



Introduction to **EBImage**, an image processing and analysis toolkit for R

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

May 12, 2008

Contents

1	Introduction	1
2	Importing and exporting images	1
3	Displaying images and accessing the image data	3
4	Image processing	3
5	Colour modes	6
6	Creating images and further data manipulation	7
7	Image segmentation and image analysis	7

1 Introduction

In this manual we demonstrate the **EBImage** package. In creating this package, the application that we have in mind is the analysis of sets of images acquired in cell-based RNAi or compound screens with automated microscopy readout, and the extraction of numerical descriptors of the cells in these images. Relations between the descriptor values and biological phenotypes are sought. Some of the choices we have made in the design of the package reflect this application. This vignette tries to give a general overview of the package functionality, it does not provide a recipe for the analysis of cell-based RNAi screens.

The package makes use of two external libraries for some of its functionality, **ImageMagick** and **GTK+**. In particular, **ImageMagick** is used for image import and export, and we provide an interface to many of its image processing capabilities. **GTK+** provides facilities for the display of and user interaction with images. Reliance on these external (to R and the **EBImage** package) libraries makes the installation of **EBImage** a bit more tricky than that of a self-contained package. Please see the vignette **EBImage installation HOWTO** for hints on the installation.

2 Importing and exporting images

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

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. The `chooseImage` function provides a dialogue window that lets you interactively choose an image from the file system. It is available if the package was compiled with GTK+ support (this is always the case on Windows):

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

For programmatic use, the `readImage` function can read local files, or files on the internet via the 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. Images and image stacks are stored in objects of class `Image`:

```
> imgdir = file.path(system.file(package="EBImage"), "images")
> fG = dir(imgdir, pattern="_G.tif", full.names=TRUE)
> iG = readImage(fG[1], Grayscale)
> class(iG)
```

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

```
> dim(iG)
```

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

```
> fR = dir(imgdir, pattern="_R.tif", full.names=TRUE)
> iR = readImage(fR[1])
```

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

Images can be read from remote URLs 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="/"))
```

The `writeImage` function can be used to write images to files.

```
> outdir = tempdir()
> writeImage(iR, file.path(outdir, "test.png"))
> compression(iR)
```

```
[1] "JPEG"
```

```
> writeImage(iR[, ,1], file.path(outdir, "test.jpg"), quality=90)
> file.info(dir(outdir, full.names=TRUE))[ ,c(1,5)]
```

	size	ctime
E:\\biocbld\\bbs-2.2-bioc\\tmpdir\\RtmpiaGx3W\\Rf2d1239b3	0	2008-05-12 03:40:39
E:\\biocbld\\bbs-2.2-bioc\\tmpdir\\RtmpiaGx3W\\test-0.png	51665	2008-05-12 03:40:35
E:\\biocbld\\bbs-2.2-bioc\\tmpdir\\RtmpiaGx3W\\test-1.png	65662	2008-05-12 03:40:35
E:\\biocbld\\bbs-2.2-bioc\\tmpdir\\RtmpiaGx3W\\test-2.png	98917	2008-05-12 03:40:37
E:\\biocbld\\bbs-2.2-bioc\\tmpdir\\RtmpiaGx3W\\test-3.png	105709	2008-05-12 03:40:37
E:\\biocbld\\bbs-2.2-bioc\\tmpdir\\RtmpiaGx3W\\test.jpg	7341	2008-05-12 03:40:39

3 Displaying images and accessing the image data

The preferred method for displaying objects of class *Image* is the `display` function.

```
> display(iR)
> animate(iR)
```

`display` and `animate` create their own windows, using GTK+ or ImageMagick functions. Sometimes it may also be useful to use the `image` method for the *Image* class. This method plots the image on the current R device.

```
> image(iR[, , 1])
```

One can access the data in the same way as for a 3-dimensional array, where the first two dimensions correspond to x - and y -directions and applications can use the third dimension for the z -direction, for different channels, for a time covariate, or indeed for any other experimental covariate. Alternatively, the size of the third dimension can be set to 1, which corresponds to a simple 2-dimensional image. The *Image* class contains (inherits from) the base R class *array*, and one can use all the statistical summaries and plotting functions for arrays.

```
> is(iG, "array")
```

```
[1] TRUE
```

For example, for the image stack `iR`, which consists of 4 images, we can draw histogrammes of the intensities in each image (see Figure 1).

