Analysis of biological images with EBImage

Oleg Sklyar, Wolfgang Huber
osklyar@ebi.ac.uk

November 15, 2007

In this manual the EBImage package is used to analyse a set of images acquired in a large-scale RNAi microscopy screen and to extract numerical descriptors of the cells in these images. The descriptors define biological phenotypes. The descriptors can be further analysed statistically e. g. to classify genes by their phenotypic effect.

The images used in this manual are available in the package lab can be downloaded from
http://www.ebi.ac.uk/~osklyar/BioC2007/data

The images are 16-bit grayscale TIFF's and were measured at two different wavelengths corresponding to two different stainings: DNA content measured in the green channel (suffix G) and one of a cytoplasm protein in red (suffix R).

1 Handling images

EBImage provides two functions to read images, `readImage` and `chooseImage`. They both allow users to specify whether the result should be in grayscale or RGB mode.

An interactive window for loading images is provided by the `chooseImage` function, which is available if the package was compiled with GTK+ support (always the case on Windows):

```
> x = chooseImage(TrueColor)
```

The `readImage` function can read local or network files via HTTP or anonymous FTP protocols. Multiple files can be loaded into a single object, an image stack. If the source image has multiple frames they all will be read into a stack. Single, multi-frame or multiple images read are stored in objects of class `Image`:

```
> ddir <- paste(system.file(package = "EBImage"), "images", sep = "/")
> fG = paste(ddir, dir(path = ddir, pattern = "_G.tif"), sep = "/")
> iG = readImage(fG[1], Grayscale)
> class(iG)

[1] "Image"
attr(,"package")
[1] "EBImage"

> dim(iG)

[1] 508 508   4

> fR = paste(ddir, dir(path = ddir, pattern = "_R.tif"), sep = "/")
> iR = readImage(fR[1])
```

Images can be read from remote URL's as in the following example:

```
> baseurl = "http://www.ebi.ac.uk/~osklyar/BioC2007/data/"
> a = readImage(paste(baseurl, c("Gene1_R.tif", "Gene2_R.tif"),
+     sep = ""))
```

2 Exploring data

Objects of the class *Image* can be displayed using either the `display` function or the standard `image` method for the *Image* class:

```
> image(iG[, , 1])
> display(iR)
> animate(iR)
```

Beside displaying one can use `hist` and standard `print` methods to explore the data contained in images. For example, for the image stack `iR` consisting of 4 images, individual histograms are obtained with the following code (results shown in Figure 1):

```
> split.screen(c(2, 2))
> for (i in 1:4) {
+     screen(i)
+     hist(iR[, , i], xlim = c(0, 1))
+ }
> close.screen(all = TRUE)
```

Try additionally the following commands to investigate the structure of the data:

```
> str(iR)
> print(iR)
> dim(iR)
> range(iR)
```

Try also to do the same for `iG`, the images of the DNA content.

3 Image processing

The `normalize` function allows you to perform normalization of image data to a given range (the default is $[0, 1]$). Images in a stack can be normalized either separately from each other or simultaneously as parts of a large dataset. In our case, we want to normalize each image separately. Try the following two normalizations and compare the outputs:

```
> iRn.wrong = normalize(iR, separate = FALSE)
> apply(iRn.wrong, 3, range)

          [,1]       [,2]       [,3]       [,4]
[1,] 0.0000000 0.03076923 0.04615385 0.06923077
[2,] 0.2615385 0.51538462 1.00000000 0.94615385

> iRn = normalize(iR, separate = TRUE)
> apply(iRn, 3, range)

     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    1    1    1    1

> iGn = normalize(iG, separate = TRUE)
```

