Introduction to *EBImage*

Andrzej Oleś, Gregoire Pau, Oleg Sklyar, Wolfgang Huber

16 February 2017

Abstract

EBImage (http://bioconductor.org/packages/EBImage) provides general purpose functionality for image processing and analysis. In the context of (high-throughput) microscopy-based cellular assays, EBImage offers tools to segment cells and extract quantitative cellular descriptors. This allows the automation of such tasks using the R programming language and facilitates the use of other tools in the R environment for signal processing, statistical modeling, machine learning and visualization with image data.

Package

EBImage 4.17.10

1 Getting started

EBImage (http://bioconductor.org/packages/EBImage) is an R package distributed as part of the Bioconductor (http://bioconductor.org) project. To install the package, start R and enter:

```
source("http://bioconductor.org/biocLite.R")
biocLite("EBImage")
```

Once *EBImage* is installed, it can be loaded by the following command.


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

2 Reading, displaying and writing images

Basic *EBImage* functionality includes reading, writing, and displaying of images. Images are read using the function `readImage`, which takes as input a file name or an URL. To start off, let us load a sample picture distributed with the package.

```
f = system.file("images", "sample.png", package="EBImage")
img = readImage(f)
```

EBImage currently supports three image file formats: `jpeg`, `png` and `tiff`. This list is complemented by the *RBioFormats (https://github.com/aoles/RBioFormats)* package providing support for a much wider range of file formats including proprietary microscopy image data and metadata.

The image which we just loaded can be visualized by the function `display`.

```
display(img)
```

When called from an interactive R session, `display` opens the image in a JavaScript viewer in your web browser. Using the mouse or keyboard shortcuts, you can zoom in and out of the image, pan, and cycle through multiple image frames. Alternatively, the image can be displayed using R's build-in plotting facilities by calling `display` with the argument `method = "raster"`. The image is then drawn on the current device. This allows to easily combine image data with other plotting functionality, for instance, add text labels.

```
display(img, method="raster")
text(x = 20, y = 20, label = "Parrots", adj = c(0,1), col = "ora
nge", cex = 2)
```

The graphics displayed in an R device can be saved using *base* R functions `dev.print` or `dev.copy`. For example, lets save our annotated image as a JPEG file and verify its size on disk.

```
filename = "parrots.jpg"
dev.print(jpeg, filename = filename , width = dim(img)[1], heigh
t = dim(img)[2])
```

```
png
  2
```

```
file.info(filename)$size
```

```
[1] 37850
```

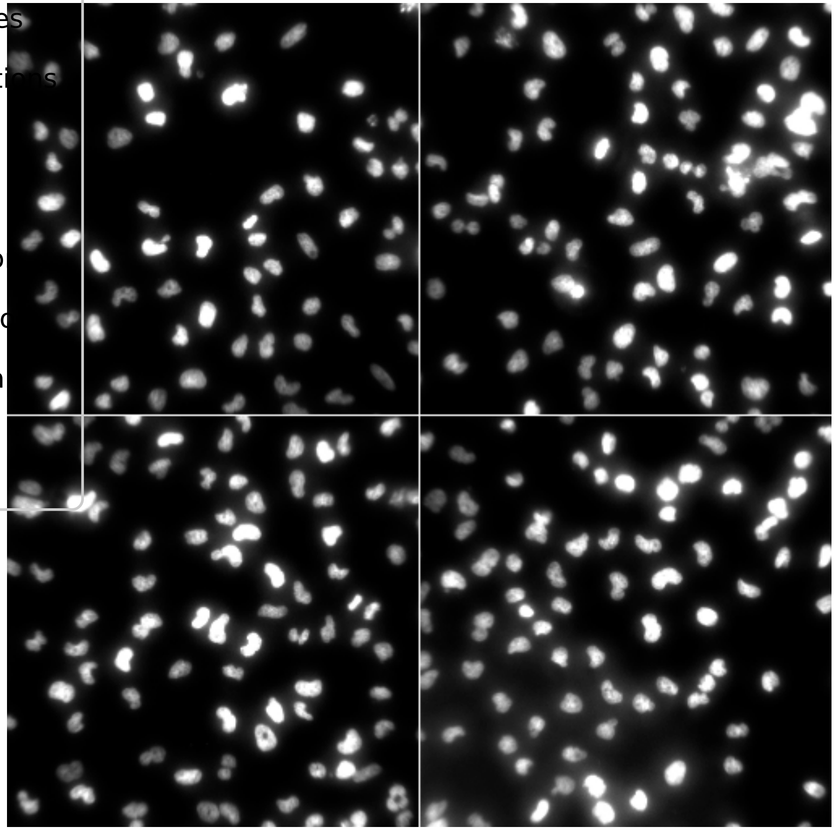
If R is not running interactively, e.g. for code in a package vignette, "raster" becomes the default method in display. The default behavior of display can be overridden globally be setting the "options("EBImage.display") to either "browser" or "raster". This is useful, for example, to preview images inside RStudio.

It is also possible to read and view color images,

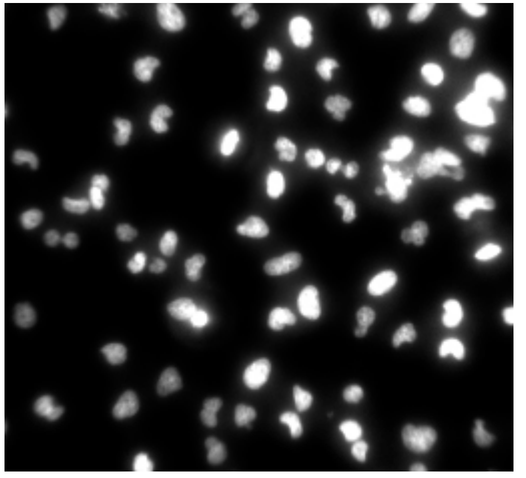
```
imgcol = readImage(system.file("images", "sample-color.png", pac
kage="EBImage"))
display(imgcol)
```

or images containing several frames. If an image consists of multiple frames, they can be displayed all at once in a grid arrangement by specifying the function argument `all = TRUE`,

```
nuc = readImage(system.file("images", "nuclei.tif", package="EBImage"))
display(nuc, method = "raster", all = TRUE)
```



or we can just view a single frame, for example, the second one.



