

Analysis of microscopy images and cellular phenotypes using EBIImage

Grégoire Pau
European Molecular Biology Laboratory, Heidelberg, Germany

July 23, 2009

Contents

1	Motivation	1
2	Preprocessing images and visualisation	1
3	Cell segmentation	3
3.1	Nucleus segmentation by thresholding	3
3.2	Cell membrane determination	4
3.3	Batch processing	6
4	Cellular features	6

1 Motivation

EBImage is a Bioconductor package which provides general purpose functionality for the reading, writing, processing and analysis of images. In this tutorial, we use **EBImage** to transform images coming from the microscope, segment cells, extract quantitative cellular descriptors and predict phenotypes from cellular populations.

We are using a set of 8 human HeLa cell populations perturbed by small interfering RNAs (siRNA) targeting different genes (Table 1). After transfection, cells were grown for 48 h, stained with three immunofluorescent markers (DNA, tubulin and actin) and imaged by microscopy.

Image name	Target gene	Expected phenotype
001-01-C03	<i>Rluc</i>	Wild type
001-01-D03	<i>Rluc</i>	Wild type
001-01-E04	KIF11	Mitotic arrest
001-01-F04	KIF11	Mitotic arrest
001-01-I04	CLSPN	Giant cells
001-01-M04	<i>Rluc</i>	Wild type
002-01-B20	PKM2	Undefined
003-01-H03	TRAPPC3	Elongated cells

Table 1: Experiment/perturbation mapping.

2 Preprocessing images and visualisation

After having started R, the package **EBImage** is loaded by typing the following command.

```
> library("EBImage")
```

The function **readImage** reads images and supports many image formats, including JPEG, TIFF, GIF and PNG. The following example reads a microscopy image and assigns it to the variable **a**. The image **a** is then displayed using **display**.

```
> a = readImage('images/001-01-C03-actin.jpeg')
> display(a)
```

The image `a` shows the actin channel of a population of HeLa cells. We now read the DNA channel and assign it to the variable `d`.

```
> d = readImage('images/001-01-C03-dna.jpeg')
> display(d)
```

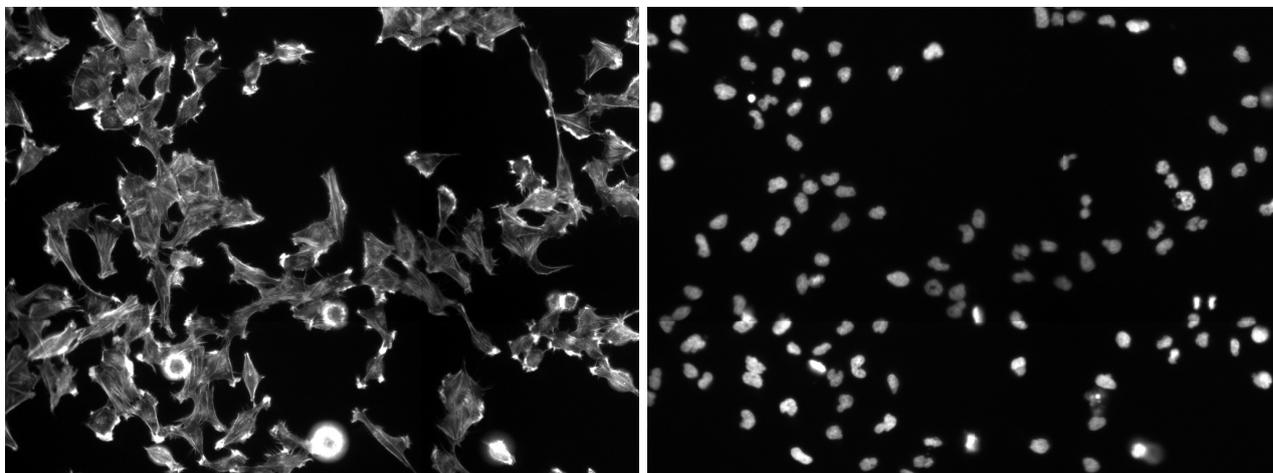


Figure 1: The actin (`a`, left) and DNA (`d`, right) channels of the cell population “001-01-C03”.