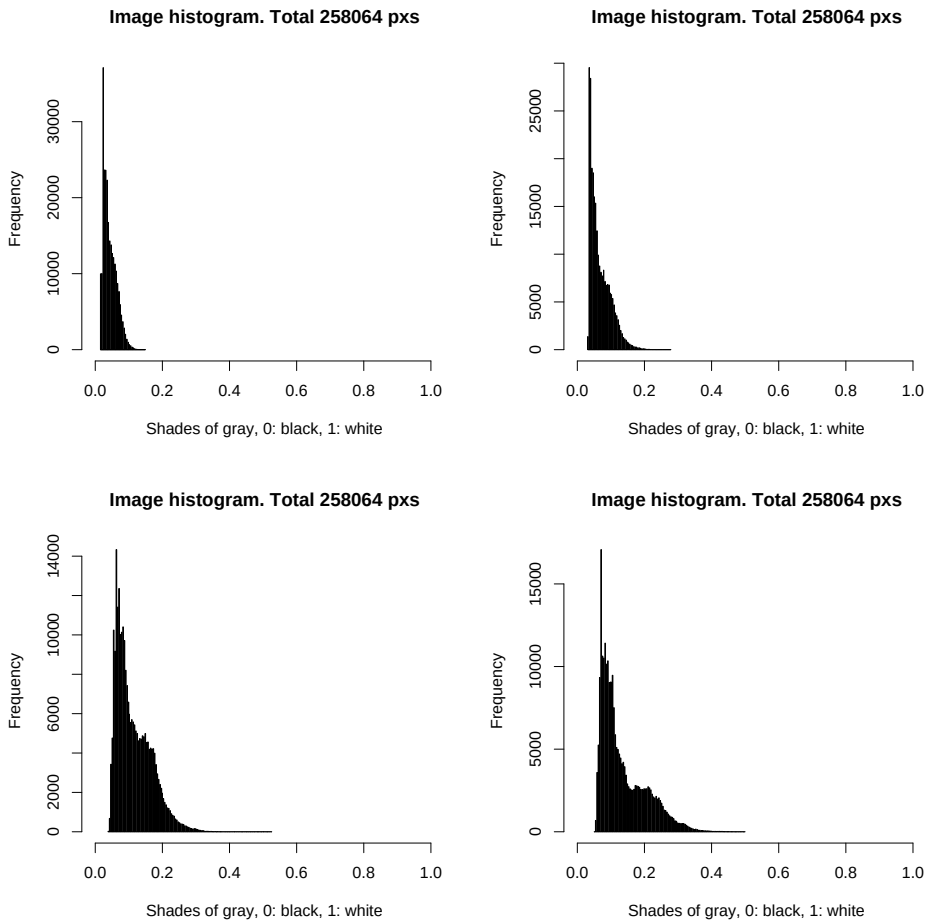

Figure 1: Histograms of individual images in 'Gene1_R.tif'. The frames have significantly different dynamic ranges and need to be normalized before intensity-based descriptors can be used.

EBImage provides a set of functions to manipulate the quality of images like `enhance`, `contrast`, `despeckle` or `denoise`. Try them out on the images in `iRn`. Note that such manipulations can be useful for visualisation, but they are not appropriate for quantitative analyses, such as when intensity-derived features are used as phenotypic descriptors.

We can also apply non-linear transformations to improve image quality. Another reason for some of these transformations is to achieve a higher object contrast against the background, which turns useful in consequent image segmentation. Please bear in mind the same note: images transformed in such a way cannot (should not) be used to extract intensity-based descriptors and are assumed for visualisation or segmentation purposes.

Let us define several functions for transforming the range $[0, 1]$ into itself.

```
> modif1 = function(x) sin((x - 1.2)^3) + 1
> modif2 = function(x, s) (exp(-s * x) - 1)/(exp(-s) - 1)
> modif3 = function(x) x^1.5
```

The graphs of these functions are shown in Figure 2 on page 4. Non-linear transformations of the range $[0, 1]$ into itself are often used in image processing to prepare an image for subsequent operations. I remark as a sidenote that this is also one of the most useful filters in Photoshop or GIMP when it comes to processing holiday photos.

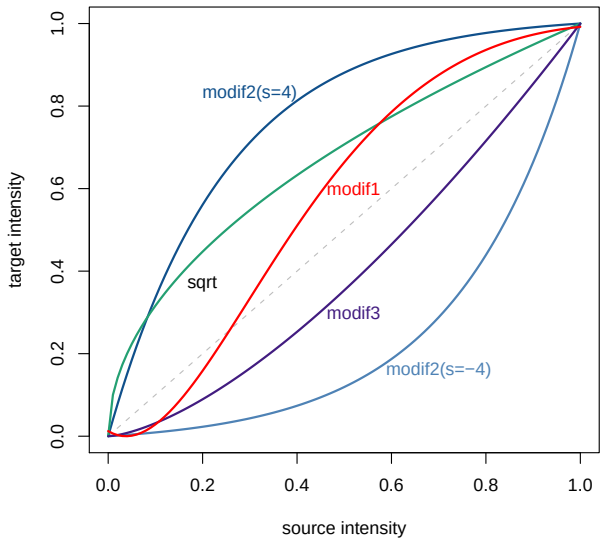

Figure 2: Non-linear transformations for data in the range $[0, 1]$ onto $[0, 1]$. The *curves* filter in Photoshop or The GIMP.

