, each in the integer-based range of $[0,255]$. Because of this, arithmetic and other functions are generally meaningless for RGB-images; although they can be useful in some special cases, as shown in the example code in the following section. Support for RGB-images is included to enhance the display of the analysis results. Most analysis routines require grayscale data though.

Image processing

The ImageMagick library provides a number of image processing routines, so-called *filters*. Many of those are ported to R by the package. The missing ones

may be added at a later stage. We have also implemented additional image processing routines that we found useful for work on cell-based assays.

Filters are implemented as functions acting on objects of class `Image` and returning a new `Image`-object of the same or appropriately modified size. One can divide them into four categories: image *enhancement*, *segmentation*, *transformation* and *colour correction*. Some examples are listed below.

sharpen, unsharpMask generate sharpened versions of the original image.

gaussFilter applies the Gaussian blur operator to the image, softening sharp edges and noise.

thresh segments a grayscale image into a binary black-and-white image by the adaptive threshold algorithm.

mOpen, mClose use erosion and dilation to enhance edges of objects in binary images and to reduce noise.

distMap performs a Euclidean distance transform of a binary image, also known as *distance map*. On a distance map, values of pixels indicate how far are they away from the nearest background. Our implementation is adapted from the Scilab image processing toolbox (SIP Toolbox, 2005) and is based on the algorithm by Lotufo and Zampirolli (2001).

normalize shifts and scales colours of grayscale images to a specified range, normally $[0, 1]$.

sample.image proportionally resizes images.

The following code demonstrates how grayscale images recorded using three different microscope filters (Figure 2 **a**, **b** and **c**) can be put together into a single *false-colour* representation (Figure 2 **d**), and conversely, how a single false-colour image can be decomposed into its individual channels.

```
> files <- c("ch1.png","ch2.png","ch3.png")
> orig <- read.image(files, rgb=FALSE)
> abc <- orig[150:550, 120:520,]
> a <- toGreen(abc[,,1])          # RGB
> b <- toRed(abc[,,2])            # RGB
> d <- a + b + toBlue(abc[,,3])
> C <- getBlue(d)                 # gray
```

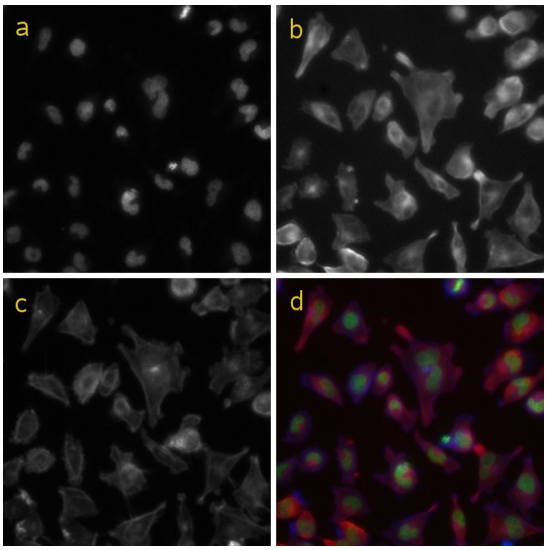

Figure 2: Composition of a false-colour image (d) from a set of grayscale microscopy images for three different luminescent compounds: (a) – DAPI, (b) – tubulin and (c) – phalloidin

Displaying images

The package defines the method `display` which shows images in an interactive X11 window, where image stacks can be animated and browsed through. This function does not use R graphics devices and cannot be redirected to any of those. To redirect display into an R graphics device, the method `plot.image` can be used, which is a wrapper for the `image` function from the **graphics** package. Since each pixel is drawn as a polygon, `plot.image` is much slower compared to `display`; also, it shows only the first image of a stack:

```
> display(abc)          # displays all 3
> plot.image(abc[,,2]) # can display just 1
```

Drawables

Pixel values can be set either by using the conventional subset assignment syntax for arrays (as in the third code example, `C[C < 0] = 0`) or by using *drawables*. **EBImage** defines the following instantiable classes for drawables (derived from the virtual `Drawable`): `DrawableCircle`, `DrawableLine`, `DrawableRect`, `DrawableEllipse` and `DrawableText`. The stroke and fill colours, the fill opacity and the stroke width can be set in the corresponding slots of `Drawable`. As the opportunity arises, we plan to provide drawables for text, poly-lines and polygons. Drawables can be drawn on `Images` with the method `draw`; both grayscale and RGB images are supported with all colours automatically converted to gray levels on grayscale images.

