Analysis of Biological Images with EBImage
BioC-2007 Practicals

Oleg Sklyar
osklyar@ebi.ac.uk

August 6, 2007

In this lab 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 lab can be downloaded from http://www.ebi.ac.uk/~osklyar/BioC2007/data

These images are raw data coming directly from the automated screening microscope[1]. 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). The examples in this lab assume that the image files are in the 'data' sub-directory of the working directory:

```
> files = dir(path = "data", pattern = "tif")
> print(files)
```

```
 [1] "Gene1_G.tif" "Gene1_R.tif" "Gene2_G.tif" "Gene2_R.tif" "Gene3_G.tif"
 [6] "Gene3_R.tif" "Gene4_G.tif" "Gene4_R.tif" "Gene5_G.tif" "Gene5_R.tif"
[11] "Gene6_G.tif" "Gene6_R.tif"
```

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

1 Handling images

EBImage provides two functions to read images, `read.image` and `choose.image`. 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 `choose.image` function, which is available if the package was compiled with GTK+ support:

```
> x = choose.image(TrueColor)
```

The `read.image` 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:

```
> fG = paste("data", dir(path = "data", pattern = "_G.tif"), sep = "/")
> iG = read.image(fG[1], Grayscale)
> print(dim(iG))
```

```
[1] 508 508   4
```

```
> fR = paste("data", dir(path = "data", pattern = "_R.tif"), sep = "/")
> iR = read.image(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 = read.image(paste(baseurl, c("Gene1_R.tif", "Gene2_R.tif"),
+     sep = ""))
```

[1] I have slightly preprocessed for this lab.

2 Exploring data

Image objects 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 look like this:

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

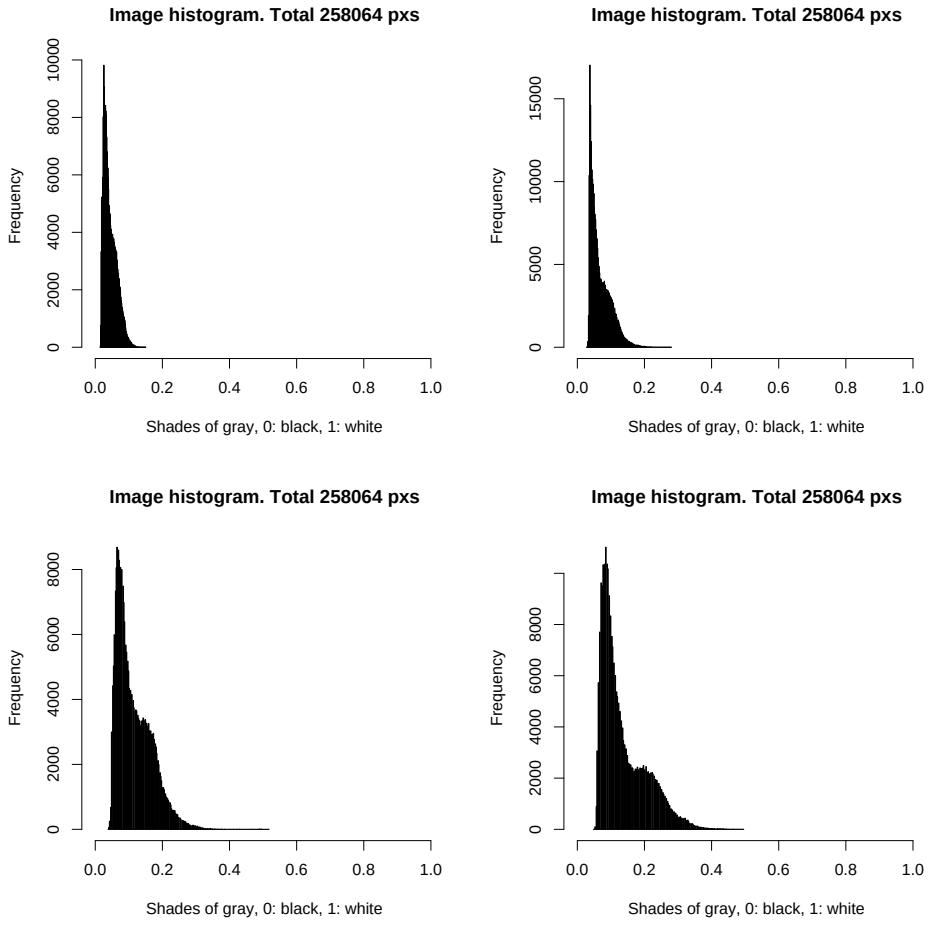

Figure 1: Histograms of individual images in 'Gene1_R.tif'. The frames have significantly different dynamic ranges and need to be normalized before intensity-based descriptors can be used.

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 the images each 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 0.02656858 0.04845574 0.0703125
[2,] 0.2718872 0.53049003 1.00000000 0.9562561

> 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)

> display(iRn)
```

EBImage provides a set of functions to manipulate images like `enhance`, `contrast`, `despeckle`, `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. Let us define some functions.

```
> 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, for $x \in [0, 1]$ and different values of s, are shown on page 4.

Try applying the above functions to `iRn` and `iGn` and observe the differences. We will proceed with the following.

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

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.

After the normalization and the above transformation, our histograms look like the following:

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

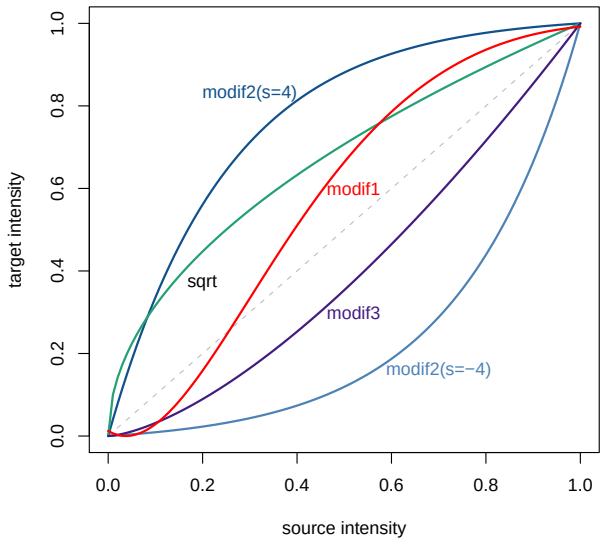

Figure 2: Non-linear transformations.