Images are represented as matrices. Hence, they can be manipulated in R with algebraic operators such as sum, product or comparison, with numbers or other matrices. These basic operations are the building blocks of all image processing operators and can be used to adjust the dynamic range of an image.

```
> display(a*2)
> display(a+d)
> display(d+0.3)
```

EImage supports many geometric transformations such as resizing or rotation.

```
> display(resize(a, w=256))
> display(rotate(a, 30))
```

Gray images can be combined into color images using the function `rgbImage`. The following example shows how to combine the tree fluorescent channels of the image “001-01-C03” into the RGB color image `x`.

```
> t = readImage('images/001-01-C03-tubulin.jpeg')
> x = rgbImage(red=a, green=t, blue=d)
> display(x)
```

Images can be written using `writeImage`. The output image format is guessed according to the filename extension.

```
> writeImage(x, 'images/001-01-C03-rgb.jpeg')
```

Automatic image processing is useful when batch processing thousand of images. The following script combines the color channels of the 8 experiments and writes corresponding the output “-rgb.jpeg” images.

```
> ids = unique(substr(dir('images'), 1, 10))
> for (id in ids) {
+   cat(id, '\n')
+   a = readImage(file.path('images', paste(id, '-actin.jpeg', sep='')))
+   t = readImage(file.path('images', paste(id, '-tubulin.jpeg', sep='')))
+   d = readImage(file.path('images', paste(id, '-dna.jpeg', sep='')))
+   x = rgbImage(red=a, green=t, blue=d)
+   writeImage(x, file.path('images', paste(id, '-rgb.jpeg', sep='')))
+ }
```

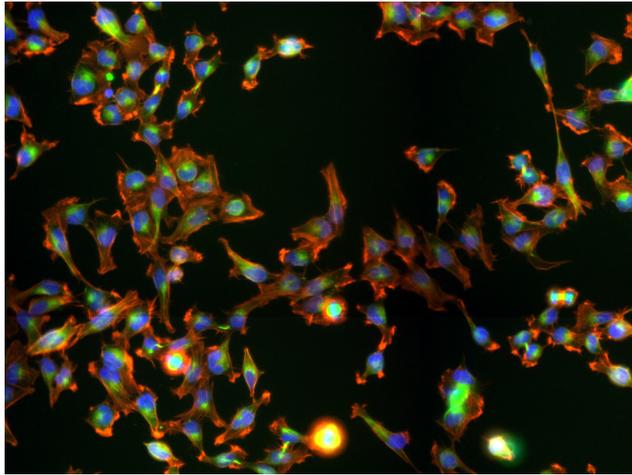


Figure 2: The cell population “001-01-C03” (x) showing actin (red), tubulin (green) and DNA (blue) components.

The new RGB combined images can be viewed using an external viewer. A more convenient option is to generate a web page showing all the thumbnail images. The following script generates the thumbnails by reducing all the images located of the directory “images”.

```
> dir.create('thumbs')
> img = dir('images')
> for (i in img) {
+   cat(i, '\n')
+   y = readImage(file.path('images', i))
+   writeImage(resize(y, w=256), file.path('thumbs', i))
+ }
```

A web page, showing all the images with thumbnails, is generated using the HTML package `hwriter`.

```
> library(hwriter)
> m1 = matrix(file.path('thumbs', img), nrow=length(ids), byrow=TRUE)
> m2 = matrix(file.path('images', img), nrow=length(ids), byrow=TRUE)
> m1 = hwriteImage(m1, link=m2, table=FALSE)
> rownames(m1) = ids
> hwrite(m1, 'gallery.html')
> browseURL('gallery.html')
```

3 Cell segmentation

Segmenting objects from an image is a standard task in image analysis. Depending on the kind of objects to be isolated, segmentation algorithms can range from really basic ones to very complex models. The images we are analysing here are simple: each cell contains a nucleus, and nuclei are not superposed, well separated and lie in a dark, uniform medium. Thresholding-based segmentation algorithms are simple and powerful enough to separate the nuclei in our images. We are using a two-step algorithm where the nuclei are segmented first, followed by the determination of the cell membranes.

3.1 Nucleus segmentation by thresholding

Global thresholding is a simple segmentation method which consists in finding a threshold that separates the objects from the background. The next example uses global thresholding to segment the nuclei from the DNA channel `d`.

```
> display(d)
> nmask1 = d>0.15
> display(nmask1)
```

Results are correct but some nuclei next to each other have been fused by the algorithm. A variant of global thresholding, adaptive thresholding, consists in comparing the intensity of every pixels to their neighborhood, resulting in a binary image. Adaptive thresholding is efficiently implemented by the function `thresh`.

```
> nmask2 = thresh(d, 15, 15, 0.005)
> display(nmask2)
> display(rgbImage(red=nmask1, green=nmask2))
```

The results are better, as shown by the red and green image comparing the two segmentation approaches. Remaining single pixels or small artefacts from the mask `nmask2` are removed using the function `opening`, which performs a morphological opening on `nmask2`, rubbing the foreground by the structural element `mk3`. The function `fillHull` removes internal holes in objects.

```
> mk3 = makeBrush(3, shape='diamond')
> nmask2 = opening(nmask2, mk3)
> nmask2 = fillHull(nmask2)
> display(nmask2)
```