We will use `modif1` to improve the contrast and proceed as follows. The modification used (if any) is problem specific and is determined by the quality of original images:

```
> iGn = modif1(iGn)
> iRn = modif1(iRn)
```

The histograms after the normalization and the above transformation are given in Figure 3:

```
> split.screen(c(2, 2))
> for (i in 1:4) {
+     screen(i)
+     hist(iRn[, , i], xlim = c(0, 1))
+ }
> close.screen(all = TRUE)
```

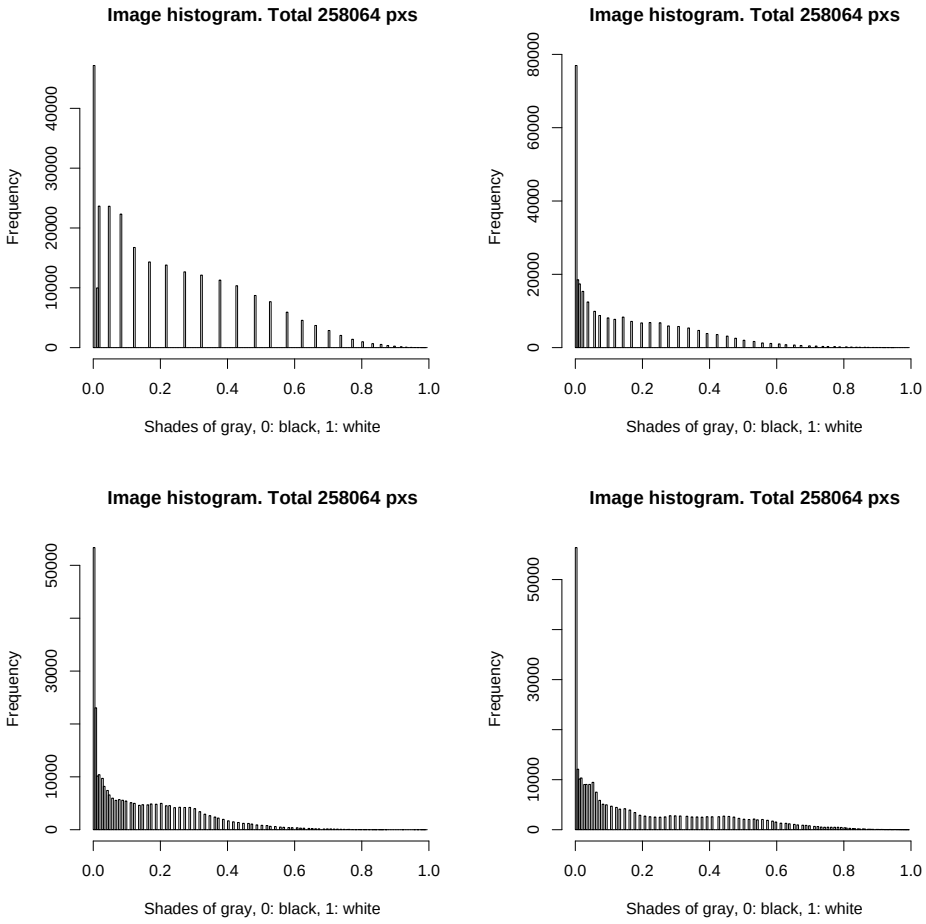

Figure 3: Histograms of individual images in 'Gene1_R.tif' after the normalization.

4 Colour modes

For visual representations, images can be converted between the grayscale and true colour modes; a grayscale image can also be converted into one of the RGB channels if required and channels can be added together as in the following example:

```
> iRG = channel(iRn, "asred") + channel(iGn, "asgreen")
```

```
> display(iRG)
> display(channel(iRG, "gray"))
> display(channel(iRG, "asred"))
```