Images can be saved to files using the `writeImage` function. The image that we loaded was a PNG file; suppose now that we want to save this image as a JPEG file. The JPEG format allows to set a quality value between 1 and 100 for its compression algorithm. The default value of the `quality` argument of `writeImage` is 100, here we use a smaller value, leading to smaller file size at the cost of some reduction in image quality.

```
writeImage(imgcol, "sample.jpeg", quality = 85)
```

Similarly, we could have saved the image as a TIFF file and set which compression algorithm we want to use. For a complete list of available parameters see `?writeImage`.

3 Image data representation

EBImage uses a package-specific class `Image` to store and process images. It extends the R base class `array`, and all *EBImage* functions can also be called directly on matrices and arrays. You can find out more about this class by typing `?Image`. Let us peek into the internal structure of an `Image` object.


```
str(img)
```

```
Formal class 'Image' [package "EBImage"] with 2 slots
  ..@ .Data    : num [1:768, 1:512] 0.447 0.451 0.463 0.455 0.46
3 ...
  ..@ colormode: int 0
```

The `.Data` slot contains a numeric array of pixel intensities. We see that in this case the array is two-dimensional, with 768 times 512 elements, and corresponds to the pixel width and height of the image. These dimensions can be accessed using the `dim` function, just like for regular arrays.

```
dim(img)
```

```
[1] 768 512
```

Image data can be accessed as a plain R `array` using the `imageData` accessor,

```
imageData(img)[1:3, 1:6]
```

```
          [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
[1,] 0.4470588 0.4627451 0.4784314 0.4980392 0.5137255 0.5294118
[2,] 0.4509804 0.4627451 0.4784314 0.4823529 0.5058824 0.5215686
[3,] 0.4627451 0.4666667 0.4823529 0.4980392 0.5137255 0.5137255
```

and the `as.array` method can be used to coerce an `Image` to an `array`.

```
is.Image( as.array(img) )
```

```
[1] FALSE
```


The distribution of pixel intensities can be plotted in a histogram, and their range inspected using the `range` function.

```
hist(img)
```

```
range(img)
```

```
[1] 0 1
```

A useful summary of `Image` objects is also provided by the `show` method, which is invoked if we simply type the object's name.

```
img
```

```
Image
  colorMode    : Grayscale
  storage.mode : double
  dim          : 768 512
  frames.total : 1
  frames.render: 1

imageData(object)[1:5,1:6]
          [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
[1,] 0.4470588 0.4627451 0.4784314 0.4980392 0.5137255 0.5294118
[2,] 0.4509804 0.4627451 0.4784314 0.4823529 0.5058824 0.5215686
[3,] 0.4627451 0.4666667 0.4823529 0.4980392 0.5137255 0.5137255
[4,] 0.4549020 0.4666667 0.4862745 0.4980392 0.5176471 0.5411765
[5,] 0.4627451 0.4627451 0.4823529 0.4980392 0.5137255 0.5411765
```

For a more compact representation without the preview of the intensities array use the `print` method with the argument `short` set to `TRUE`.

```
print(img, short=TRUE)
```

```
Image
  colorMode    : Grayscale
  storage.mode : double
  dim          : 768 512
  frames.total : 1
  frames.render: 1
```

Let’s now have a closer look a our color image.

```
print(imgcol, short=TRUE)
```

```
Image
  colorMode    : Color
  storage.mode : double
  dim          : 768 512 3
  frames.total : 3
  frames.render: 1
```

It differs from its grayscale counterpart `img` by the property `colorMode` and the number of dimensions. The `colorMode` slot turns out to be convenient when dealing with stacks of images. If it is set to `Grayscale`, then the third and all higher dimensions of the array are considered as separate image frames corresponding, for instance, to different z-positions, time points, replicates, etc. On the other hand, if `colorMode` is `Color`, then the third dimension is assumed to hold different color channels, and only the fourth and higher dimensions—if present—are used for multiple image frames. `imgcol` contains three color channels, which correspond to the red, green and blue intensities of the photograph. However, this does not necessarily need to be the case, and the number of color channels is arbitrary.

The “frames.total” and “frames.render” fields shown by the object summary correspond to the total number of frames contained in the image, and to the number of rendered frames. These numbers can be accessed using the function `numberOfFrames` by specifying the `type` argument.

```
numberOfFrames(imgcol, type = "render")
```

```
[1] 1
```

```
numberOfFrames(imgcol, type = "total")
```

```
[1] 3
```

Image frames can be extracted using `getFrame` and `getFrames`. `getFrame` returns the i-th frame contained in the image y. If `type` is `"total"`, the function is unaware of the color mode and returns an xy-plane. For `type="render"` the function returns the i-th image as shown by the display function. While `getFrame` returns just a single frame, `getFrames` retrieves a list of frames which can serve as input to `lapply`-family functions. See the "Global thresholding" section for an illustration of this approach.

Finally, if we look at our cell data,

```
nuc
```

```
Image 
  colorMode    : Grayscale 
  storage.mode : double 
  dim          : 510 510 4 
  frames.total : 4 
  frames.render: 4 

imageData(object)[1:5,1:6,1]
           [,1]       [,2]       [,3]       [,4]       [,5]       [,6]
[1,] 0.06274510 0.07450980 0.07058824 0.08235294 0.10588235 0.09803922
[2,] 0.06274510 0.05882353 0.07843137 0.09019608 0.09019608 0.10588235
[3,] 0.06666667 0.06666667 0.08235294 0.07843137 0.09411765 0.09411765
[4,] 0.06666667 0.06666667 0.07058824 0.08627451 0.08627451 0.09803922
[5,] 0.05882353 0.06666667 0.07058824 0.08235294 0.09411765 0.10588235
```

We see that it contains 4 total frames that correspond to the 4 separate greyscale images, as indicated by “frames.render”.