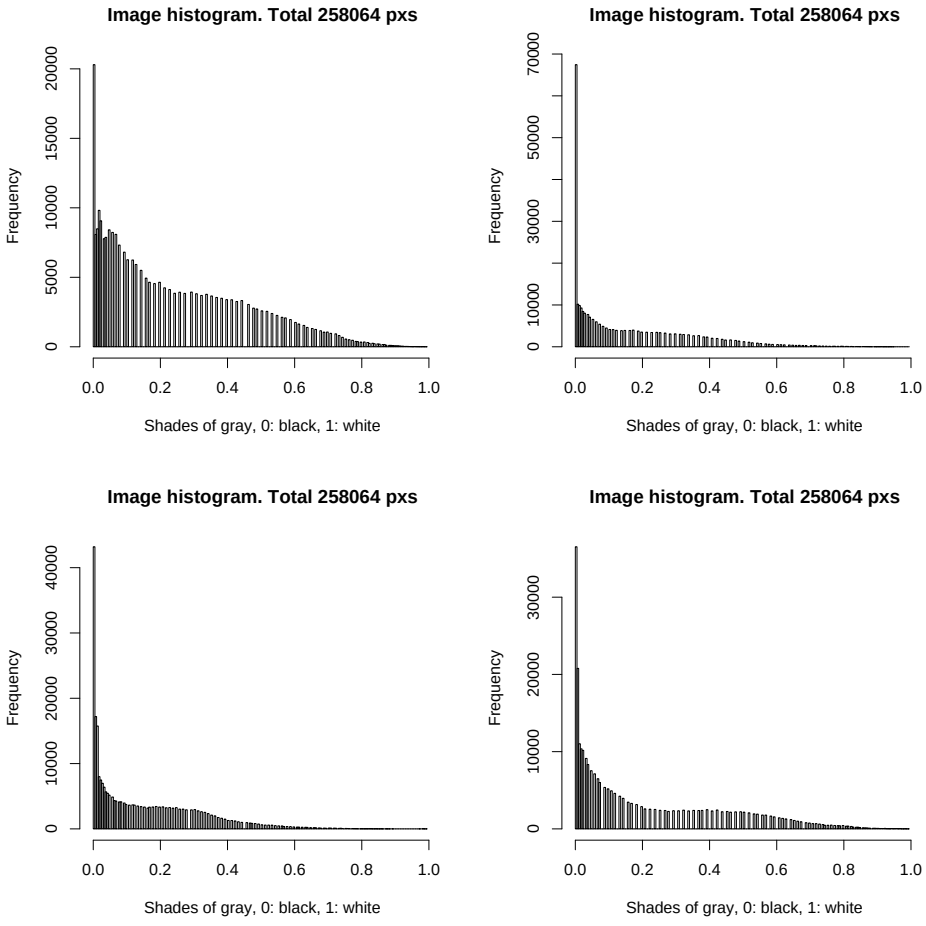

Figure 3: Histograms of individual images in 'Gene1_R.tif' after normalization.

4 Colour modes

For visual representations, images can be converted between grayscale and true colour. A grayscale image can be converted into one of the RGB channels:

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

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

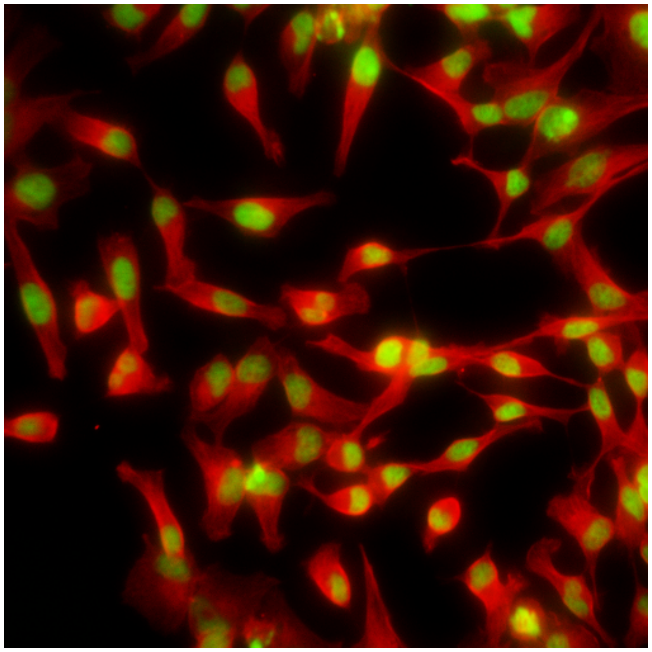

Figure 4: False colour representation of the image data.

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  8355711 11711154
```

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

```
[1] "333333" "7F7F7F" "B2B2B2"
```

Grayscale to X11 hexadecimal color strings:

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

```
[1] "#333333" "#7F7F7F" "#B2B2B2"
```

Color strings to RGB:

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

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

RGB to grayscale:

