Analysis of Biological Images with EBImage
BioC-2006 Practicals

Oleg Sklyar
osklyar@ebi.ac.uk

July 27, 2006

1 Introduction

In this lab we will use the EBImage Bioconductor package to preprocess and analyse a subset of images from a large-scale RNAi microscopy screen. The aim of the lab is to learn how to extract simple object descriptors that describe cellular or nuclear phenotypes from the images. The descriptors will be such as size, density, shape factor or intensity of cells and nuclei. These 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.

The package vignette provides an overview of the EBImage functionality for the analysis of cell-based assays. It also contains many example that can assist in completing the exercises in this course. If you want to access the vignette before you install the package, download it from:

www.ebi.ac.uk/~osklyar/projects/EBImage/help/EBImageHOWTO.pdf

2 Software, Data and Manual

2.1 Dependencies

EBImage depends on *Magick++* of the ImageMagick image processing library. Install it before installing the package. To enable TIFF and PNG file support in ImageMagick, and thus in EBImage, 'libtiff' and 'libpng' have to be installed before ImageMagick is compiled.

The ImageMagick library can be compiled and installed from source available from www.imagemagick.org. This also installs *Magick++* headers required to compile EBImage. Alternatively, precompiled binary `rpm` and `deb` packages are available for many Linux distributions. The FINK project (fink.sourceforge.net) contains precompiled binaries of ImageMagick-dev, libtiff, libpng for MacOS (untested). If installing precompiled binaries, please ensure that you include `Magick++-devel` (or similarly named depending on the distribution) and that you also install libtiff and libpng if those are not automatically required by dependency tracking.

MacOS users, do not use ImageMagick that comes with your system, install it new – either from source or from FINK. If you want to keep your system's ImageMagick version, please refer to the 'INSTALL' file included in the EBImage tarball for further instructions.

2.2 Download repositories

The package can be compiled and installed from source on Linux/UNIX and with minor limitations on MacOS. At the moment, there is no MS Windows version available.

This workshop assumes that EBImage version 1.3.102 and later is used. The Bioconductor release 1.8 contains version 1.2.2, which cannot be used in the course.

The source-code tarball can be downloaded from:

www.ebi.ac.uk/huber-srv/data/Rrepos/src/contrib/

The package can be installed or updated either from the source tarball or by issuing:

```
> install.packages("EBImage",
+   repos="http://www.ebi.ac.uk/huber-srv/data/Rrepos")
```

After a successful installation, use the `library` function to load the package:

```
> library(EBImage)
```

2.3 Data files

Images used in this lab are available for download in as a single gzipped file:

www.ebi.ac.uk/~osklyar/projects/EBImage/examples/BioC2006images.tar.gz

Please download and unpack this file in an empty directory, which will be used as your working directory for this lab. You do this with the following shell command

```
tar -zxvf BioC2006images.tar.gz
```

After unpacking, a subdirectory named 'images' will be created containing 33 images. These are raw data – images saved directly by the microscope control software. The images are 16-bit grayscale TIFF's, in which only 12 bits are used for data because of the CCD camera specs. Therefore, when displayed they all look black because the data range (0...4095) is only a fraction of the 16-bit range (0...65535). The 33 images are 11 randomly selected probes measured with 3 different wavelengths (channels) per probe. Different channels are marked with suffixes `w1`, `w2` and `w3` in the file names, and they constitute the nuclear channel, tubulin distribution and cellular membranes, respectively. An additional false-colour RGB image, combining data of different wavelengths in different colour channels, is available on the network to practice reading files over the network. A copy of this file is included in the 'images' subfolder in case the network is not available.

If you are unable to install libtiff to support TIFF files, there is an analogous set of JPG files, already normalized, that can be obtained here:

www.ebi.ac.uk/~osklyar/projects/EBImage/examples/BioC2006imagesJPG.tar.gz

2.4 Limitations and known bugs

`display` can only show one window at a time; close the display window before you call `display` again otherwise an error message is generated.

The function `display` will not work on MacOS and via `ssh`. As a temporary workaround, please use `plot.image`. This function supports **only** grayscale images, therefore one can use the `plot.image(toGray(myRGBimage))` to display grayscale previews of RGF images. Furthermore, it can only display signle 2D images and does not provide image browsing like `display`, use therefore the following synax for image stacks (multiple 2D images, 3D images): `plot.image(toGray(myRGBimage[■i]))` or `plot.image(myGrayImage[■i])`.

3 Handling images

In this section we will exercise reading images, creating new images and storing them in files.

The EBImage function `read.image` can read local or network files. Multiple files can be loaded into a single variable producing an image stack (a 3D `matrix`). The size of images in a stack is determined by the size of the first read: other images, if bigger – are cropped, if smaller – a black background is generated.

Hereafter, to get help on the package functions use `help.start()` to start the help browser and browse to the package help files, or call `help(EBImage)` before or after issuing `help.start()`.

Exercise 1
Read the supplied images into three separate objects depending on the channel, `w1`*,* `w2` *and* `w3`*. Investigate the class and the structure of objects created using* `class`*,* `slotNames`*,* `str` *and other* R *functions. Try loading images for one channel as RGB and then converting them to grayscales. Ensure that final images are grayscales or converted to grayscales after loading: we will need grayscale images for further analysis.* **Tip:** *use* `dir` *supplying a matching pattern to select files for every channel.*

```
> f1 <- paste("images", dir("images", pattern = "w1"), sep = "/")
> w1 <- read.image(f1)
> class(w1)

[1] "Image"

> slotNames(w1)

[1] ".Data" "rgb"

> f2 <- paste("images", dir("images", pattern = "w2"), sep = "/")
> w2 <- read.image(f2)
> f3 <- paste("images", dir("images", pattern = "w3"), sep = "/")
> w3 <- read.image(f3, rgb = TRUE)
> w3@rgb

[1] TRUE
```

```
> w3 <- toGray(w3)
> w3@rgb

[1] FALSE
```

The analysis below requires a lot of memory and processing resources. You will need about 500 Mb RAM to perform the analysis on all 36 images (3 added later). If you have less memory, reduce the number of images you analyse to 5-6 as follows. This will still ocupy about 150-200 Mb of RAM:

```
> w1 <- w1[, , 1:5]
> w2 <- w2[, , 1:5]
> w3 <- w3[, , 1:5]
```

As mentioned above, images use 12-bit out of 16-bit scale. Therefore they need to be normalized before further analysis can be performed. Because different images in one stack can have different intensity range, they have to be normalized separately (`independent = TRUE` in calls to `normalize`).