```
> par(mfrow=c(2,2))
> for(i in 1:4)
+   hist(iR[, , i], breaks=20, xlim=c(0,1))
```

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

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

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

4 Image processing

One often distinguishes between image processings, operations that map images into transformed images, and image analysis, operations that extract features of interest or compute numerical or categorical summaries.

One of the simplest image processing operations is *normalization* of the image intensities, i.e. adjusting the intensities from different images such that they are on the same range,

$$x \mapsto \frac{x - b}{s} \quad (1)$$

This is provided by the `normalize` function. The default target range is $[0, 1]$, which corresponds to $b = \min\{x\}$, $s = \max\{x\} - \min\{x\}$. The minimum and maximum can be computed either for each image separately, or simultaneously for all images in a stack. Let us try the following two normalizations and compare the results.

```
> iRnns = normalize(iR, separate=FALSE)
> apply(iRnns, 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)
```

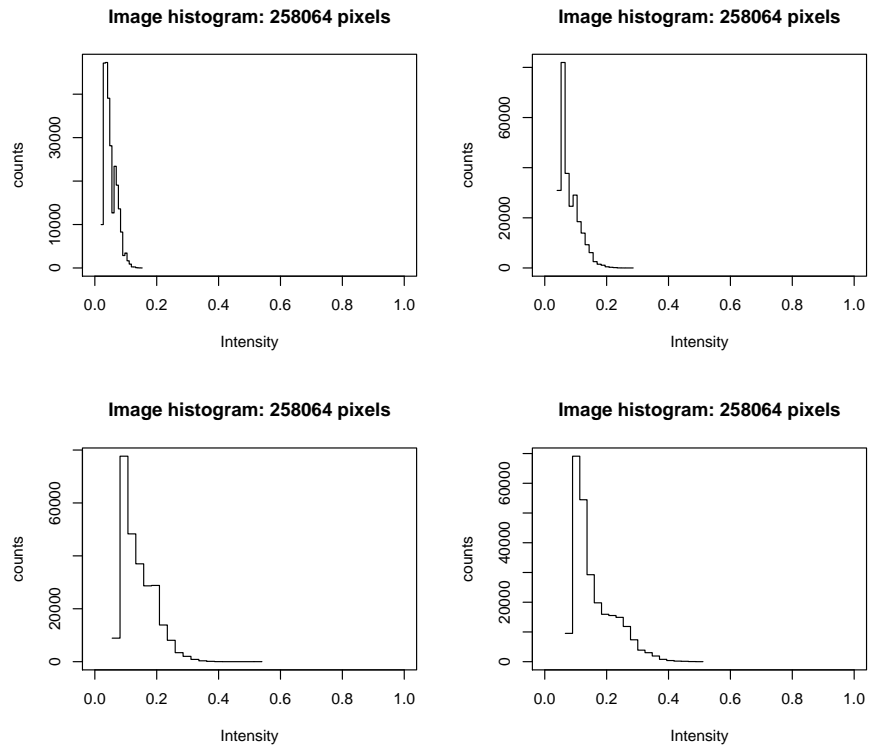


Figure 1: Histogrammes of intensities of the four images in 'Gene1.R.tif'. Note that, in general, histogrammes may be subject to binning artifacts, and smooth density estimates (as for example provided by the `density` function from the `stats` package or the `multidensity` function from the `geneplotter` package) are often more appropriate.

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

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

More generally, one might also normalize not on range but on other distribution properties such as interquantile ranges or other measures of location and scale.

Functions from **ImageMagick** are provided to manipulate images, such as **enhance**, **contrast**, **despeckle** and **denoise**. Try them out on the images in **iRn**. Such manipulations can be useful for visualisation, but their results are likely not appropriate for quantitative analyses.

Sometimes it is useful to apply non-linear transformations. When displaying a grayscale image, this can improve contrast in regions of high interest. For example, a power transformation $x \mapsto x^\alpha$ with $0 < \alpha < 1$ will enhance contrast at lower intensities and suppress contrast at higher intensities. Transformations are sometimes also used in model-based data analyses. For example, when the noise component of a signal has a constant coefficient of variation, rather than a constant standard deviation, then a logarithmic transformation is often convenient. Box-Cox type transformations have many uses in regression analysis. 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^0.5
```

The graphs of these functions are shown in Figure 2.