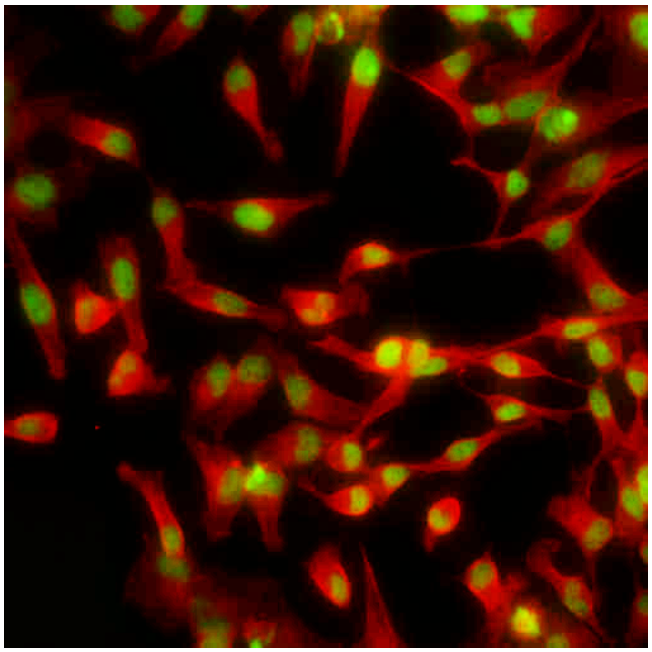

Figure 4: False-colour representation of the data from two independent channels, red and green.

The `channel` function can also be used to convert vectors containing colour data from one format to another. Grayscale to RGB integers:

```
> ch = channel(c(0.2, 0.5, 0.7), "rgb")
> ch
```

```
[1]  3355443  8421504 11776947
```

```
> sprintf("%X", ch)
```

```
[1] "333333" "808080" "B3B3B3"
```

Grayscale to X11 hexadecimal color strings:

```
> channel(c(0.2, 0.5, 0.7), "x11")
```

```
[1] "#333333" "#808080" "#B3B3B3"
```

Color strings to RGB:

```
> channel(c("red", "green", "#0000FF"), "rgb")
```

```
[1]      255    32768 16711680
```

RGB to grayscale:

```
> channel(as.integer(c(3355443, 8355711, 11711154)), "gray")

[1] 0.2000000 0.4980392 0.6980392
```

Images can be stored in files as in the following example for `iRG`:

```
> writeImage(iRG, "composite.tif")
> compression(iRG) = "JPEG"
> writeImage(iRG, "composite.jpg", quality = 98)
```

5 Creating images and further data manipulation

Images can be created either using the default constructor `new` for class `Image` or using a wrapper function, `Image`:

```
> a = Image(runif(200 * 100), c(200, 100))
> image(a)
```

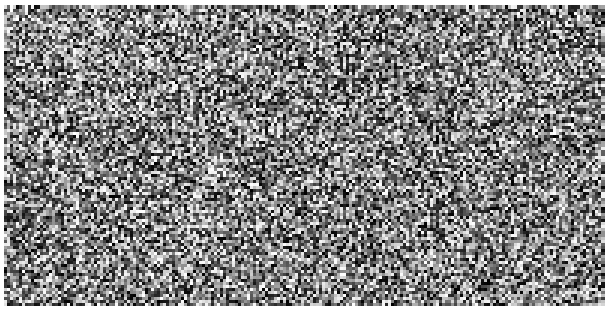

Figure 5: An image of uniform random numbers.

One can also use the data of other images to create new ones:

```
> a = Image(iRn, dim(iRn), colormode = colorMode(iRn))

> display(a)
```

In the above example, it is not necessary to supply the dimensions as those will be automatically detected from the data if that has the `dim` method defined.

Transformations can be performed using `resize`, `rotate` etc. Try the following two to increase the image by a factor of 1.3 or to rotate it by 15 degrees:

```
> a = resize(iRn, dim(iRn)[1] * 1.3)

> display(a)

> a = rotate(iRn, 15)

> display(a)
```

Simple data manipulations can be performed by subsetting. For example, the following lines represent simple thresholding:

```
> a = iRn
> a[a > 0.6] = 1
> a[a <= 0.6] = 0
```

On a grayscale image the values of 0 and 1 in the above example create a black-and-white mask. If now we want to mark the background e.g. in blue and foreground in red, this can be achieved as follows:

```
> b = channel(a, "rgb")
> b[a >= 0.1] = channel("red", "rgb")
> b[a < 0.1] = channel("#114D90", "rgb")

> display(b)
```