Exercise 2
Normalize images `w1`, `w2` *and* `w3` *saving results in objects* `n1`, `n2` *and* `n3`. *Using another variable, say* `nx`, *try normalizing with* `independent=FALSE`. *Display* `n1` *and* `nx` *one after the other and check the differences, also make histograms of* `n1` *and* `nx` *excluding background (pixels with intensity smaller than* 0.05*). Why is* `nx` *darker?*

```
> n1 <- normalize(w1, independent = TRUE)
> hist(n1[n1 > 0.05], 50)
> nx <- normalize(w1)
> hist(nx[nx > 0.05], 50)
> n2 <- normalize(w2, independent = TRUE)
> n3 <- normalize(w3, independent = TRUE)
```

EBImage can read remote files via HTTP and FTP protocols. For this lab, one RGB image is provided on the package examples' page:

www.ebi.ac.uk/~osklyar/projects/EBImage/examples/N16_RGB.TIF

This image is a false-colour RGB combination of three different channels for one probe. Channel 1 is given in red, channel 2 in green and channel 3 in blue. In this image, channels were initially normalized to the full grayscale range $[0, 1]$.

Exercise 3
If a network connection is available, read the above RGB file into an R *object,* `w`, *otherwise read it locally from the* 'images' *directory. Ensure that you ontained an RGB image, check if* `w@rgb == TRUE` *and display the image on the screen for visual inspection.* **Tip:** `read.image` *does not use* R *connections, URL's should be passed as* `characters`.

```
> url <- "http://www.ebi.ac.uk/~osklyar/projects/EBImage/examples/N16_RGB.TIF"
> w <- read.image(url, rgb = TRUE)
> display(w)
```

Exercise 4
Check if the downloaded image has the same size as `n1` *(stack size, 3rd dimension is different). Split the false-colour image into three grayscale images, one for each channel,* `c1`, `c2` *and* `c3`. **Tip:** `getRed` *and similar functions can be used to extract channels from RGB images.*

```
> dim(w)
```

```
[1] 696 520   1
```

```
> dim(n1)
```

```
[1] 696 520  11
```

```
> c1 <- getRed(w)
> c2 <- getGreen(w)
> c3 <- getBlue(w)
```

One can use the `Image` function to create new images or combine existing ones into a single image. The function accepts three arguments: `data`, `dim` and `rgb`. The vignette explains how new images can be created and the following example demonstrates how `c1` can be added to `n1` containing the rest of the images for channel 1 (we assume that x and y dimensions of `n1` and `c1` are equal and that only the stack depth, dimension 3, is different):

```
> newdim <- dim(n1)
> newdim[3] <- newdim[3] + dim(c1)[3]
> n1 <- Image(c(n1, c1), newdim, FALSE)
> dim(n1)
```

```
[1] 696 520  12
```

Exercise 5
Following the above example, combine `c2` *and* `c3` *with* `n2` *and* `n3`, *respectively. Ensure that objects* `n1`, `n2` *and* `n3` *have equal dimensions.* **Tip:** *if you did not decrease the number of images before, the result of* `dim` *should produce* `696,520,12`, *i.e. 12 images of size 696 to 520 pixels.*

```
> n2 <- Image(c(n2, c2), newdim, FALSE)
> n3 <- Image(c(n3, c3), newdim, FALSE)
> dim(n2)
```

```
[1] 696 520  12
```

```
> dim(n3)
```

```
[1] 696 520  12
```

Now all images are loaded into the R session.

Before we proceed, try saving images onto your local disk. EBImage supports all formats supported by ImageMagick. The only restriction at the moment is that it saves images with standard values of resolution and compression. This will be changed in future versions. One does not need to specify the format for writing, it is selected internally by the file extension. A correct file extension must be supplied, which can be in lower or higher case, i.e. both '.TIFF' and '.tiff' are supported (as well as probably '.TiFf' etc.).

Exercise 6

Generate a false-colour RGB image by adding together normalized images `n1`*,* `n2`*, and* `n3` *first making copies of those converted to red, green and blue, respectively. Refer to the vignette for an example. Write the RGB image to a TIFF file and to a PNG file passing both times just one file name as argument to* `write.image`*. Check what was written in the TIFF file using* `ping.image`*. What is the difference in files generated and why?* **Tip:** *in a new terminal window, the shell command* 'display' *can be used to display resulting files on the screen.*

```
> rgb <- toRed(n1) + toGreen(n2) + toBlue(n3)

> write.image(rgb, "images/output.tif")
> ping.image("images/output.tif")
> write.image(rgb, "images/output.png")
```

Exercise 7

If you have other images of your own in different formats, use those to practice read/write operations and displaying images with EBImage *(please do not alter the variables created above).*

4 Basic image processing

The package provides two complementary ways to perform image processing. One is to use simple mathematical and arithmetical functions on images using the fact that images are arrays. The other one is using high-level image filters.

4.1 Mathematics and arithmetics with images

Images in EBImage are objects of class *Image*. This class is derived from *array*, therefore most of operations that are valid for arrays can be performed on images. These include arithmetic operations (`+, -, *, /, ^`), mathematical functions like `sin, cos, exp, sqrt` etc., functions like `summary, mean, median, hist, range, rank` etc. Although one can use the above functions on both, grayscale and RGB images, due to the nature of data storage the results will be meaningless for RGB images (RGB images are stored as *integer*s with one-colour-per-byte coding).