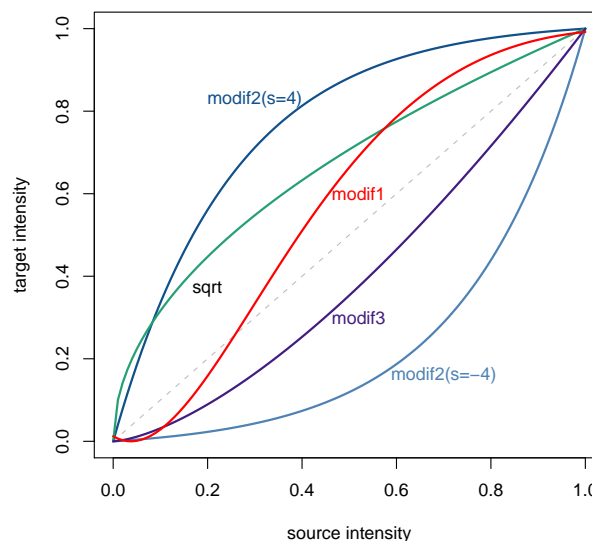


Figure 2: Non-linear transformations of the range $[0, 1]$ onto itself.

Transformations of image size and orientation can be performed using **resize**, **rotate** etc. The following increases the size of the image by a factor of 1.3 or rotates it by 15 degrees:

```
> a1 = resize(iRn, dim(iRn)[1]*1.3)
> a2 = rotate(iRn, 15)

> display(a1)
> display(a2)
```

5 Colour modes

For visual representations, images can be converted between 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"))
```

The image `iRG` is shown in Figure 3. The `channel` function can also be used to convert vectors containing

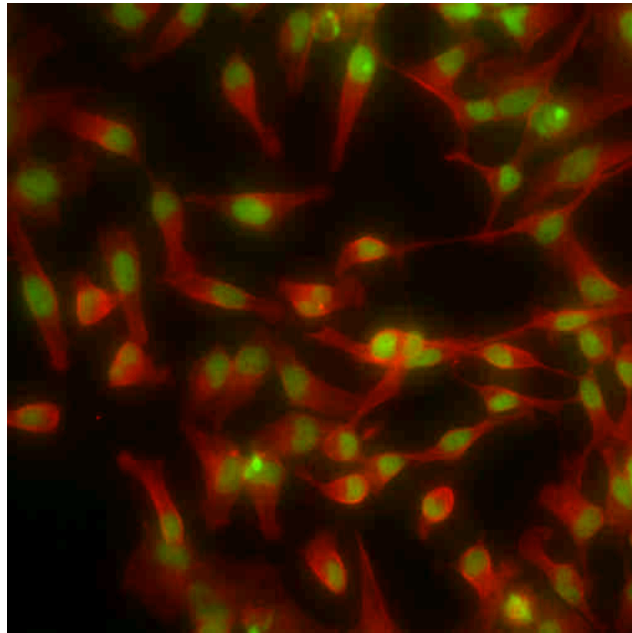


Figure 3: The image `iRG`, a false-colour representation of the data in the two channels `iRG` and `iG`.

colour data from one format to another. Grayscale to RGB integers:

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

```
[1] 3355443 8421504
```

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

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

Grayscale to X11 hexadecimal color strings:

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

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

Color strings to RGB:

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

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

24-bit RGB integers to grayscale:

```
> channel(as.integer(c(0x0000f0, 0x00f000, 0xf0000, 0x808080)), "gray")
```

```
[1] 0.28140688 0.55246815 0.00669871 0.50196078
```

6 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))
```

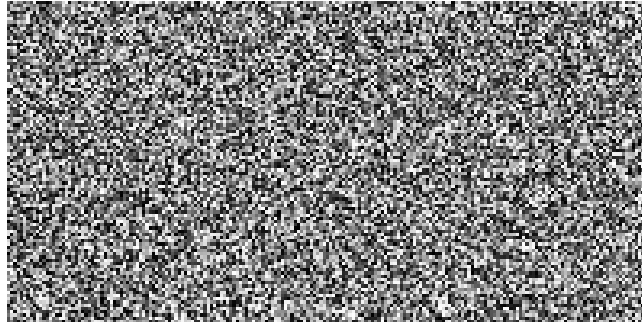


Figure 4: An image of uniform random numbers.