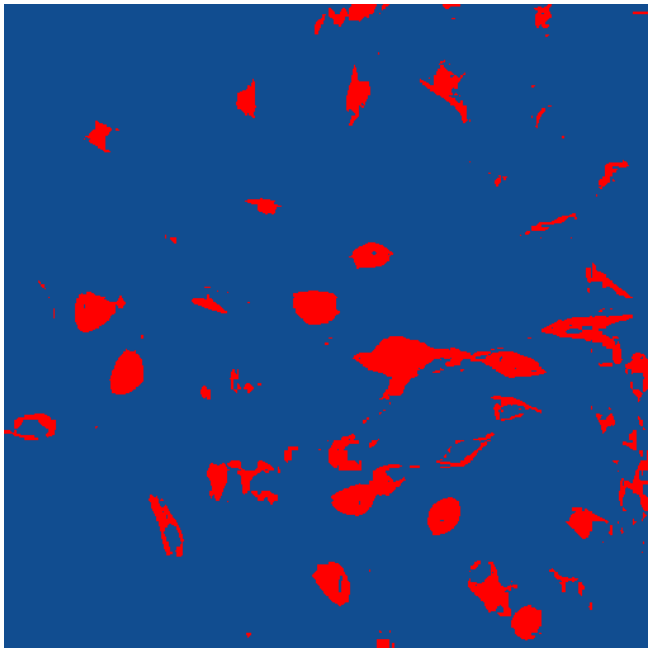

Figure 6: Colour-marked binary mask.

6 Image segmentation and image analysis

The purpose of segmentation is to mask the objects of interest from the background prior to identifying them. The quality of the segmentation will generally define the quality of the subsequent object indexing and feature extraction. We need something better than the mask in Figure 6.

For further object indexing We will make use of the fact that we have two images corresponding to the same location of the microscope – one of the nuclei staining and one for the cytoplasm protein. Assuming that every cell has a nucleus we will use indexed nuclei (after segmentation, indexing and feature extraction) to index cells. Therefore, we start start with segmenting nuclei, images `iG`.

The function `thresh` provides an implementation of an adaptive threshold filter that takes into account inequalities in background intensity across the image. For `iG` the segmented image can be obtained as follows:

```
> mask = thresh(iGn, 15, 15, 0.002)
```

The parameters `w`, `h` of the function are related to the size of the objects we expect to find in the image: objects of different size would require adjustment of these parameters. The `offset` is determined by the local intensity differences. Try using different parameters and compare the segmentation. The quality of segmentation is vital for good quality of object indexing and feature extraction, therefore it worth to spend time tuning the parameters. For comparable results, images across the experiment should be segmented using the same parameter set. This might lead to artifacts in segmentation in some cases, but will ensure that same types of cells look similar in different images! The result of the above segmentation is shown in Figure 7.

Some further smoothing of the mask is necessary. A useful set of instruments for this is provided by *mathematical morphology* implemented in the morphological operators `dilate`, `erode`, `opening` and `closing`:

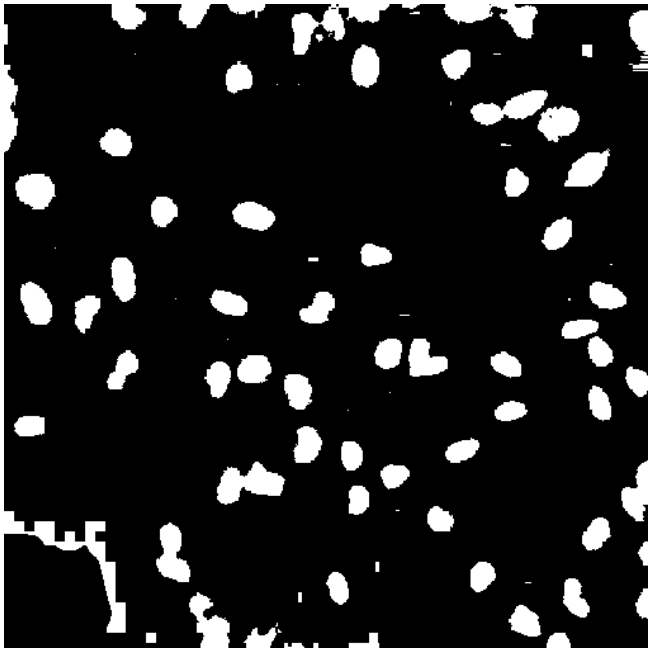

Figure 7: Preliminary nuclei segmentation.

```
> mk3 = morphKern(3)
> mk5 = morphKern(5)
> mask = dilate(erode(closing(mask, mk5), mk3), mk5)
```

Here, several operators were used sequentially. You can observe the results of each of these operators separately by looking at the intermediate images. You can also try different kernels, i. e. different parameters for the function `morphKern`. The current result is shown in Figure 8.