Subscripting can be used in exactly the same manner as with arrays to retrieve or substitute data.

In many cases, microscopy images contain just a few very bright objects, whereas the majority are dark. One can try to correct the contrast in such situations using power functions, for example `sqrt`. We use $\{0, 1\}$ as the fix points for the power transformation $x \to x^{\alpha}$ (meaning that the valus in these points do not change in the transformation). For $\alpha < 1$, dark objects become lighter while keeping light objects at about the same gray level; for $\alpha > 1$, darker objects become even darker thus allowing brighter objects to stand out clearer.

Exercise 8

Try applying power function `^0.7` *followed by* `^1.2` *to* `n1` *and inspect the differences. Although stated above, try to explain why the former is lighter than the original image?* **Tip:** *evaluate the shape of $x \to x^{0.7}$ in the range $[0, 1]$. Why applying* `^1.2` *makes the image darker again? Experiment with* `sqrt` *and* `^2`*.*

```
> display(n1^0.7)
> display((n1^0.7)^1.2)
> display(n1^0.7 - (n1^0.7)^1.2)
> display(sqrt(n1))
```

Exercise 9
(Optional) Write a simple normalize function for grayscale images, transforming them to the range $[0, 1]$*, using image airthmetics (*`+, -, /` *operators). Apply the function to* `w1`*. Inspect ranges or histograms before and after normalization.*

```
> donorm <- function(x) {
+     r <- range(x)
+     if (diff(r) == 0)
+         return(x)
+     return((x - r[1])/diff(r))
+ }
> range(w1)
```

```
[1] 0.001907378 0.022034028
```

```
> range(donorm(w1))
```

```
[1] 0 1
```

Image thresholding is an integral part of preprocessing microcopy images for object detection. Simple thresholding can be performed by setting a certain subset of an image to 0 (for background) or to 1 (for foreground), or to any other value. If for example, we want all pixels of image 2 of a copy of `n1` lower than 0.1 to be background, we can use the following syntax:

```
> nx <- n1
> nx2 <- n1[, , 2]
> nx2[nx2 < 0.1] <- 0
> nx[, , 2] <- nx2
```

Here, we made modifications on the extracted image `nx2`, which is then reinserted back instead of image 2 of `n1`.

Exercise 10
(Optional) Explain why we could not call `nx[nx < 0.1] <- 0` *in the above example. Use* 0.3 *threshold to reset parts of image* `nx` *to foreground* (1)*. Create an RGB copy of* `nx` *and reset all regions of this image that are above the threshold* 0.3 *in* `nx` *to* 255 *(red). Display the result to observe what was done.*

Exercise 11
(Optional) Now perform the same thresholding as in the previous example (with converting to RGB) with `nx <- n1^0.5`*. Compare the final RGB image to the previous case, what has changed?*

```
> nx[nx > 0.3] = 1
> nxrgb <- toRGB(nx)
> nxrgb[nx == 1] <- 255
> display(nxrgb)

> nx <- n1^0.5
> nx[nx > 0.3] = 1
> nxrgb <- toRGB(nx)
> nxrgb[nx == 1] <- 255

> display(nxrgb)
```

For practical use, EBImage has a powerful threshold function `thresh` used below.

4.2 Image filters

The package comes with a substantial number of image filters, many of which are interfaced from ImageMagick. For preprocessing microscopy data, one will generally need the following fitlers: `gaussFilter`, `despeckle`, `thresh`, `mOpen` and `mClose`, and `distMap`. Please use the R `help` command to get more information on these filters and their arguments.

Utilising these filters, the preprocessing workflow for microscopy images used in the lab can look as follows.

- Load images from files (already done)
- Normalize images independently (already done)
- Reduce or remove noise if necessary – `despeckle` filter
- Threshold images to obain object masks: locations of objects will be marked with white (1), whereas the rest will be marked with black (0) – `thresh` filter. `thresh` can use some preprocessing, which is effectively the blurring `gaussFilter` used to reduce the noise in images
- Use morphological opening and closing to refine thresholding – `mOpen` and `mClose` filters
- Obtain distance map transforms for all masks, which will be used to identify objects, see the vignette for more details, – function `distMap`

We will omit the `despeckle` step (although some users may try it before continuing) and go directly to thresholding.

Refer to the help page of function `thresh` for the description of its arguments. Generally, it is reasonable to use frame sizes that can fully accomodate several average-sized objects that are expected. Therefore, for the dimensions of objects in the images supplied, reasonable values for thresholding nuclei images (`n1`) are about 100 and for cellular images (`n2`) about 300. The value of offset should be determined empirically as such that gives the best object separation but still does not make objects too small. It is an offset from the mean value used as threshold for data that is assumed to be in the range $[0, 1]$, try the range $[0.0, 0.05]$.

Hereafter, we continue working with channel 1 (images `n1`) and channel 2 (`n2`) only.

Exercise 12
Threshold images `n1` *and* `n2` *into objects* `t1` *and* `t2` *enable preprocessing. Find an optimum between different values of frame size and offset. Check what happens with cellular images if the threshold is set to a low value of* -0.02.

```
> t1 <- thresh(n1, 100, 100, 0.01, TRUE)
> t2 <- thresh(n2, 300, 300, 0, TRUE)

> display(thresh(n2, 300, 300, -0.02, TRUE))
```

Morphological operators of opening and closing allow for refining the thresholded images eliminating sharp edges and small noise. Ususally both filters are applied sequentially and produce different outputs depending on the order in which they are applied. A kernel plays the major role in the results obtained from these filters. A kernel defines a pixel mask, that, scanning either background (opening) or foreground (closing), defines which pixels stay and which are reset. Function `mKernel` generates kernels of different sizes of either round of square shape. Reasonable kernel sizes for nuclear images are between 3 and 7 and for cellular images between 5 and 9. The vignette provides a good example of the usage of these operators.