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

```
> a = iRn
> a[ a > 0.6 ] = 1.0
> a[ a <= 0.6 ] = 0.0
```

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)
```

7 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 5.

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 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 is worth spending 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 6.

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`:

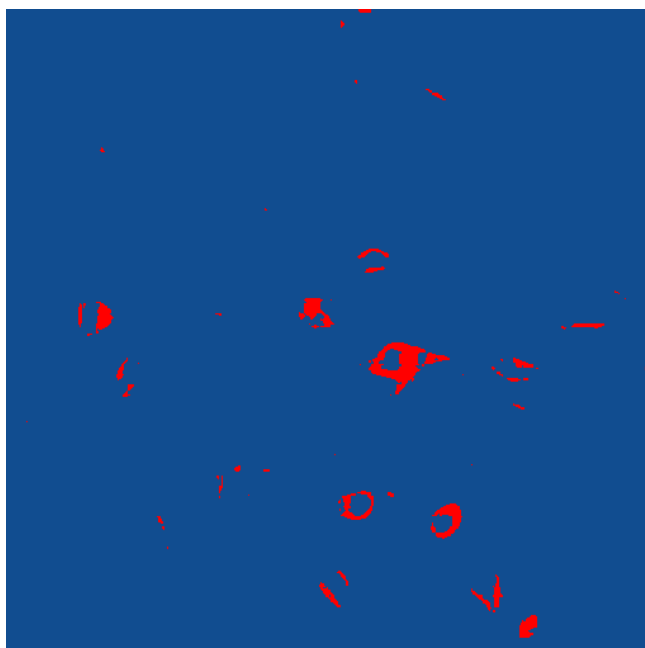


Figure 5: Colour-marked binary mask.

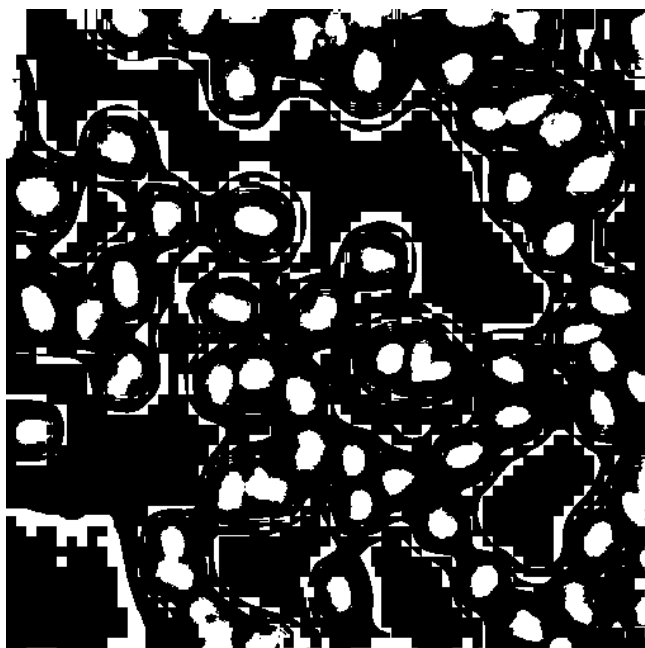


Figure 6: Preliminary nuclei segmentation.

```

> 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 7.

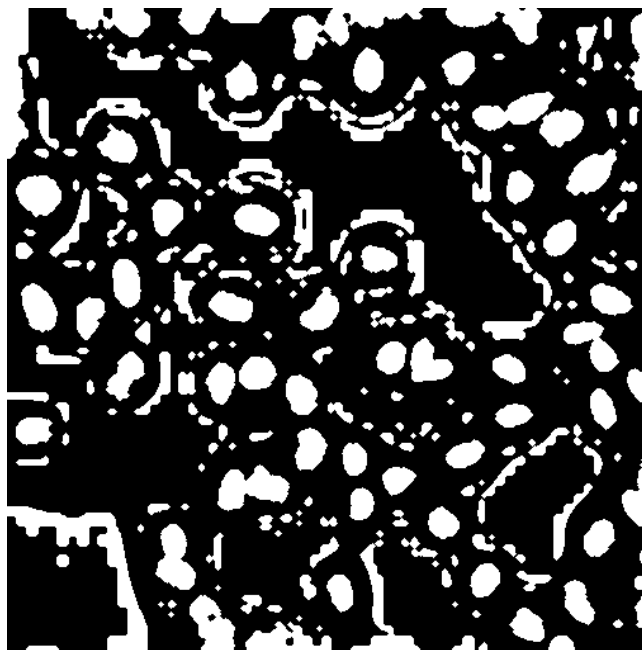


Figure 7: Nuclei segmentation after smoothing and noise removal.