4 Color management

As described in the previous section, the class `Image` extends the base class `array` and uses `colorMode` to store how the color information of the multi-dimensional data should be handled. The function `colorMode` can be used to access and change this property, modifying the rendering mode of an image. For example, if we take a `Color` image and change its mode to `Grayscale`, then the image won’t display as a single color image anymore but rather as three separate grayscale frames corresponding to the red, green and blue channels. The function `colorMode` does not change the actual content of the image but only changes the way the image is rendered by *EBImage*.


```
colorMode(imgcol) = Grayscale
display(imgcol, all=TRUE)
```

Color space conversions between `Grayscale` and `Color` images are performed using the function `channel`. It has a flexible interface which allows to convert either way between the modes, and can be used to extract color channels. Unlike `colorMode`, `channel` changes the pixel intensity values of the image.

`Color` to `Grayscale` conversion modes include taking a uniform average across the RGB channels, and a weighted luminance preserving conversion mode better suited for display purposes.

The `asred`, `asgreen` and `asblue` modes convert a grayscale image or array into a color image of the specified hue.

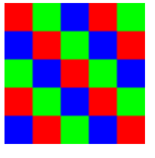
The convenience function `toRGB` promotes a grayscale image to RGB color space by replicating it across the red, green and blue channels, which is equivalent to calling `channel` with mode set to `rgb`. When displayed, this image doesn't look different from its grayscale origin, which is expected because the information between the color channels is the same. To combine three grayscale images into a single rgb image use the function `rgbImage`.

The function `Image` can be used to construct a color image from a character vector or array of named R colors (as listed by `colors()`) and/or hexadecimal strings of the form "#rrggbb" or "#rrggbbaa".

```
colorMat = matrix(rep(c("red","green", "#0000ff"), 25), 5, 5)
colorImg = Image(colorMat)
colorImg
```

```
Image
  colorMode    : Color
  storage.mode : double
  dim          : 5 5 3
  frames.total : 3
  frames.render: 1

imageData(object)[1:5,1:5,1]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    1    0
[2,]    0    1    0    0    1
[3,]    0    0    1    0    0
[4,]    1    0    0    1    0
[5,]    0    1    0    0    1
```

```
display(colorImg, interpolate=FALSE)
```



5 Manipulating images

Being numeric arrays, images can be conveniently manipulated by any of R's arithmetic operators. For example, we can produce a negative image by simply subtracting the image from its maximum value.

```
img_neg = max(img) - img
display( img_neg )
```

We can also increase the brightness of an image through addition, adjust the contrast through multiplication, and apply gamma correction through exponentiation.

```
img_comb = combine(
  img,
  img + 0.3,
  img * 2,
  img ^ 0.5
)

display(img_comb, all=TRUE)
```

In the example above we have used `combine` to merge individual images into a single multi-frame image object.

Furthermore, we can crop and threshold images with standard matrix operations.

```
img_crop = img[366:749, 58:441]
img_thresh = img_crop > .5
display(img_thresh)
```

The thresholding operation returns an `Image` object with binarized pixels values. The R data type used to store such an image is `logical`.

```
img_thresh
```

```
Image
  colorMode    : Grayscale
  storage.mode : logical
  dim          : 384 384
  frames.total : 1
  frames.render: 1

imageData(object)[1:5,1:6]
      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
[1,] FALSE FALSE FALSE FALSE FALSE FALSE
[2,] FALSE FALSE FALSE FALSE FALSE FALSE
[3,] FALSE FALSE FALSE FALSE FALSE FALSE
[4,] FALSE FALSE FALSE FALSE FALSE FALSE
[5,] FALSE FALSE FALSE FALSE FALSE FALSE
```

For image transposition, use `transpose` rather than R's *base* function `t`. This is because the former one works also on color and multiframe images by swapping its spatial dimensions.

```
img_t = transpose(img)
display( img_t )
```

6 Spatial transformations

We just saw one type of spatial transformation, transposition, but there are many more, for example translation, rotation, reflection and scaling. `translate` moves the image plane by the specified two-dimensional vector in such a way that pixels that end up outside the image region are cropped, and pixels that enter into the image region are set to background.

```
img_translate = translate(img, c(100,-50))
display(img_translate)
```

The background color can be set using the argument `bg.col` common to all relevant spatial transformation functions. The default sets the value of background pixels to zero which corresponds to black. Let us demonstrate the use of this argument with `rotate` which rotates the image clockwise by the given angle.

```
img_rotate = rotate(img, 30, bg.col = "white")
display(img_rotate)
```

To scale an image to desired dimensions use `resize`. If you provide only one of either width or height, the other dimension is automatically computed keeping the original aspect ratio.

```
img_resize = resize(img, w=256, h=256)
display(img_resize )
```

The functions `flip` and `flop` reflect the image around the image horizontal and vertical axis, respectively.

```
img_flip = flip(img)
img_flop = flop(img)

display(combine(img_flip, img_flop), all=TRUE)
```

Spatial linear transformations are implemented using the general `affine` transformation. It maps image pixel coordinates `px` using a 3x2 transformation matrix `m` in the following way: `cbind(px, 1) %*% m`. For example, horizontal sheer mapping can be applied by

```
m =  matrix(c(1, -.5, 128, 0, 1, 0), nrow=3, ncol=2)
img_affine = affine(img, m)
display( img_affine )
```

7 Filtering

7.1 Linear filters

A common preprocessing step involves cleaning up the images by removing local artifacts or noise through smoothing. An intuitive approach is to define a window of a selected size around each pixel and average the values within that neighborhood. After applying this procedure to all pixels, the new, smoothed image is obtained.

Mathematically, this can be expressed as

$$f'(x,y) = \frac{1}{N} \sum_{s=-a}^{a} \sum_{t=-a}^{a} f(x+s, y+t),$$

where $f(x,y)$ is the value of the pixel at position (x,y), and a determines the window size, which is $2a+1$ in each direction. $N=(2a+1)^2$ is the number of pixels averaged over, and f' is the new, smoothed image.

More generally, we can replace the moving average by a weighted average, using a weight function w, which typically has the highest value at the window midpoint ($s=t=0$) and then decreases towards the edges.

$$(w * f)(x,y) = \sum_{s=-\infty}^{+\infty} \sum_{t=-\infty}^{+\infty} w(s,t)\, f(x+s, y+s)$$

For notational convenience, we let the summations range from $-\infty$ to $+\infty$, even if in practice the sums are finite and w has only a finite number of non-zero values. In fact, we can think of the weight function w as another image, and this operation is also called the *convolution* of the images f and w, indicated by the the symbol $*$. Convolution is a linear operation in the sense that $w*(c_1f_1+c_2f_2)=c_1w*f_1+c_2w*f_2$ for any two images f_1, f_2 and numbers c_1, c_2.