Exercise 13
Use `mOpen` *and* `mClose` *sequentially to refine* `t1` *and* `t2`*. Try different execution order and different kernels and find optimal parameters.*

```
> t1 <- mClose(mOpen(t1, 1, mKernel(7)), 1, mKernel(7))
> t2 <- mOpen(mClose(t2, 1, mKernel(9)), 1, mKernel(9))
```

The next step in the preprocessing workflow is generating distance maps. `distMap` function does not have any additional arguments except the image itself. The image argument must be supplied a thresholded binary image with values $0, 1$, if a grayscale image is supplied, all non-zero pixels will be treated as 1. In the result, every non-zero pixel vaule will be assigne the distance from this pixel to the nearest background. The resulting image will be grayscale but the data range will be much larger than $[0, 1]$, therefore for displaying it should be normalized.

Exercise 14
Generate distance maps `dm1` *and* `dm2` *from thresholded* `t1` *and* `t2`*.* **Tip:** *do not store normalized values of distance maps, use* `normalize` *only for display purposes.*

```
> dm1 <- distMap(t1)
> dm2 <- distMap(t2)
```

5 Analysis

In this section we will use distance maps generated in the previous section to perform the extraction of basic image descriptors for normalized images `n1` and `n2`. As a result, we want to obtain tables of image descriptors for our 12 probes, which we could further evaluate using statistical methods. The statistical evaluation is out of scope of this course.

5.1 Detecting nuclei

EBImage provides the `wsObjects` function that identifies objects taking distance map images as input. Please read `help(wsObjects)` for full description of this function. More description is also provided in the vignette, where also the arguments and the value of the function are discussed.

When detecting nuclei we do not supply `seeds` as we do not have any preferred locations at this moment. We will value for `seeds` that contains nuclei locations to `wsObjects` when detecting cells. This should significantly improve the detection.

Values of `mind` and `minr` are determined empirically, such that no oversegmentation is observed (not too small) and small objects are not merged into large ones (not too large). As a hint, for nuclei detection, the reasonable range of `mind` is about $[10, 20]$ and of `minr` about $[5, 15]$; for cells, $[20, 50]$ and $[10, 20]$ respectively.

Exercise 15
Use `wsObjects` to detect nuclei from images `dm1`, supply `n1` as a reference, save results to `x1`. Investigate the structure of `x1`, where is the information about objects size and intensity located? What is the reference image used for? Try performing the detection without a reference image, what has changed in the result?

```
> x1 <- wsObjects(dm1, 15, 10, 0.2, ref = n1)
> x1[[1]]$objects[1:2, ]

     [,1] [,2] [,3]     [,4] [,5] [,6]
[1,]  468  440  377 85.68125   80    0
[2,]  191  413  644 85.96354   93    0

> xx1 <- wsObjects(dm1, 15, 10, 0.2)
> xx1[[1]]$objects[1:2, ]

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  468  440  377    0   80    0
[2,]  191  413  644    0   93    0
```

5.2 Detecting cells

Cells can be detected in exactly the same as nuclei supplying `seeds` with detected nuclei locations:

```
> x1[[1]]$objects[1:3, ]

     [,1] [,2] [,3]      [,4] [,5] [,6]
[1,]  468  440  377  85.68125   80    0
[2,]  191  413  644  85.96354   93    0
[3,]  173  235 1205 214.45000  128    0

> seeds <- lapply(x1, function(x) {
+     x$objects[, 1:2]
+ })
> seeds[[1]][1:2, ]

     [,1] [,2]
[1,]  468  440
[2,]  191  413
```

Exercise 16
Analogously to the previous exercise, detect cells from `dm2` *supplying* `seeds`*; store results in* `x2`*. Try the same function without* `seeds`*, store results in other variables for future visual evaluation.*

```
> x2 <- wsObjects(dm2, 35, 15, 0.2, seeds, ref = n2)
> xx2 <- wsObjects(dm2, 35, 15, 0.2, ref = n2)
```

The above method for cell detection has, however, its drawbacks. Cellular images are much more difficult to threshold than nuclear ones, which leads to poor object identification. To omit thresholding, we can supply the scaled original image instead of a distance map to `wsObjects` as in the example below. Here is the explanation why.

Whereas we are guaranteed to have the highest values for points in the object centres on distance maps, we can assume the same for cellular images due to their nature. The images are not that sharp as distance maps, and values are different, otherwise they represent a similar 3D surface suitable for watershed detection.

To assist the detection we supply `seeds` from previously detected nuclei. Images need to be scaled from the range $[0, 1]$ to a significantly large range usually achieved in distance maps, e.g. $[0, 200]$: the algorithm will internally segment the images to integer-based levels, therefore the need for scaling (the top value should not be much larger than the expected object radius):

```
> nx <- n2 * 200
> xx <- wsObjects(nx, 35, 15, 0.2, seeds, ref = n2)
```

The results of `wsObjects` will be visualized in the following section.

5.3 Visualisation

One can use the `wsPaint` function to draw outlines of detected objects on a reference image and fill objects with different semi-transparent colours for controlling the quality of detection by visual inspection. The value of `wsPaint` is an image that can be used as input for marking further objects. In the example below, we first draw detected cells on `n2` and then use the result to draw detected nuclei on top:

```
> im2 <- wsPaint(x2, n2, opac = 0.15)
> im2 <- wsPaint(x1, im2, opac = 0.15)

> display(im2)
> plot.image(toGray(im2))
```

In this way, `im2` is an RGB preview of detected objects.