As the next step, one needs to index regions in the segmented image that correspond to the different objects. A classic algorithm for this is computing the distance map transform followed by the `watershed` transform (see Figure 9):

```
> sG = watershed(distmap(mask), 1.5, 1)
```

Finally, when we are happy with the result of the watershed transform, we can remove nuclei that are either too small or too dark or fall on the edge of the images, etc (see Figure 10):

```
> ft = hullFeatures(sG)
> mf = moments(sG, iGn)
> for (i in seq_along(ft)) ft[[i]] = cbind(ft[[i]], mf[[i]])
> sG = rmObjects(sG, lapply(ft, function(x) which(x[, "h.s"] <
+     150 | x[, "h.s"] > 10000 | x[, "int"] < 30 | 0.4 * x[, "h.p"] <
+     x[, "h.edge"])))
```

here `h.s` and `h.p` stand for the hull size and perimeter, `h.edge` for the number of pixels at the image edge and `int` for the intensity of the region (as returned by `hullFeatures` and `moments`). Investigate the structure of `ft` and `mf` and explain what kind of objects were removed.

What we have finally obtained is an `IndexedImage` for the nuclei, where each nucleus is given an index from 1 to `max(sG)`. One can now directly use functions like `getFeatures` or `moments` etc. to obtain numerical descriptors of each nucleus.

In principle, the same distance-map/watershed algorithm could be used to segment the cells, however we often find that neighbouring cells are touching and lead to segmentation errors. We can use the already identified nuclei as seed points to detect corresponding cells – assuming that each cell has exactly one nucleus. This method falls short of detecting multi-nuclear cells, but it improves the quality of detection for all other cells tremendously. We start similarly to the nuclei segmentation, however instead of using `watershed`, we use

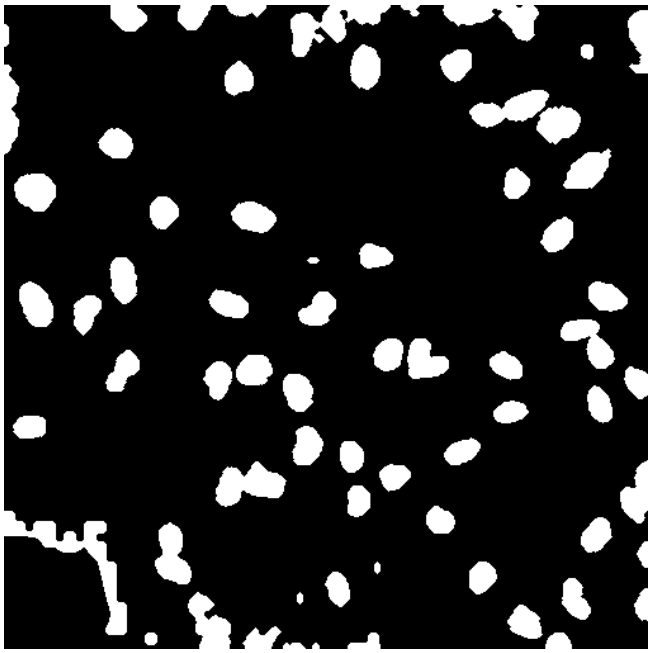

Figure 8: Nuclei segmentation after smoothing and noise removal.

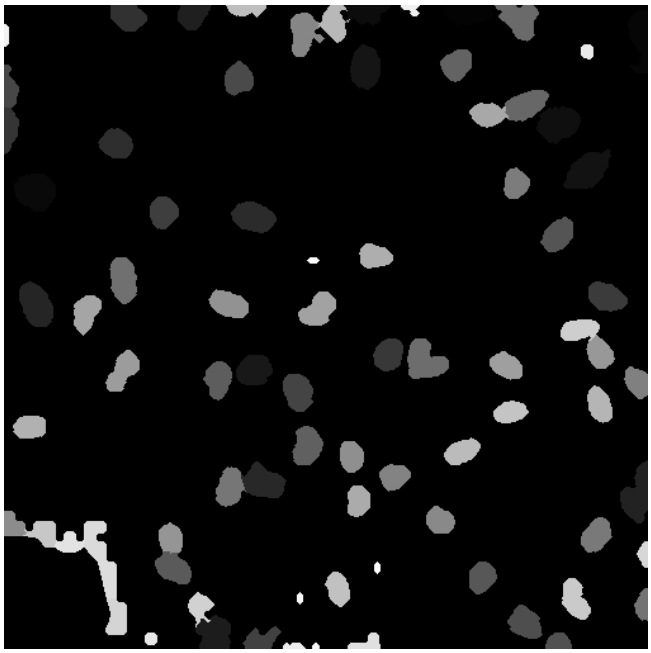

Figure 9: Nuclei segmentation by watershed (before artifact removal).