Nuclei are now extracted from the mask using `bwlabel`, which labels connected sets in a binary image.

```
> nseg = bwlabel(nmask2)
> nn = max(nseg)
> display(nseg/nn)
> print(nn)
```

`nseg` contains a matrix where every object is defined by a group of pixels set to the same unique increasing integer, starting with 0 for the background. Therefore, the total number of objects is given by the maximum integer of the segmented image. Here, 147 objects were found but many are too small to be considered *a priori* as nuclei. The following step computes the geometric features of the nuclei and prints them. Features include coordinates (`g.x`, `g.y`), size in pixels `g.s` and perimeter `g.p`.

```
> nf = hullFeatures(nseg)
> print(nf)
```

The objects with a size lower than 150 pixels cannot be considered as nuclei and are removed using `rmObjects`. 121 nuclei are now present.

```
> nr = which(nf[, 'g.s'] < 150)
> nseg = rmObjects(nseg, nr)
> nn = max(nseg)
> print(nn)
> display(nseg/nn)
```

3.2 Cell membrane determination

Since cells are touching, membrane determination cannot be done by thresholding. We are using instead the Voronoi segmentation algorithm, starting with seeds (here the nuclei) and propagating the cell membranes according to a metric based on the image gradient. Before doing so, we have to compute the cell mask `cmask`, indicating whether each pixel belongs to a cell or not. Cell mask computation is done by adaptive thresholding, through the function `filter2` which convolutes an image with a 2D filter.

```
> mix = sqrt(a^2+t^2+d^2)
> display(mix)
> cmask = filter2(mix, mk3/5)>=0.12
> cmask = closing(cmask, mk3)
> nseg[!(cmask>0)] = 0
> display(rgbImage(red=cmask, green=nseg))
> display(x)
```

The Voronoi segmentation is performed by `propagate`, using the nuclei as seeds.

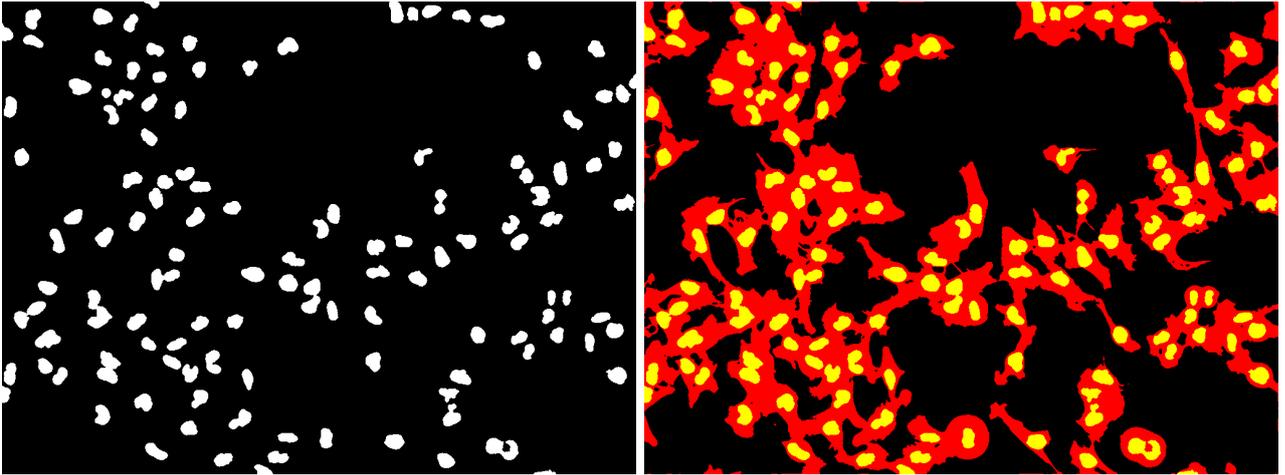


Figure 3: Segmented nuclei (`nseg`, left). Cell mask (red) and nuclei (yellow) (`rgbImage(red=cmask, green=nseg)`, right).

```
> cseg = propagate(mix^0.2, nseg, cmask, 0.1, 0)
> cseg = fillHull(cseg)
> display(cseg/max(cseg))
```

Cells with a size lower than 150 pixels are removed. Cell features are stored in the list `acf`.

```
> cf = hullFeatures(cseg)
> cr = which(cf[, 'g.s'] < 150)
> cseg = rmObjects(cseg, cr)
> nseg = rmObjects(nseg, cr)
> acf = list()
> acf[[id]] = cf

> print(max(cseg))
> seg = paintObjects(nseg, x, col=c('#ffff00'))
> seg = paintObjects(cseg, seg, col=c('#ff00ff'))
> display(seg)
> writeImage(seg, file.path('images', paste(id, '-seg.jpeg', sep='')))
```