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 8):

```

> 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 9):

```

> ft = hullFeatures(sG)  ## need these for edge and perimeter
> mf = moments(sG, iGn)  ## need these for intensity and size
> for ( i in seq_along(ft) ) ft[[i]] = cbind(ft[[i]], mf[[i]])
> sG = rmObjects(sG,
+               laply(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

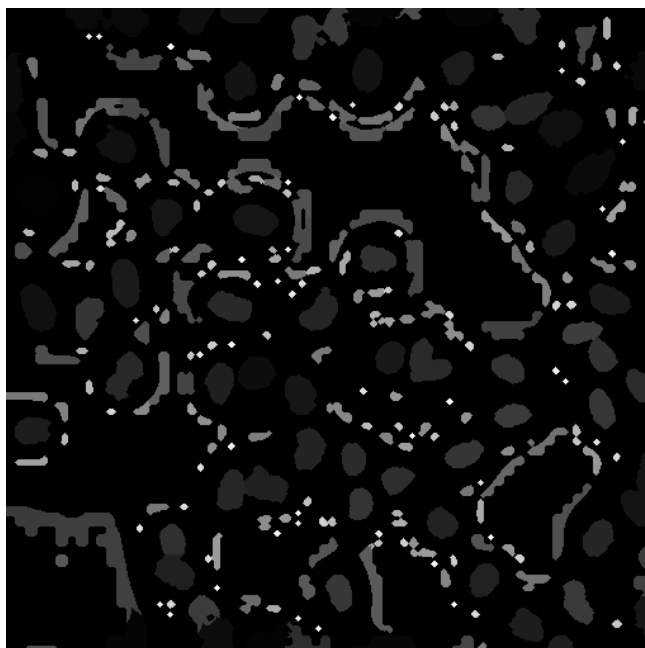


Figure 8: Nuclei segmentation by watershed (before artifact removal).

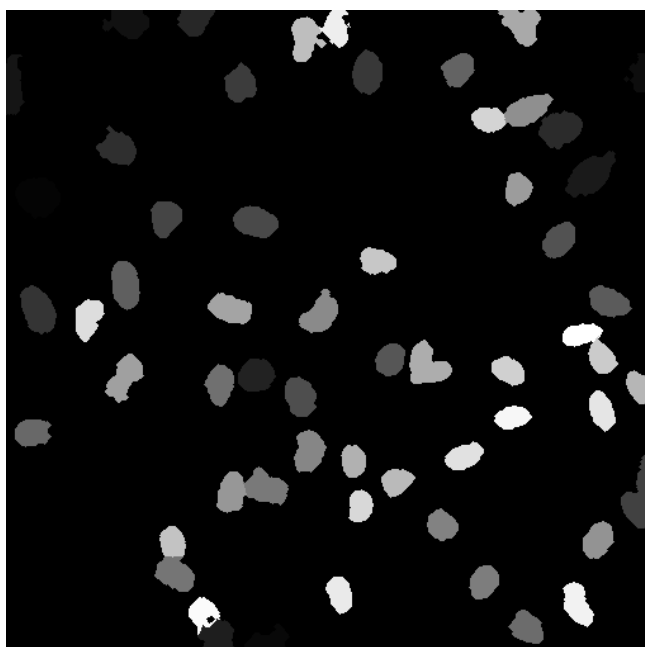


Figure 9: Nuclei segmentation by watershed (after artefact removal).

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 `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-5, 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 10.