Exercise 17
Generate preview image `im1` *for* `x1` *and reference* `n1`*. Repeat the last example for cells detected without* `seeds`*, compare the differences. Repeat the same for for object* `xx` *from the last example in the previous subsetion, compare the differences.*

```
> im1 <- wsPaint(x1, n1, opac = 0.3)
> im2x <- wsPaint(xx2, n2, opac = 0.15)
> im2x <- wsPaint(x1, im2x, opac = 0.15)
> imx <- wsPaint(xx, n2, opac = 0.15)
> imx <- wsPaint(x1, imx, opac = 0.15)
```

Exercise 18
Use `wsImages` *to generate single-object image previews from* `xx`, `prev`. **Tip***: the result will be a list of 3D images, use* `display(prev[[1]])` *to display the results, this object will be used below to generate a tiled image.*

```
> prev <- wsImages(xx, n2)
```

5.4 Image descriptors

We will use data from fields `$objects` in `x1[[i]]` and `x2[[i]]` to build summary tables of the following image descriptors (one row per image): image index, number of objects in an image (nuclei for `x1` and cells for `x2`), average object size, average intensity and average perimeter. One can implement many other descriptors by analysing single-object images in `prev`, but this is out of scope of this course.

```
> nimages <- dim(n1)[3]
> obj1 <- matrix(0, ncol = 5, nrow = nimages)
> obj2 <- matrix(0, ncol = 5, nrow = nimages)
> colnames(obj1) <- c("index", "nobj", "size", "intens", "perim")
> colnames(obj2) <- colnames(obj1)
```

The number of detected objects in an image can be obtained from the number of rows in the corresponding the `$objects` matrix. Furthermore, the average size can be obtained by finding the mean value over the corresponding column in the `$objects` matrix:

```
> obj1[, 1] <- 1:nimages
> obj1[, 2] <- as.integer(lapply(x1, function(x) {
+     dim(x$objects)[1]
+ }))
> obj1[, 3] <- as.numeric(lapply(x1, function(x) {
+     mean(x$objects[, 3])
+ }))
> print(obj1[1:2, ])

     index nobj     size intens perim
[1,]     1  102 706.5882      0     0
[2,]     2   27 829.7407      0     0
```

Exercise 19
Calculate `obj1[,4]` *– average intensity, and* `obj1[,5]` *– average perimeter, analogously to the above example using data from* `x1`*. Calculate all data for* `obj2` *using data of* `x2` *analogously.*

```
> obj2[, 1] <- 1:nimages
> obj2[, 2] <- as.integer(lapply(x2, function(x) {
+     dim(x$objects)[1]
+ }))
> for (i in 3:5) {
+     obj1[, i] <- as.numeric(lapply(x1, function(x) {
+         mean(x$objects[, i])
+     }))
+     obj2[, i] <- as.numeric(lapply(x2, function(x) {
+         mean(x$objects[, i])
+     }))
+ }
> print(obj2[1:2, ])
```

```
     index nobj     size   intens    perim
[1,]     1   76 1863.658 443.3452 177.4211
[2,]     2   23 2067.609 373.0148 181.6957
```

Exercise 20
Sort data in `obj1` *by the average number of nuclei accending. Display images for the first indexed in the first and in the last rows - minimum and maximum number of nuclei – was the detection correct?*

```
> obj1 <- obj1[order(obj1[, 2]), ]

> display(n1[, , obj1[1, 1]])
> display(n1[, , obj1[nimages, 1]])
```

The tables `obj1` and `obj2` can now be updated with data for further images from the whole screen. These data can be analysed further using statistical methods, this, however, is out of scope of this course, because the image analysis step is finished at this point.

5.5 Further visualization possiblities

The following function (will be integrated in the package in future) will allow you to make a tiled image from an image stack that is more suitable for simultaneous inspection of multiple images:

```
> dotile <- function(x, width = 5) {
+     .dim <- dim(x)
+     nrow <- as.integer((.dim[3] - 1)/width) + 1
+     res <- Image(0, c(.dim[1] * width, .dim[2] * nrow, 1), rgb = x@rgb)
+     for (i in 1:width) for (j in 1:nrow) if (i + (j - 1) * width <=
+         .dim[3]) {
+         si <- (i - 1) * .dim[1] + 1
+         ei <- si + .dim[1] - 1
+         sj <- (j - 1) * .dim[2] + 1
+         ej <- sj + .dim[2] - 1
+         res[si:ei, sj:ej, 1] <- x[, , i + (j - 1) * width]
+     }
+     return(res)
+ }
```

Exercise 21
Use the above function with `im1`*,* `im2`*,* `im2x` *and* `imx` *to generate images like in Figures 1–4, display the results for visual inspection.* **Tip:** *use* `width = 3`*. Do the same for* `prev[[1]]` *or* `prev[[2]]` *or any other using* `width = 10`*; you should obtain something similar to Figure 5*

```
> tiled1 <- dotile(im1, 3)
> tiled2 <- dotile(im2, 3)
> tiled2x <- dotile(im2x, 3)
> tiledx <- dotile(imx, 3)
> tiledprev <- dotile(prev[[1]], 10)
```

Using the following code (assuming 12 images, 3x4) you one draw the grid lines:

```
> dx <- dim(n1)[1]
> dy <- dim(n1)[2]
> w <- dim(tiled1)[1]
> h <- dim(tiled1)[2]
> x1 <- c(1, 1, 1, 1, dx, 2 * dx)
> x2 <- c(w, w, w, w, dx, 2 * dx)
> y1 <- c(dy, 2 * dy, 3 * dy, 4 * dy, 1, 1)
> y2 <- c(dy, 2 * dy, 3 * dy, 4 * dy, h, h)
> ls <- DrawableLine(x1, y1, x2, y2)
> ls@strokeColor <- "red"
> ls@doFill <- FALSE
> tiled1 <- draw(tiled1, ls)
> tiled2 <- draw(tiled2, ls)
> tiled2x <- draw(tiled2x, ls)
> tiledx <- draw(tiledx, ls)
```