The code below illustrates how drawables can be used to mark the positions and relative sizes of the nuclei detected from the image in Figure 2**a**. It assumes that `x1` is the result of the function `wsObjects`, which uses a watershed-based image segmentation for object detection. `x1` contains matrix `objects` with object coordinates (columns 1 and 2) and areas (column 3). The resulting image is shown in Figure 3**b**. This is just an illustration, we do not assume circular shapes of nuclei. For comparison, the actual segmentation boundaries are colour-marked in Figure 3**a** using the function `wsPaint`:

```
> src <- toRGB(abc[,,1])
> x <- x1$objects[,1]
> y <- x1$objects[,2]
> r <- sqrt(x1$objects[,3] / pi)
> cx <- DrawableCircle(x, y, r)
> b <- draw(src, cx)
> a <- wsPaint(x1, src)
```

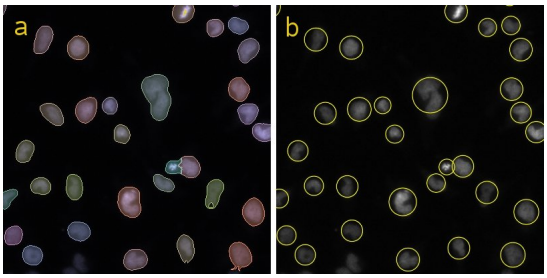

Figure 3: Colour-marked nuclei detected with function `wsObjects`: (a) – as detected, (b) – illustrated by `DrawableCircle`'s.

Analysing an RNAi screen

Consider an experiment in which images like those in Figure 2 were recorded for each of ≈ 20,000 genes, using a whole-genome RNAi library to test the effect of gene knock-down on cell viability and appearance. Among the image descriptors of interest are the number, position, size, shape and the fluorescent intensities of cells and nuclei.

The package provides functionality to identify objects in images and to extract image descriptors in the function `wsObjects`. The function identifies different objects in parallel using a modified watershed-based segmentation algorithm and using image distance maps as input. The result is a list of three matrix elements `objects`, `pixels` (indices of pixels constituting the objects) and `borders` (indices of pixels constituting the object boundaries). If the supplied image is an image stack, the result is a list of such lists. Each row in the matrix `objects` corresponds to a detected object, with different object descriptors in the columns: x and y coordinates, size, perimeter, number of pixels on the image edge, acircularity, effective radius, perimeter to radius ratio, etc. Objects on the image edges can be automatically removed if the ratio of the detected edge pixels to the perimeter is larger than a given threshold. If the original image, from which the distance map was calculated, is specified in the argument `ref`, the overall intensity of the object region is calculated as well.

For every gene, the image analysis workflow looks, therefore, as follows: load and normalize images, segment, enhance the segmented images by morphological opening and closing, generate distance maps, identify cells and nuclei, extract image descriptors, and, finally, generate image previews with the identified objects marked.

Object descriptors can then be analysed statistically to cluster genes by their phenotypic effect, generate a list of genes that should be studied further in more detail (hit list), e.g., genes that have a specific phenotypic effect of interest, etc. The image previews can be used to verify and audit the performance of the algorithm through visual inspection.

A schematic implementation is illustrated in the following example code and in Figure 4. Here we omit the step of nuclei detection (object `x1`), from where the matrix of nuclei coordinates (object `seeds`) is retrieved to serve as starting points for the cell detection. The nuclei detection is done analogously to the cell detection without specifying starting points.

```
> for (X in genes) {
+  files <- dir(pattern=X)
+  orig <- read.image(files)
+  abc <- normalize(orig, independent=TRUE)
+  i1 <- abc[,,1]
+  i2 <- abc[,,2]
+  i3 <- abc[,,3]
+  a <- sqrt(normalize(i1 + i3))
+  b <- thresh(a, 300, 300, 0.0, TRUE)
+  C <-  mOpen(b, 1, mKernel(7))
+  C <- mClose(C, 1, mKernel(7))
+  d <- distMap(C)
+  # x1 <- wsObjects(...) - nuclei detection
+  seeds <- x1$objects[,1:2]
+  x2 <- wsObjects(d, 30, 10, .2, seeds, i3)
+  rgb <- toGreen(i1)+toRed(i2)+ toBlue(i3)
+  e <- wsPaint(x2, rgb, col="white",fill=F)
+  f <- wsPaint(x2, i3, opac = 0.15)
+  f <- wsPaint(x1, f, opac = 0.15)
+ }
```

Note that here we adopted the *record-at-a-time* approach: image data, which can be huge, are stored on a mass-storage device and are loaded into RAM in portions of just a few images at a time.

Summary

EBImage brings image processing and image analysis capabilities to R. Its focus is the programmatic

(non-interactive) analysis of large sets of similar images, such as those that arise in cell-based assays for gene function via RNAi knock-down. Image descriptors can be analysed further using R's functionalities in machine learning (clustering, classification) and hypothesis testing.