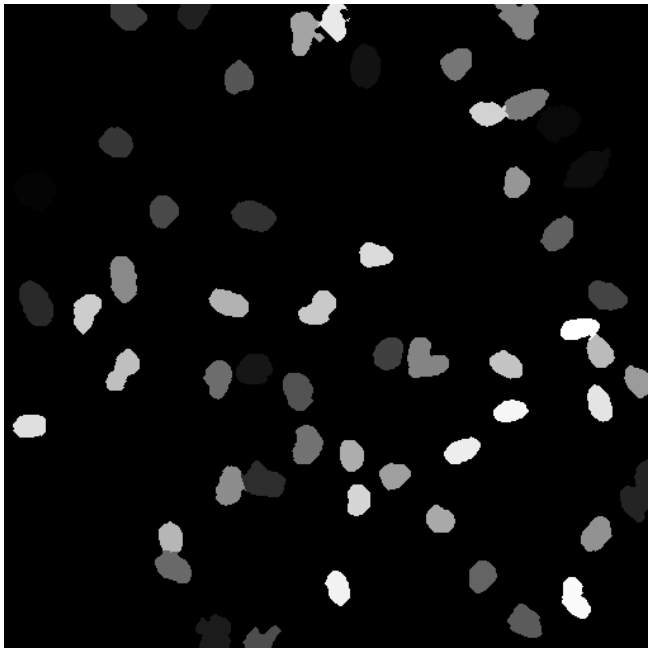

Figure 10: Nuclei segmentation by watershed (after artefact removal).

`propagate`, supplying it with an `IndexedImage` of the seed points (nuclei). The function implements an elegant algorithm that produces a Voronoi segmentation using a metric that combines Euclidean distance and intensity differences between different pixels in the image:

```
> mask = thresh(blur(iRn, 4, 1.5), 25, 25, 0.005)
> mask = erode(erode(dilate(mask, mk5), mk5), mk3)
> sR = propagate(iRn, sG, mask, 1e-05, 1.6)
```

A weighting factor is used in propagate to either give more weight to the Euclidean distance or otherwise to the intensity-driven one. We use a very low value of $1e-5$ basically minimizing the effect of the Euclidean. Also please note that we used the `blur` filter to obtain the original mask. In case of cells we use seed points and we know where the cells are, therefore the mask we use is larger and smoother to accommodate more tiny settled changed in the overall image of every individual cell. The result is shown in Figure 11.

Again, some artifacts need to be removed. In all consequent computations, the number of objects in an indexed image (as obtained from `watershed` or `propagate`) is determined by finding the maximum value. Consider that this value is N for `sR`. If the image contains pixels indexed with N, but is missing pixels with some other indexes, smaller than N, the corresponding objects will be identified with 0 data, first of all 0 size. N can be smaller than the original number of nuclei as it could happen that for some nuclei no cells were identified. There can be many reasons for this: cells masked out or too small or too dark etc. In order to preserve the 1-t-1 match of nuclei to cells, `max(sG)` must be equal N, so we mask out all nuclei with indexes larger than N:

```
> for (i in 1:dim(sR)[3]) {
+     x = sG[, , i]
+     x[x > max(sR[, , i])] = 0
+     sG[, , i] = x
+ }
```

Now as we ensured the 1-to-1 match of nuclei to cells, we can remove cells that are too small or too large to be plausible, are on the edge of the image, are too dark, etc. We also remove the corresponding nuclei (the result is given in Figure 12):

```
> ft = hullFeatures(sR)
> mf = moments(sR, iRn)
> for (i in seq_along(ft)) ft[[i]] = cbind(ft[[i]], mf[[i]])
```