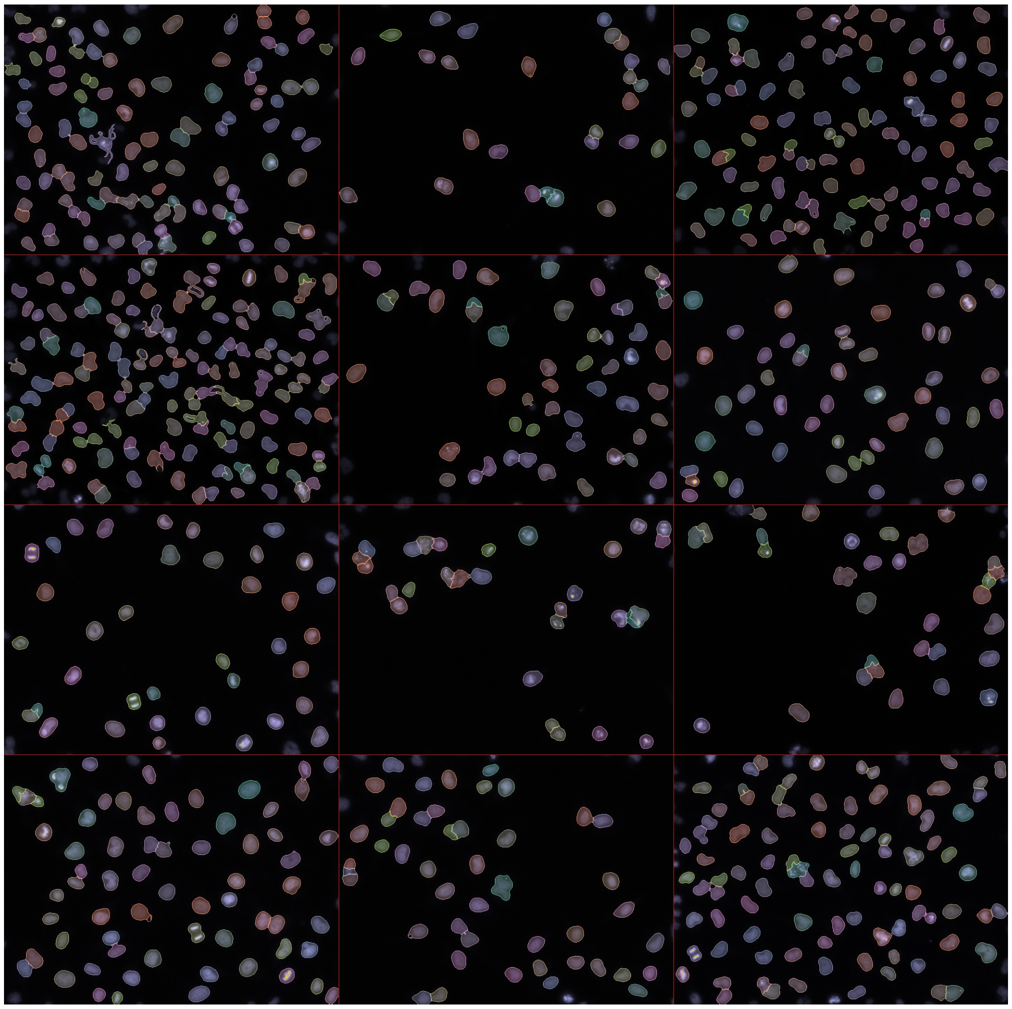
Figure 1: Detected nuclei

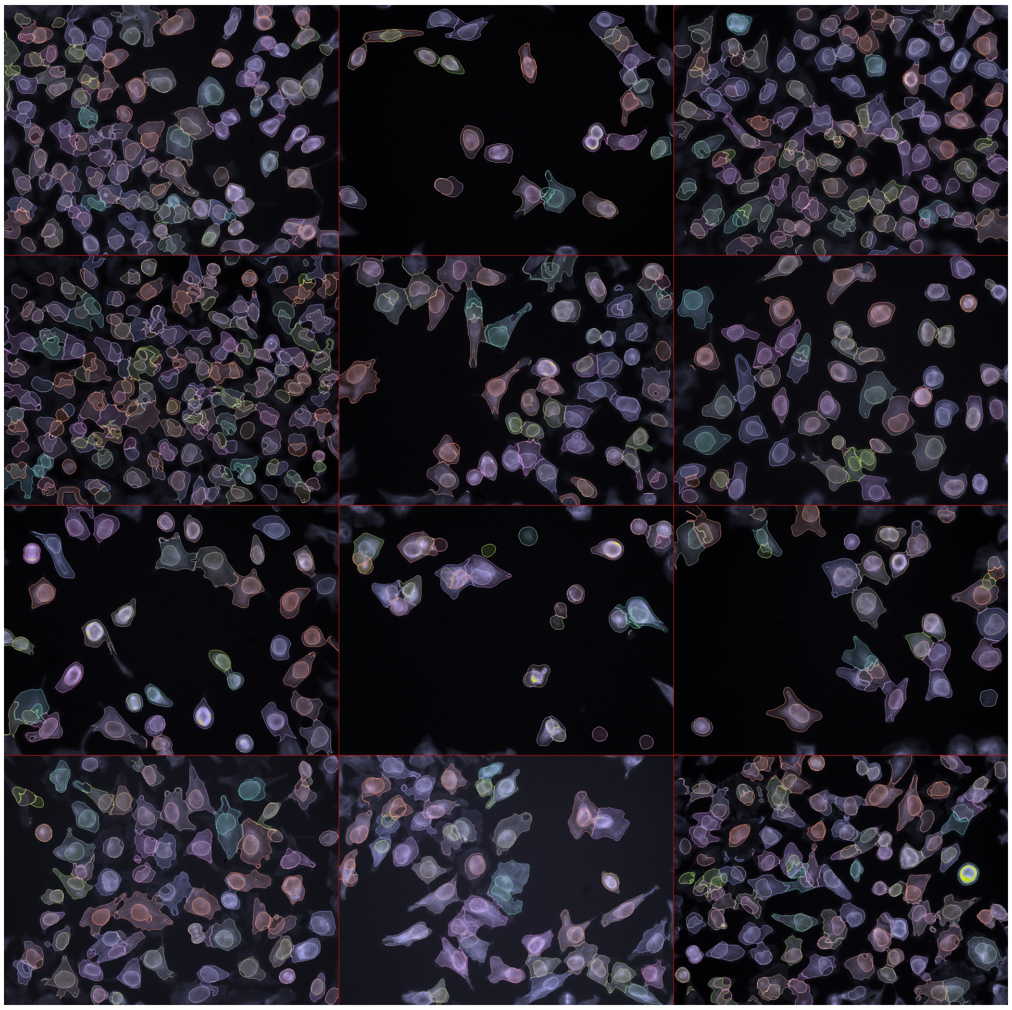

Figure 2: Cells detected using image thresholding (with seeds)