In *EBImage (http://bioconductor.org/packages/EBImage)*, the 2-dimensional convolution is implemented by the function `filter2`, and the auxiliary function `makeBrush` can be used to generate the weight function. In fact, `filter2` does not directly perform the summation indicated in the equation above. Instead, it uses the Fast Fourier Transformation in a way that is mathematically equivalent but computationally more efficient.

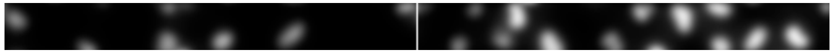
```
w = makeBrush(size = 31, shape = 'gaussian', sigma = 5)
plot(w[(nrow(w)+1)/2, ], ylab = "w", xlab = "", cex = 0.7)
```

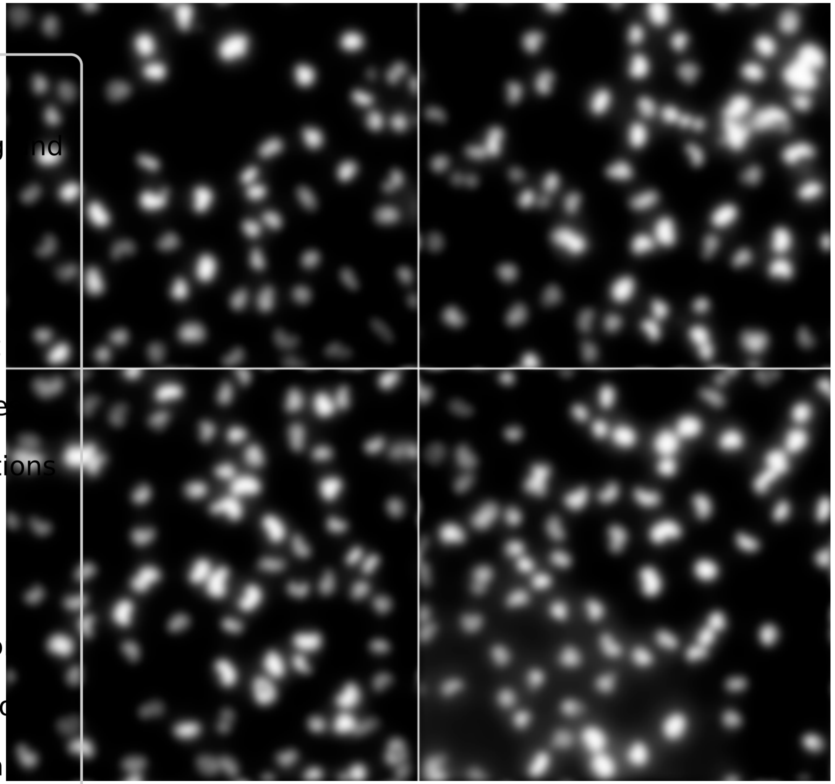
```
img_flo = filter2(img, w)
display(img_flo)
```

Here we have used a Gaussian filter of width 5 given by `sigma`. Other available filter shapes include `"box"` (default), `"disc"`, `"diamond"` and `"line"`, for some of which the kernel can be binary; see `?makeBrush` for details.

If the filtered image contains multiple frames, the filter is applied to each frame separately. For convenience, images can be also smoothed using the wrapper function `gblur` which performs Gaussian smoothing with the filter size automatically adjusted to `sigma`.

```
nuc_gblur = gblur(nuc, sigma = 5)
display(nuc_gblur, all=TRUE )
```





In signal processing the operation of smoothing an image is referred to as low-pass filtering. High-pass filtering is the opposite operation which allows to detect edges and sharpen images. This can be done, for instance, using a Laplacian filter.

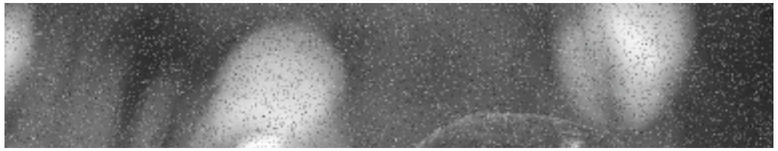
```
fhi = matrix(1, nrow = 3, ncol = 3)
fhi[2, 2] = -8
img_fhi = filter2(img, fhi)
display(img_fhi)
```

7.2 Median filter

Another approach to perform noise reduction is to apply a median filter, which is a non-linear technique as opposed to the low pass convolution filter described in the previous section. Median filtering is particularly effective in the case of speckle noise, and has the advantage of removing noise while preserving edges.

The local median filter works by scanning the image pixel by pixel, replacing each pixel by the median on of its neighbors inside a window of specified size. This filtering technique is provided in *EBImage* by the function `medianFilter` . We demonstrate its use by first corrupting the image with uniform noise, and reconstructing the original image by median filtering.

```
l = length(img)
n = l/10
pixels = sample(l, n)
img_noisy = img
img_noisy[pixels] = runif(n, min=0, max=1)
display(img_noisy)
```



```
img_median = medianFilter(img_noisy, 1)
display(img_median)
```

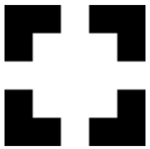
7.3 Morphological operations

Binary images are images which contain only two sets of pixels, with values, say 0 and 1, representing the background and foreground pixels. Such images are subject to several non-linear morphological operations: erosion, dilation, opening, and closing. These operations work by overlaying a mask, called the structuring element, over the binary image in the following way:

- erosion: For every foreground pixel, put the mask around it, and if any pixel covered by the mask is from the background, set the pixel to background.
- dilation: For every background pixel, put the mask around it, and if any pixel covered by the mask is from the foreground, set the pixel to foreground.

```
shapes = readImage(system.file('images', 'shapes.png', package='EBImage'))
logo = shapes[110:512,1:130]
display(logo)
```

```
kern = makeBrush(5, shape='diamond')
display(kern, interpolate=FALSE)
```



```
logo_erode= erode(logo, kern)
logo_dilate = dilate(logo, kern)

display(combine(logo_erode, logo_dilate), all=TRUE)
```

Opening and closing are combinations of the two operations above: opening performs erosion followed by dilation, while closing does the opposite, i.e, performs dilation followed by erosion. Opening is useful for morphological noise removal, as it removes small objects from the background, and closing can be used to fill small holes in the foreground. These operations are implemented by `opening` and `closing`.