```
> channel(c(3355443L, 8355711L, 11711154L), "gray")
```

```
[1] 0.2000000 0.4980392 0.6980392
```

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

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

5 Creating images and further data manipulation

Images can be created either using the default constructor `new` with objects of class `Image` or using the wrapper function `Image`:

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

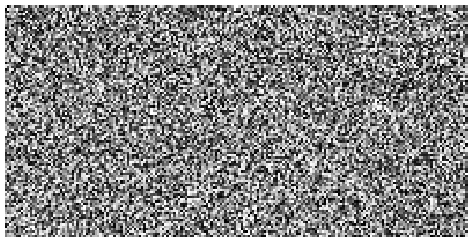

Figure 5: An image of uniform random numbers.

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

```
> a = Image(iRn, dim(iRn))
> display(a)
```

Transformations can be performed using `resize`, `rotate` etc. Try the following two:

```
> display(resize(iRn[, , 1], dim(iRn)[1] * 1.3))
> display(rotate(iRn[, , 1], 15))
```

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

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

If now instead of using black and white we want to mark the background e.g. as blue and foreground as 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)
```

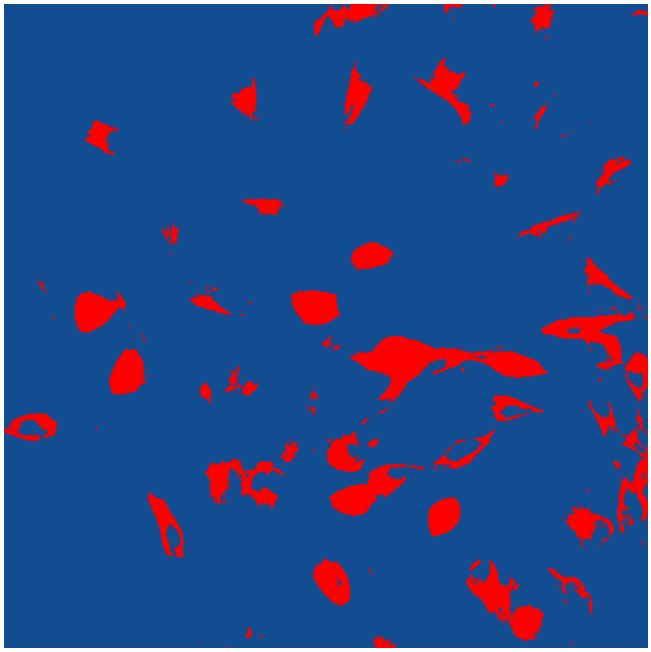

Figure 6: Marked thresholded images.

6 Image segmentation and image analysis

The purpose of segmentation is to identify the objects of interest in an image. The quality of the segmentation will generally define the quality of the subsequent feature extraction. We need something better than 6.

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 of the cytoplasm protein. Because of the nature of these images it is reasonable to start with segmenting nuclei because they are more clearly separated.

The function `thresh` provides an implementation of adaptive thresholding that takes into account inequalities in background intensity across the image. For the nuclei (the green channel, G), the segmented image can be obtained as follows:

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

The parameters `w`, `h` and `offset` of the function correspond to the size of objects we expect to find in the image: objects of different size would require adjustment of these parameters. You can try the results from using different parameters.