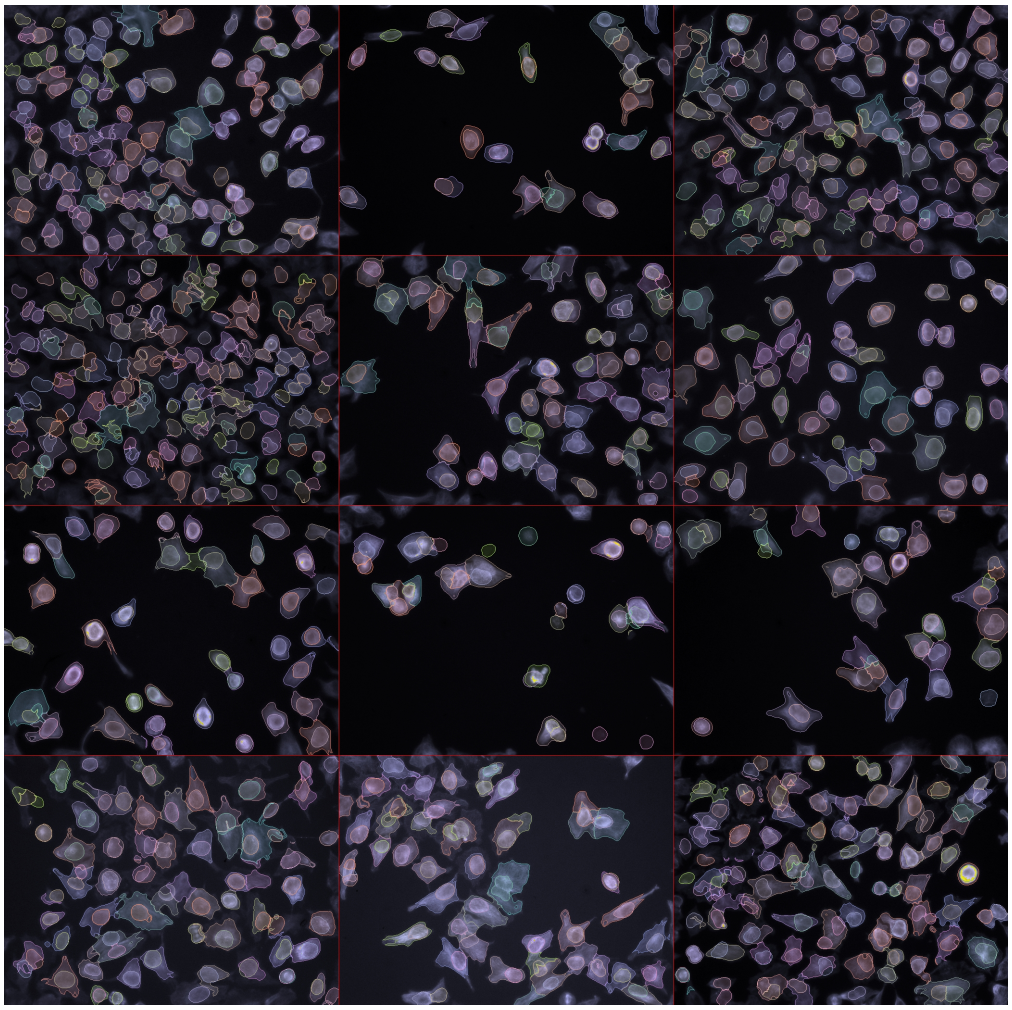

Figure 3: Cells detected using image thresholding (no seeds)