8 Thresholding

8.1 Global thresholding

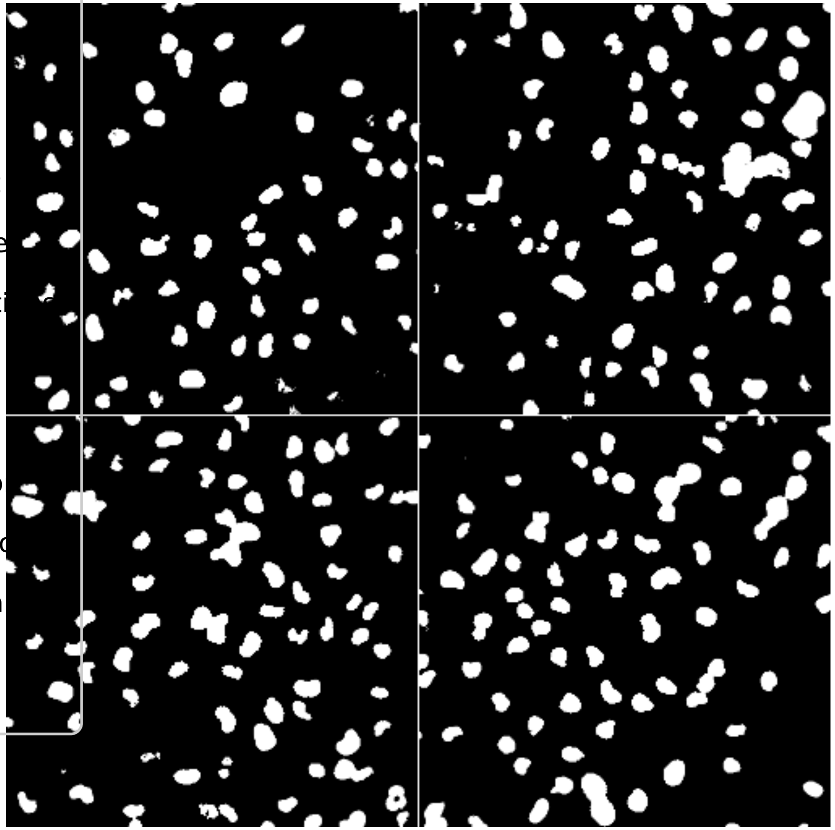
In the “Manipulating images” section we have already demonstrated how to set a global threshold on an image. There we used an arbitrary cutoff value. For images whose distribution of pixel intensities follows a bi-modal histogram a more systematic approach involves using the Otsu’s method. Otsu’s method is a technique to automatically perform clustering-based image thresholding. Assuming a bi-modal intensity distribution, the algorithm separates image pixels into foreground and background. The optimal threshold value is determined by minimizing the combined intra-class variance.

Otsu’s threshold can be calculated using the function `otsu`. When called on a multi-frame image, the threshold is calculated for each frame separately resulting in a output vector of length equal to the total number of frames in the image.

```
threshold = otsu(nuc)
threshold
```

```
[1] 0.3535156 0.4082031 0.3808594 0.4121094
```

```
nuc_th = combine( mapply(function(frame, th) frame > th, getFrames(nuc), threshold, SIMPLIFY=FALSE) )
display(nuc_th, all=TRUE)
```



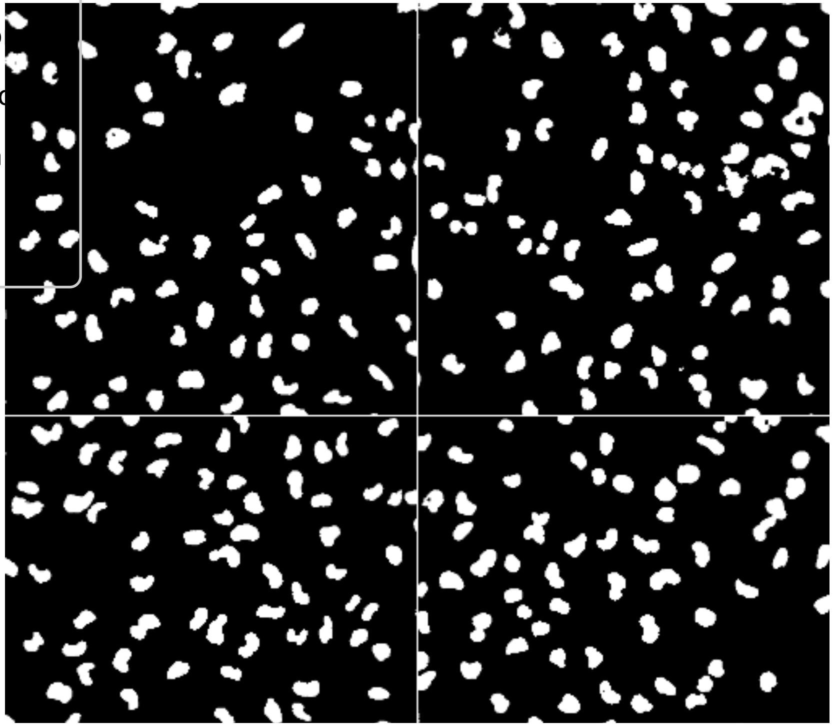
Note the use of `getFrames` to split the image into a list of individual frames, and `combine` to merge the results back together.

8.2 Adaptive thresholding

The idea of adaptive thresholding is that, compared to straightforward thresholding from the previous section, the threshold is allowed to be different in different regions of the image. In this way, one can anticipate spatial dependencies of the underlying background signal caused, for instance, by uneven illumination or by stray signal from nearby bright objects.