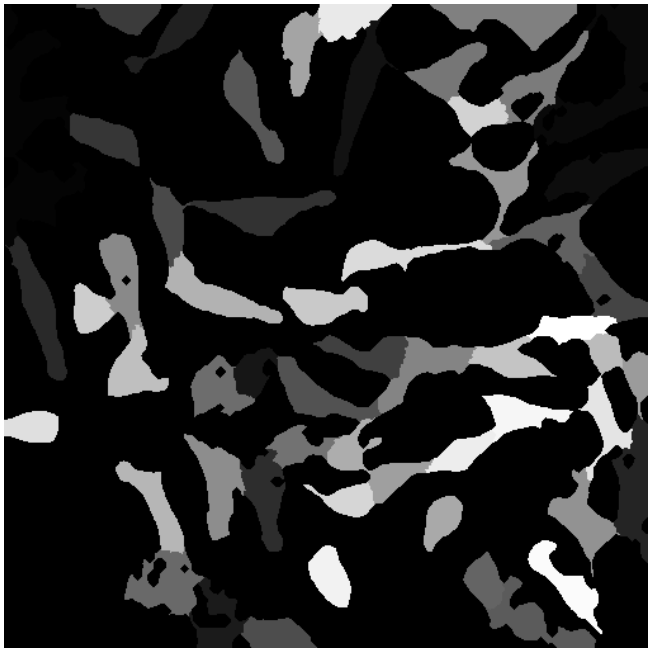

Figure 11: Cell segmentation by the `propagate` function (before artifact removal).

```
> index = lapply(ft, function(x) which(x[, "h.s"] < 150 | x[, "h.s"] >
+     15000 | x[, "int"]/x[, "h.s"] < 0.1 | 0.3 * x[, "h.p"] <
+     x[, "h.edge"]))
> sR = rmObjects(sR, index)
> sG = rmObjects(sG, index)
```

See above for the notations of the column names in `x`.

Finally, having the indexed images for cells and nuclei, the full set of descriptors can be extracted using the `getFeatures` function:

```
> sG = getFeatures(sG, iGn)
> sR = getFeatures(sR, iRn)
> nucl = do.call("rbind", features(sG))
> cells = do.call("rbind", features(sR))
> stopifnot(identical(dim(nucl), dim(cells)))
```

The resulting matrices have 209 rows (one for each of cell/nucleus) and 96 columns (one for each object descriptor).

You can now try out the following visualisations, with the first one shown in Figure 13:

```
> rgb = paintObjects(sR, iRG)
> rgb = paintObjects(sG, rgb)
> ct = tile(combine(stackObjects(sR, iRn)))
> nt = tile(combine(stackObjects(sG, iGn)))
```

The result is shown in Figure 13.

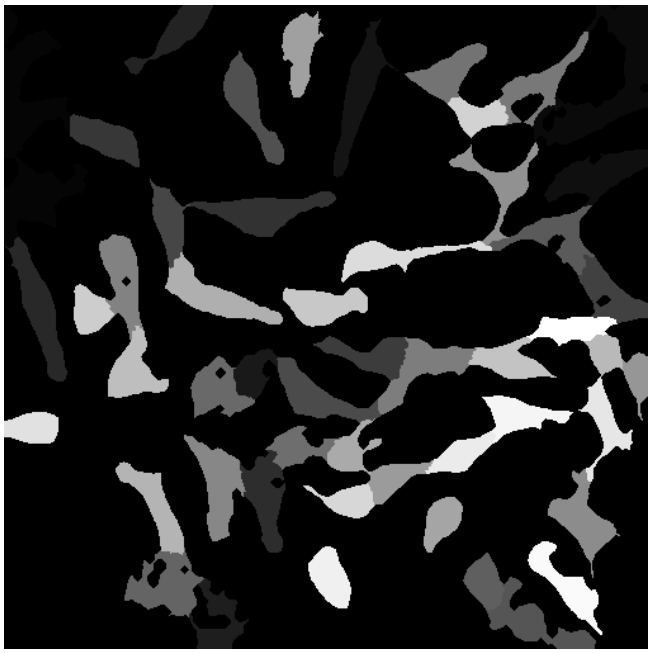

Figure 12: Cell segmentation by propagate (after artefact removal).

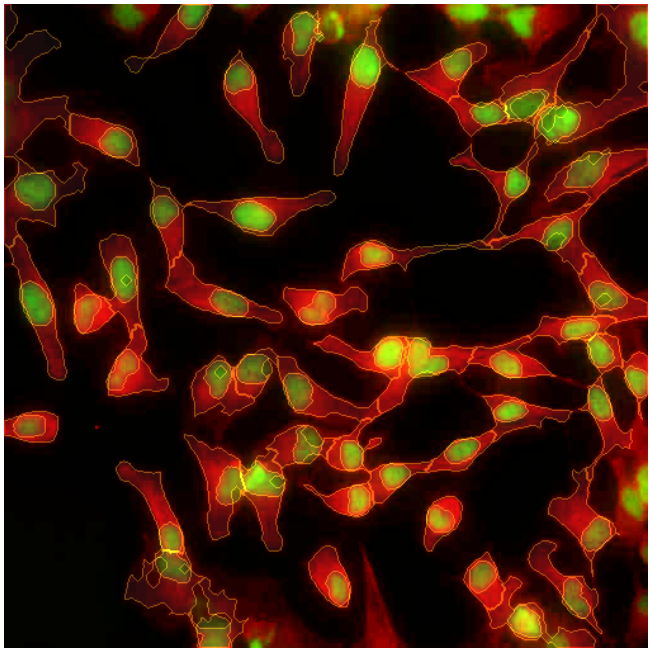

Figure 13: Results of detection.