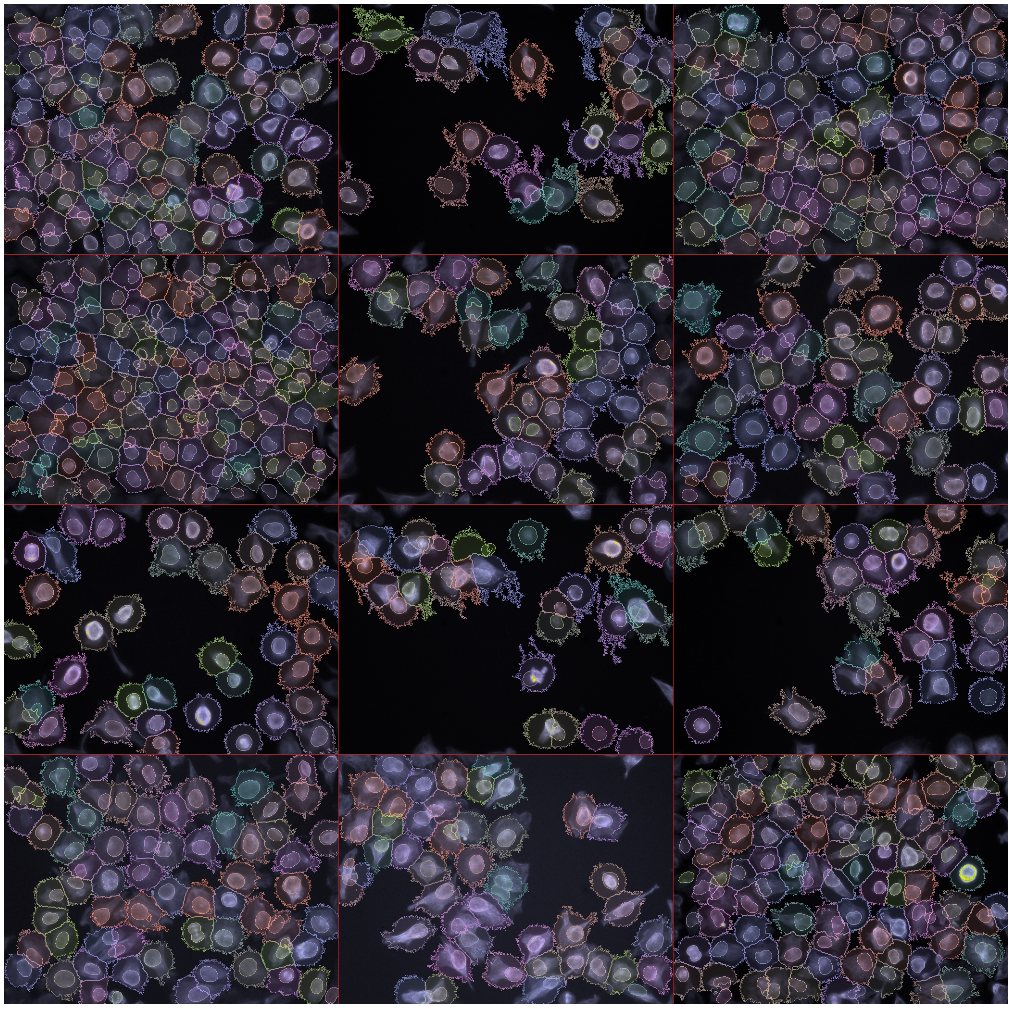

Figure 4: Cells detected by scaling the image without thresholding

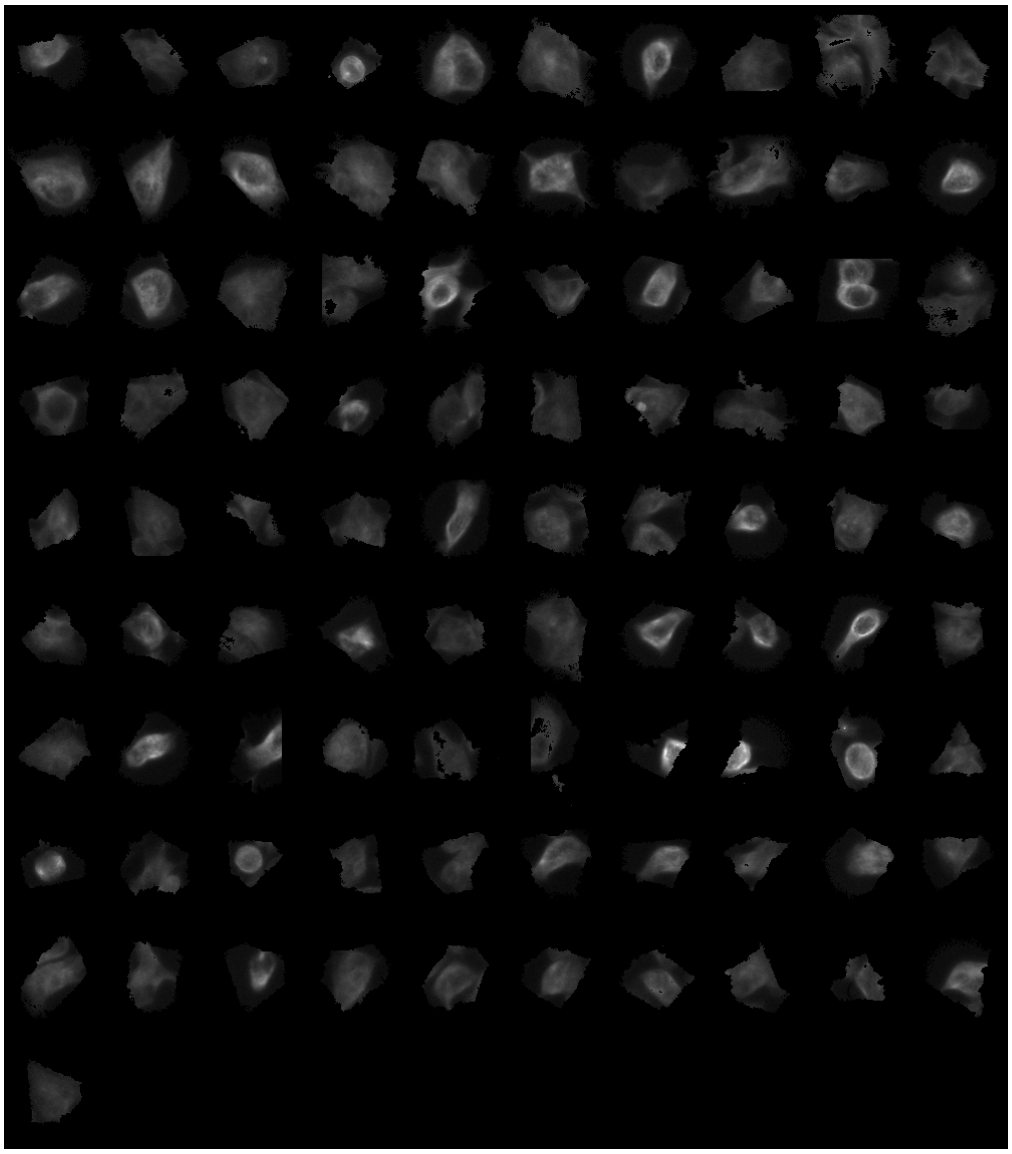

Figure 5: Single-cell images for `prev[[1]]`, tiled