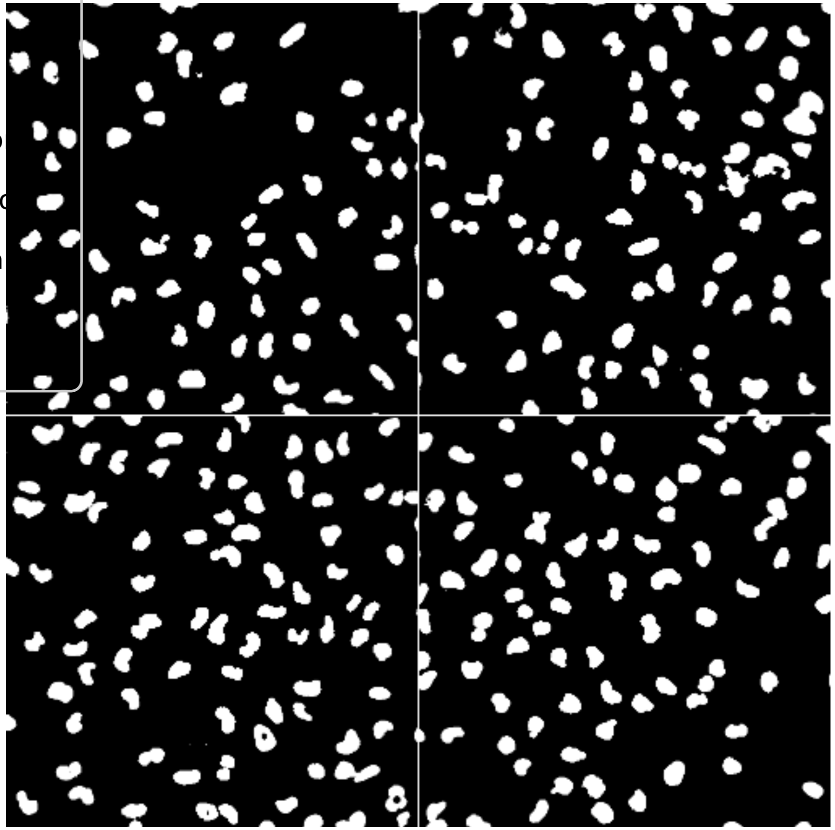
Adaptive thresholding works by comparing each pixel's intensity to the background determined from a local neighbourhood. This can be achieved by comparing the image to its smoothed version, where the filtering window is bigger than the typical size of objects we want to capture.

```
disc = makeBrush(31, "disc")
disc = disc / sum(disc)
offset = 0.05
nuc_bg = filter2( nuc, disc )
nuc_th = nuc > nuc_bg + offset
display(nuc_th, all=TRUE)
```



This technique assumes that the objects are relatively sparsely distributed in the image, so that the signal distribution in the neighborhood is dominated by background. While for the nuclei in our images this assumption makes sense, for other situations you may need to make different assumptions. The adaptive thresholding using a linear filter with a rectangular box is provided by `thresh`, which uses a faster implementation compared to directly using `filter2`.

```
display( thresh(nuc, w=15, h=15, offset=0.05), all=TRUE )
```



9 Image segmentation

Image segmentation performs partitioning of an image, and is typically used to identify objects in an image. Non-touching connected objects can be segmented using the function `bwlabel`, while `watershed` and `propagate` use more sophisticated algorithms able to separate objects which touch each other.

`bwlabel` finds every connected set of pixels other than the background, and relabels these sets with a unique increasing integer. It can be called on a thresholded binary image in order to extract objects.

```
logo_label = bwlabel(logo)
table(logo_label)
```

```
logo_label
     0     1     2     3     4     5     6     7
42217  1375  2012   934  1957  1135  1697  1063
```

The pixel values of the `logo_label` image range from 0 corresponding to background to the number of objects it contains, which is given by

```
max(logo_label)
```

```
[1] 7
```

To display the image we normalize it to the (0,1) range expected by the display function. This results in different objects being rendered with a different shade of gray.

```
display( normalize(logo_label) )
```

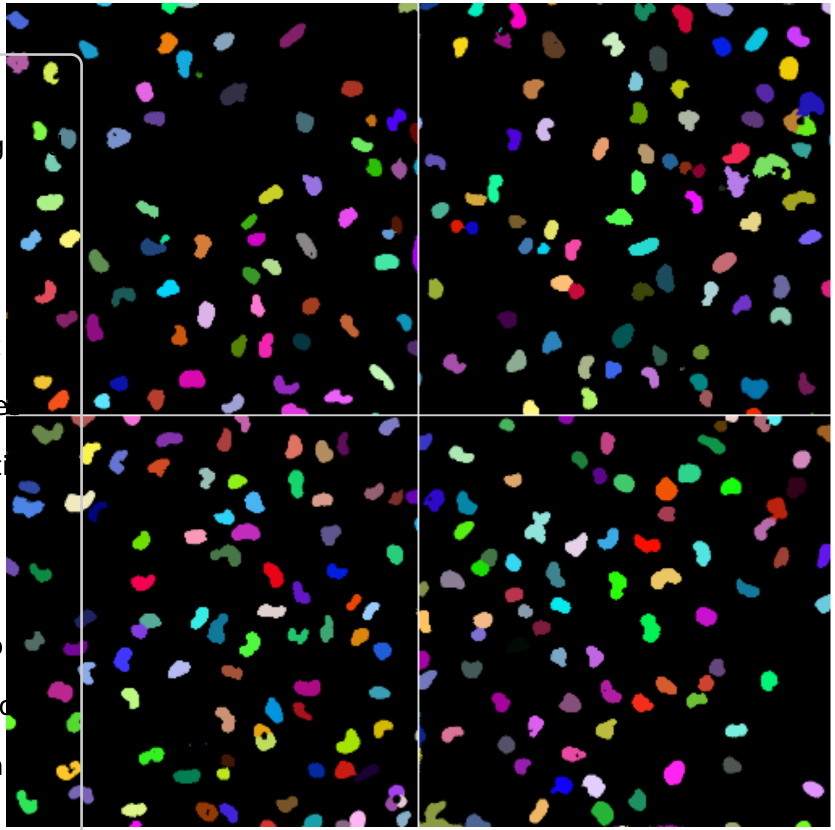
The horizontal grayscale gradient which can be observed reflects to the way `bwlabel` scans the image and labels the connected sets: from left to right and from top to bottom. Another way of visualizing the segmentation is to use the `colorLabels` function, which color codes the objects by a random permutation of unique colors.

```
display( colorLabels(logo_label) )
```