Our current work on this package focuses on more accurate object detection and algorithms for feature/descriptor extraction. Image registration, alignment and object tracking are of foreseeable interest. In addition, one can imagine many other useful features, for example, support for more ImageMagick functions, better display options (e.g., using GTK) or interactivity. Contributions or collaborations on these or other topics are welcome.

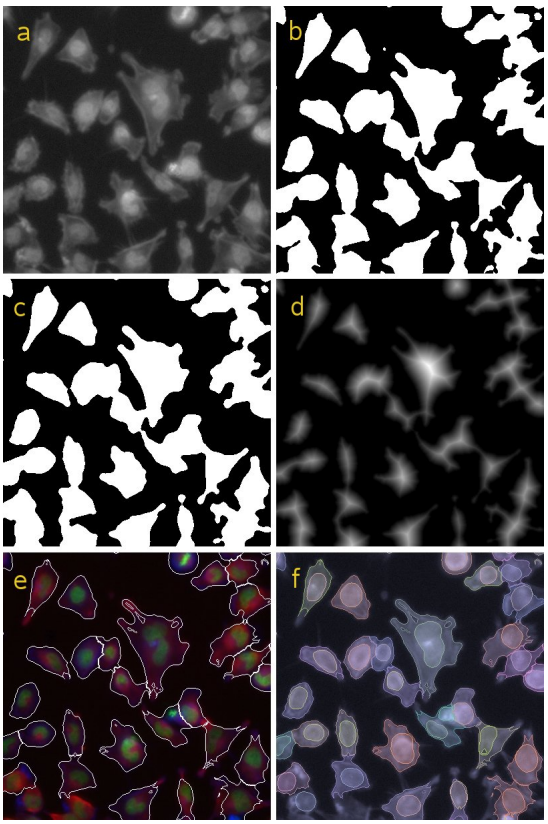

Figure 4: Illustration of the object detection algorithm: (a) – *sqrt*-brightened combined image of nuclei (DAPI from Figure 2**a**) and cells (phalloidin from Figure 2**c**); (b) – image **a** after *blur* and *adaptive thresholding*; (c) – image **b** after morphological *opening* followed by *closing*; (d) – normalized *distance map* generated from image **c**; (e) – outlines of cells detected using `wsObjects` drawn on top of the RGB image from Figure 2**d**; (f) – colour-mapped cells and nuclei as detected with `wsObjects` (one unique colour per object)

Installation

The package depends on ImageMagick, which needs to be present on the system to install the package. Please refer to the 'INSTALL' file.

Acknowledgements

We thank F. Fuchs and M. Boutros for providing their miscroscopy data and for many stimulating discussions about the technology and the biology, R. Gottardo and F. Swidan for testing the package on MacOS X and the European Bioinformatics Institute (EBI), Cambridge, UK, for financial support.

Bibliography

M. Boutros, A. Kiger, S. Armknecht, *et al.* Genome-wide RNAi analysis of cell growth and viability in Drosophila. *Science*, 303:832–835, 2004.

A. E. Carpenter and D.M. Sabatini. Systematic genome-wide screens of gene function. *Nature Reviews Genetics*, 5:11–22, 2004.

ImageMagick: software to convert, edit, and compose images. *Copyright: ImageMagick Studio LLC*, 1999-2006. URL http://www.imagemagick.org/

R. Lotufo and F. Zampirolli. Fast multidimensional parallel Euclidean distance transform based on mathematical morphology. *SIBGRAPI-2001/Brazil*, 100–105, 2001.

J. Moffat and D.M. Sabatini. Building mammalian signalling pathways with RNAi screens. *Nature Reviews Mol. Cell Biol.*, 7:177–187, 2006.

B. Neumann, M. Held, U. Liebel, *et al.* High-throughput RNAi screening by time-lapse imaging of live human cells. *Nature Mathods*, 3(5):385–390, 2006.

J. C. Russ. The image processing handbook – 4th ed. CRC Press, *Boca Raton*. 732 p., 2002

SIP Toolbox: Scilab image processing toolbox. *Sourceforge*, 2005. URL http://siptoolbox.sourceforge.net/

S. Wiemann, D. Arlt, W. Huber, *et al.* From ORFeome to biology: a functional genomics pipeline. *Genome Res.* 14(10B):2136–2144, 2004.

EBimage:R

Oleg Sklyar and Wolfgang Huber
European Bioinformatics Institute
European Molecular Biology Laboratory
Wellcome Trust Genome Campus
Hinxton, Cambridge
CB10 1SD
United Kingdom
osklyar@ebi.ac.uk; huber@ebi.ac.uk