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

```
> 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`.

As the next step, one needs to index regions in the segmented image that correspond to the different objects. A classic algorithm for segmentation is the computation of the distance map followed by the `watershed` transform.

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

The result is shown in Figure 9.

Finally, when we are happy with the results 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.:

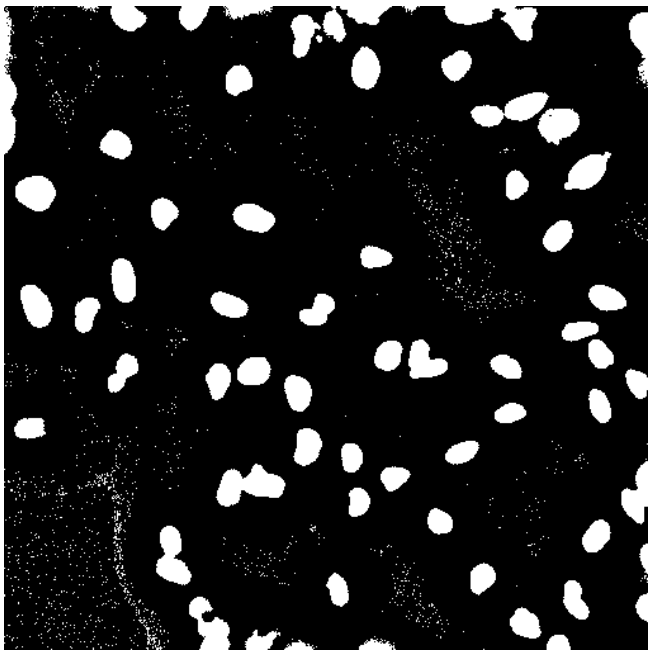

Figure 7: Preliminary nuclei segmentation.

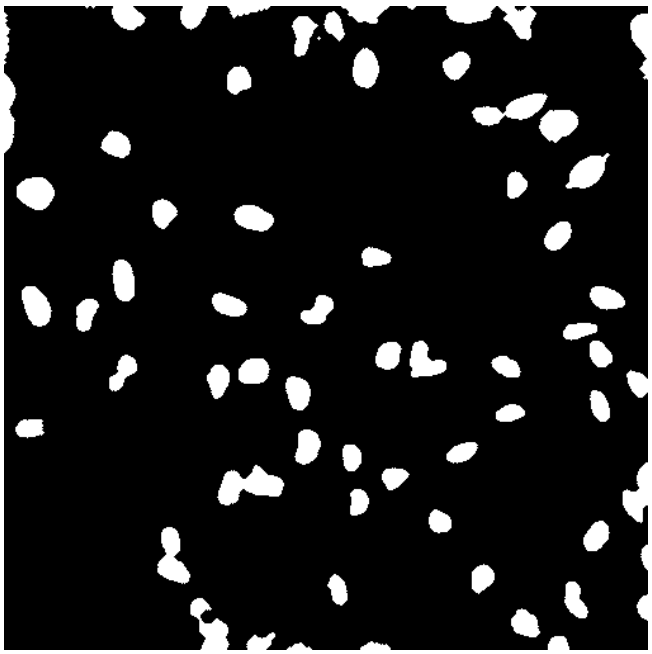

Figure 8: Nuclei segmentation after noise removal.

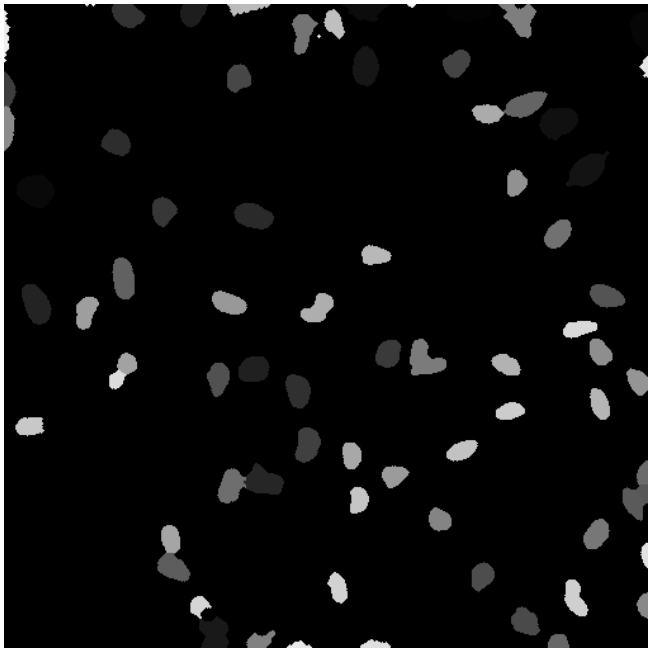

Figure 9: Nuclei segmentation by watershed (before artefact removal).