9.1 Watershed

Some of the nuclei in `nuc` are quite close to each other and get merged into one big object when thresholded, as seen in `nuc_th`. `bwlabel` would incorrectly identify them as a single object. The watershed transformation allows to overcome this issue. The `watershed` algorithm treats a grayscale image as a topographic relief, or heightmap. Objects that stand out of the background are identified and separated by flooding an inverted source image. In case of a binary image its distance map can serve as the input heightmap. The distance map, which contains for each pixel the distance to the nearest background pixel, can be obtained by `distmap`.

```
nmask = watershed( distmap(nuc_th), 2 )
display(colorLabels(nmask), all=TRUE)
```

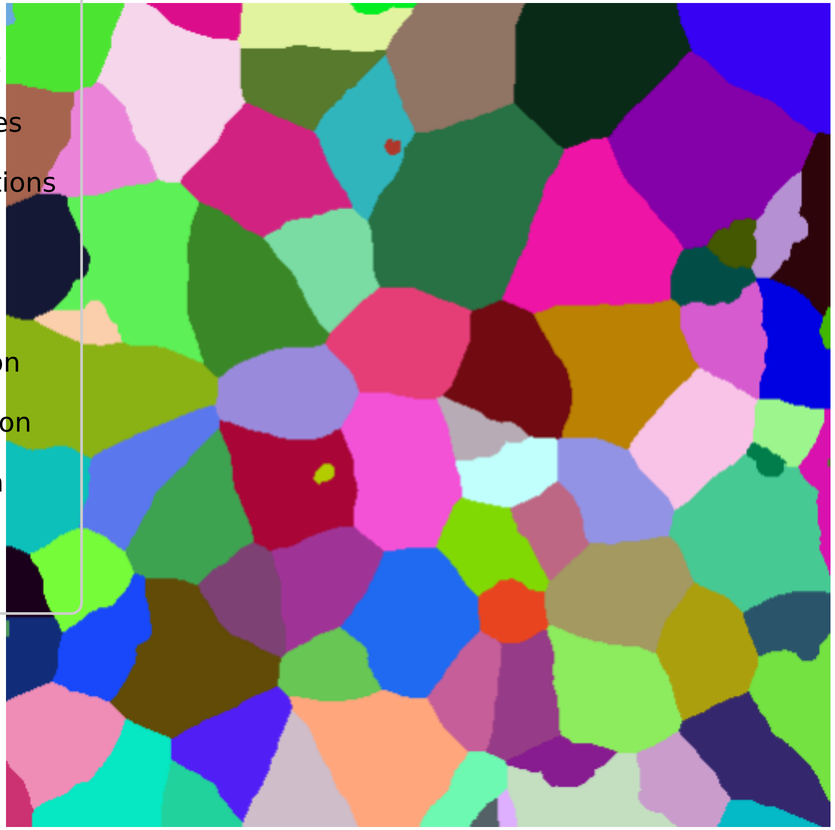


9.2 Voronoi tesselation

Voronoi tessellation is useful when we have a set of seed points (or regions) and want to partition the space that lies between these seeds in such a way that each point in the space is assigned to its closest seed. This function is implemented in *EBImage* by the function `propagate`. Let us illustrate the concept of Voronoi tessalation on a basic example. We use the nuclei mask `nmask` as seeds and partition the space between them.

```
voronoiExamp = propagate(seeds = nmask, x = nmask, lambda = 100)
voronoiPaint = colorLabels (voronoiExamp)
display(voronoiPaint)
```

```
Only the first frame of the image stack is displayed.
To display all frames use 'all = TRUE'.
```



The basic definition of Voronoi tessellation, which we have given above, allows for two generalizations:

- By default, the space that we partition is the full, rectangular image area, but indeed we could restrict ourselves to any arbitrary subspace.

This is akin to finding the shortest distance from each point to the next seed not in a simple flat landscape, but in a landscape that is interspersed by lakes and rivers (which you cannot cross), so that all paths need to remain on the land. `propagate` allows for this generalization through its `mask` argument.

- By default, we think of the space as flat – but in fact it could have hills and canyons, so that the distance between two points in the landscape not only depends on their x- and y-positions but also on the ascents and descents, up and down in z-direction, that lie in between. You can specify such a landscape to `propagate` through its `x` argument.

Mathematically, we can say that instead of the simple default case (a flat rectangle image with an Euclidean metric), we perform the Voronoi segmentation on a Riemann manifold, which can have an arbitrary shape and an arbitrary metric. Let us use the notation x and y for the column and row coordinates of the image, and z for the elevation of the landscape. For two neighboring points, defined by coordinates (x, y, z) and $(x + dx, y + dy, z + dz)$, the distance between them is given by

$$ds = \sqrt{\frac{2}{\lambda + 1}\left[\lambda\left(dx^2 + dy^2\right) + dz^2\right]}.$$

For $\lambda = 1$, this reduces to $ds = (dx^2 + dy^2 + dz^2)^{1/2}$. Distances between points further apart are obtained by summing ds along the shortest path between them. The parameter $\lambda \geq 0$ has been introduced as a convenient control of the relative weighting between sideways movement (along the x and y axes) and vertical movement. Intuitively, if you imagine yourself as a hiker in such a landscape, by choosing λ you can specify how much you are prepared to climb up and down to overcome a mountain, versus sideways walking around it. When λ is large, the expression becomes equivalent to $ds = \sqrt{dx^2 + dy^2}$, i. e., the importance of dz becomes negligible. This is what we did when we used `lambda = 100` in our `propagate` example.

A more advanced application of `propagate` to the segmentation of cell bodies is presented in the "Cell segmentation example" section.