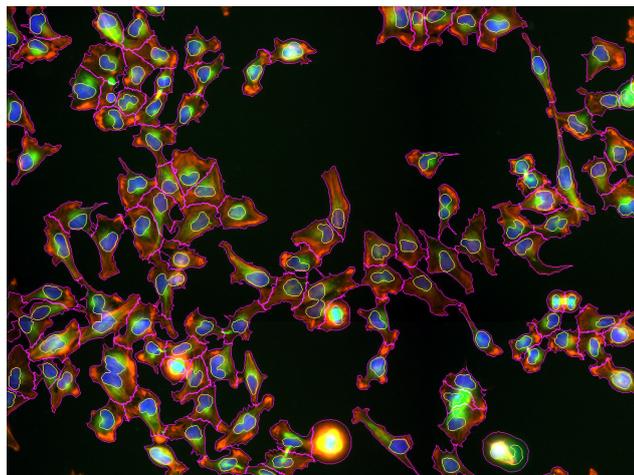


Figure 4: Final segmentation (`seg`). Nuclei are highlighted in yellow and cell boundaries in magenta.

3.3 Batch processing

Using the previous examples, write a script that, for each of the 8 images, segment it, write the segmented image “-seg.jpeg”, compute the cell features and store them into the list `acf`. The following example writes a web page containing the 8 images, before and after segmentation.

```
> img = dir('images')
> img = img[sort(c(grep('seg', img), grep('rgb', img)))]
> i2 = file.path('images', img)
> m2 = matrix(i2, nrow=length(ids), byrow=TRUE)
> m2 = hwriteImage(m2, table=FALSE)
> rownames(m2) = ids
> hwrite(m2, 'gallery2.html')
> browseURL('gallery2.html')
```

4 Cellular features

Cellular features characterize cell morphologies and harbor biologically relevant information. The following example displays the distribution of cell perimeters `g.s` in 4 images.

```
> library(geneplotter)
> cp = sapply(acf, function(ac) ac[, 'g.p'])
> multidensity(cp[c('001-01-C03', '001-01-D03', '001-01-M04', '003-01-H03')], xlim=c(0,1000), bw=100,
+             xlab='Cell perimeter (in pixels)', main='Distribution of cell perimeters')
```

Judging from the distributions, it seems that the cells of the image “003-01-H03” are longer than the ones in “001-01-C03”, “001-01-D03” and “001-01-M04”. This phenotypic difference can be formally tested. The following example shows that the cells of the image “003-01-H03” are significantly longer than the ones in “001-01-C03” (Wilcoxon rank sum test, P-value < 1e-6).

```
> wilcox.test(cp[['001-01-C03']], cp[['003-01-H03']])
> wilcox.test(cp[['001-01-C03']], cp[['001-01-M04']])
```

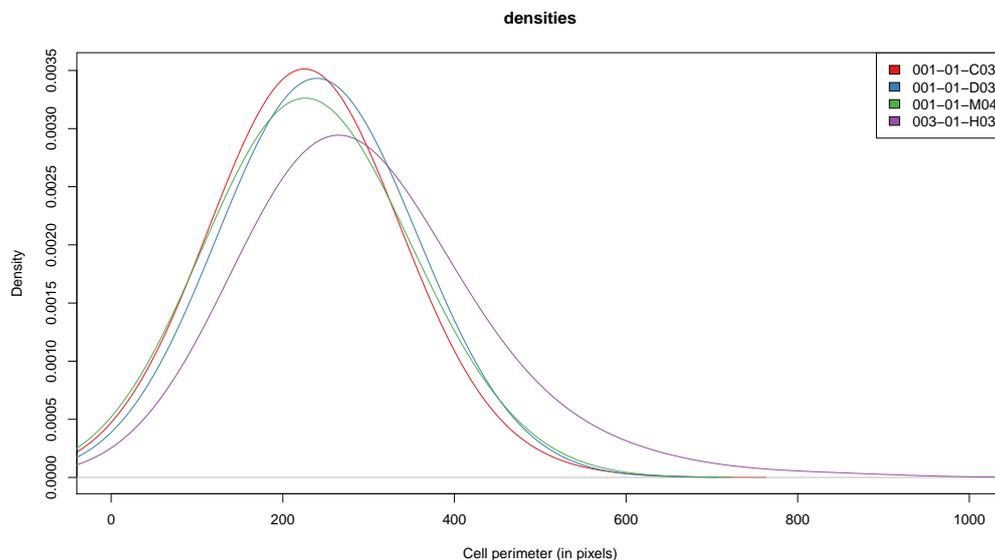


Figure 5: Distribution of cell perimeters in 4 cell populations.