```
> ft = hull.features(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[, 3] < 150 |
+     x[, 3] > 10000 | x[, 18] < 30 | 0.4 * x[, 4] < x[, 10])))
```

The result is shown in Figure 10.

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 algorithm as above (distance map – watershed transform) could be used to segment the cells, however we often find that neighbouring cells are touching and lead to segmentation errors. But we can use the already found nuclei as seed points to detect their cells – assuming that each cell has exactly one nucleus. This method falls short of detecting multinuclear 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 the function `propagate`, supplying it with the indexed image of 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)
```

The result is shown in Figure 11.

Again, some artefacts need to be removed. If we want to keep a 1-to-1 match between the cells and the nuclei we also need to take into account the situation when for some nuclei no cell was detected (the cell can be dead, or nucleus was a false positive). This is important for those nuclei at the end of the list (for those in the middle of the list, the corresponding cell will have all descriptors at 0):

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

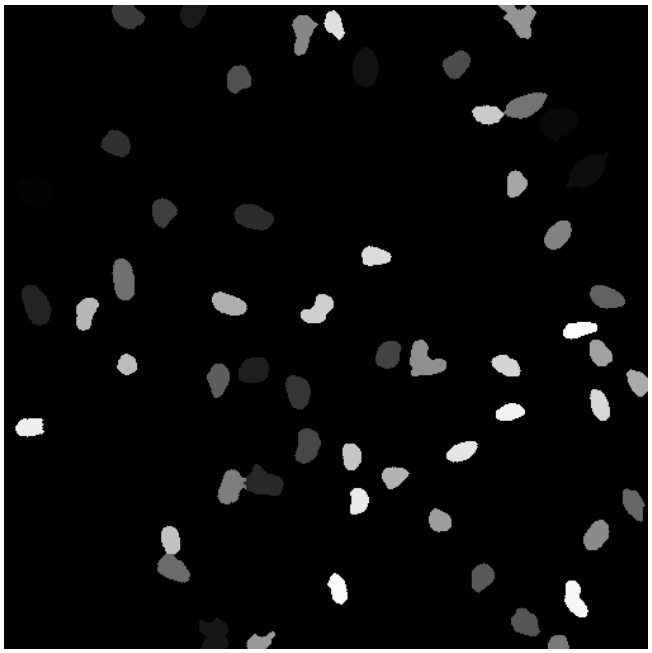

Figure 10: Nuclei segmentation by watershed (after artefact removal).

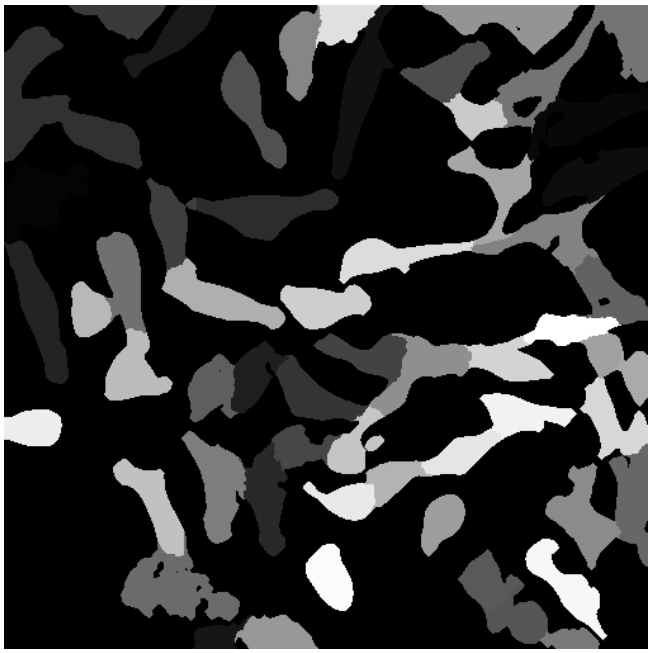

Figure 11: Cell segmentation by the `propagate` function (before artefact removal).