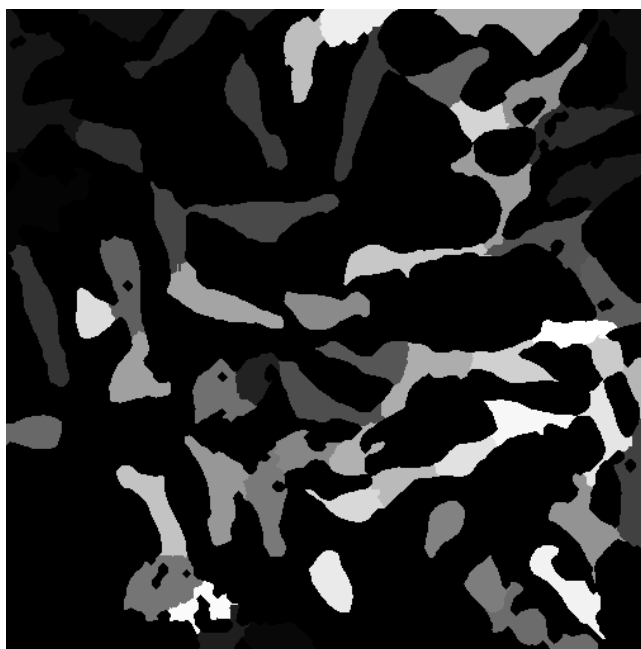


Figure 10: Cell segmentation by the `propagate` function (before artifact removal).

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]) ] = 0.0
+   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 11):

```

> ft = hullFeatures(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[, "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.

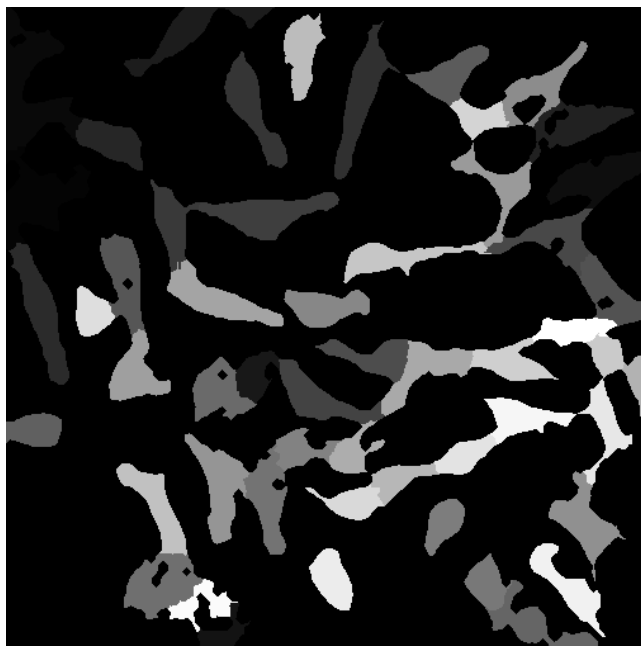


Figure 11: Cell segmentation by propagate (after artefact removal).

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

```

The resulting matrices have 212 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 12:

```

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

```

The result is shown in Figure 12.

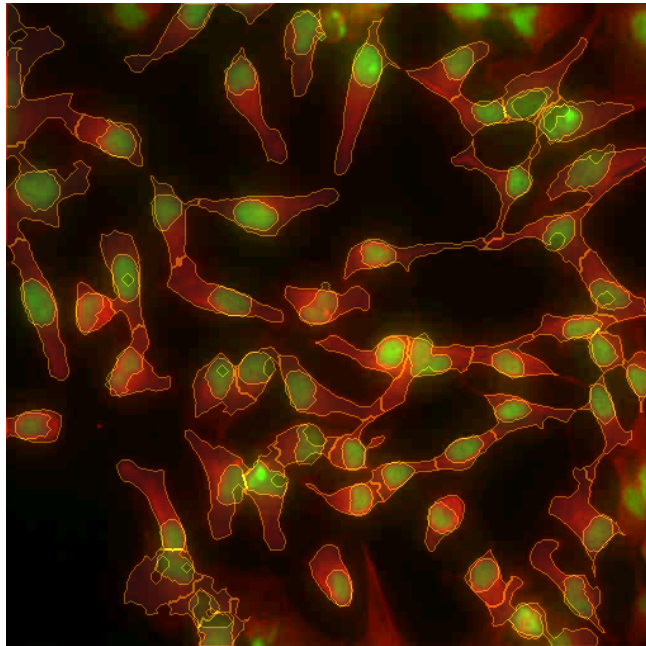


Figure 12: Results of detection.