Now as we have made that there is a 1-to-1 match, 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 their corresponding nuclei.

```
> ft = hull.features(sR)
> mf = moments(sR, iRn)
> for (i in seq_along(ft)) ft[[i]] = cbind(ft[[i]], mf[[i]])
> index = lapply(ft, function(x) which(x[, 3] < 150 | x[, 3] >
+     15000 | x[, 18]/x[, 3] < 0.1 | 0.3 * x[, 4] < x[, 10]))
> sR = rmObjects(sR, index)
> sG = rmObjects(sG, index)
```

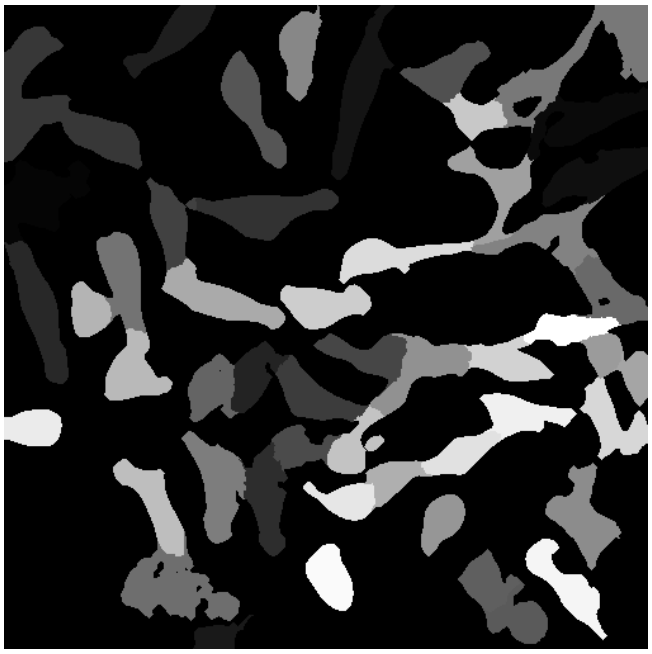

Figure 12: Cell segmentation by propagate (after artefact removal).

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))
> dim(nucl)

[1] 213  96

> cells = do.call("rbind", features(sR))
> dim(cells)

[1] 213  96
```

The resulting matrices have 213 rows (one for each of cell/nucleus) and 96 columns (one for each object descriptor).

You can now try to apply the above workflow to the other genes' data in the example dataset, or you can try out the following visualisation.

```
> 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.

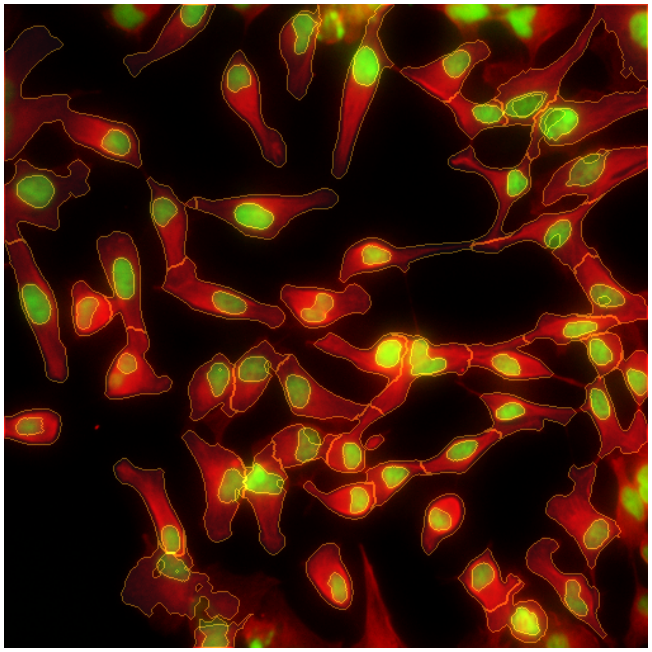

Figure 13: Results of detection.