10 Object manipulation

10.1 Object removal

EBImage defines an object mask as a set of pixels with the same unique integer value. Typically, images containing object masks are the result of segmentation functions such as `bwalabel`, `watershed`, or `propagate`. Objects can be removed from such images by `rmObject`, which deletes objects from the mask simply by setting their pixel values to 0. By default, after object removal all the remaining objects are relabeled so that the highest object ID corresponds to the number of objects in the mask. The `reenumerate` argument can be used to change this behavior and to preserve original object IDs.

```
objects = list(
    seq.int(from = 2, to = max(logo_label), by = 2),
    seq.int(from = 1, to = max(logo_label), by = 2)
    )
logos = combine(logo_label, logo_label)
z = rmObjects(logos, objects, reenumerate=FALSE)
display(z, all=TRUE)
```

In the example above we demonstrate how the object removal function can be applied to a multi-frame image by providing a list of object indicies to be removed from each frame. Additionally we have set `reenumerate` to `FALSE` keeping the original object IDs.

```
showIds = function(image) lapply(getFrames(image), function(frame) unique(as.vector(frame)))

showIds(z)
```

```
[[1]]
[1] 0 1 3 5 7

[[2]]
[1] 0 2 4 6
```

Recall that 0 stands for the background. If at some stage we decide to relabel the objects, we can use for this the standalone function `reenumarate`.

```
showIds( reenumerate(z) )
```

```
[[1]]
[1] 0 1 2 3 4

[[2]]
[1] 0 1 2 3
```

10.2 Filling holes and regions

Holes in object masks can be filled using the function `fillHull`.

```
filled_logo = fillHull(logo)
display(filled_logo)
```

`floodFill` fills a region of an image with a specified color. The filling starts at the given point, and the filling region is expanded to a connected area in which the absolute difference in pixel intensities remains below `tolerance`. The color specification uses R color names for `Color` images, and numeric values for `Grayscale` images.

```
rgblogo = toRGB(logo)
rgblogo = floodFill(rgblogo, c(50, 50), "red")
rgblogo = floodFill(rgblogo, c(100, 50), "green")
rgblogo = floodFill(rgblogo, c(150, 50), "blue")
display( rgblogo )
```


```
display( floodFill(img, c(444, 222), col=0.2, tolerance=0.2) )
```

10.3 Highlighting objects

Given an image containing object masks, the function `paintObjects` can be used to highlight the objects from the mask in the target image provided in the `tgt` argument. Objects can be outlined and filled with colors of given opacities specified in the `col` and `opac` arguments, respectively. If the color specification is missing or equals `NA` it is not painted.

```
d1 = dim(img)[1:2]
overlay = Image(dim=d1)
d2 = dim(logo_label)-1

offset = (d1-d2) %/% 2

overlay[offset[1]:(offset[1]+d2[1]), offset[2]:(offset[2]+d2[2])] = logo_label

img_logo = paintObjects(overlay, toRGB(img), col=c("red", "yellow"), opac=c(1, 0.3), thick=TRUE)

display( img_logo )
```

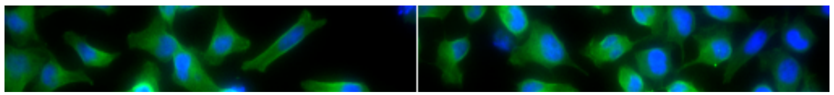
In the example above we have created a new mask `overlay` matching the size of our target image `img`, and copied the mask containing the "EBImage" logo into that overlay mask. The output of `paintObjects` retains the color mode of its target image, therefore in order to have the logo highlighted in color it was necessary to convert `img` to an RGB image first, otherwise the result would be a grayscale image. The `thick` argument controls the object contour drawing: if set to `FALSE`, only the inner one-pixel wide object boundary is marked; if set to `TRUE`, also the outer boundary gets highlighted resulting in an increased two-pixel contour width.

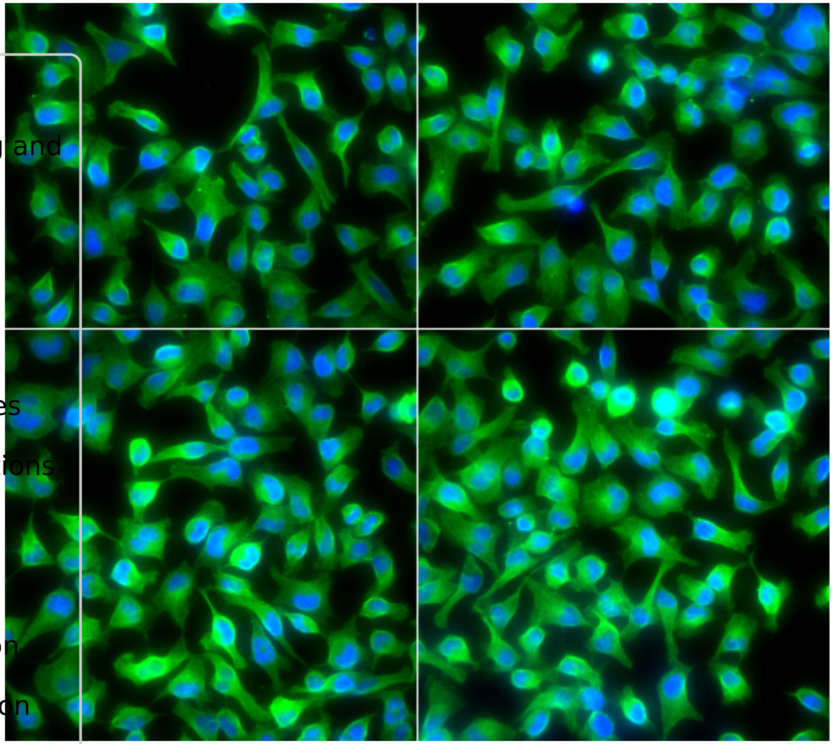
11 Cell segmentation example

We conclude our vignette by applying the functions described before to the task of segmenting cells. Our goal is to computationally identify and qualitatively characterize the cells in the sample fluorescent microscopy images. Even though this by itself may seem a modest goal, this approach can be applied to collections containing thousands of images, an that need no longer to be an modest aim!

We start by loading the images of nuclei and cell bodies. To visualize the cells we overlay these images as the green and the blue channel of a false-color image.

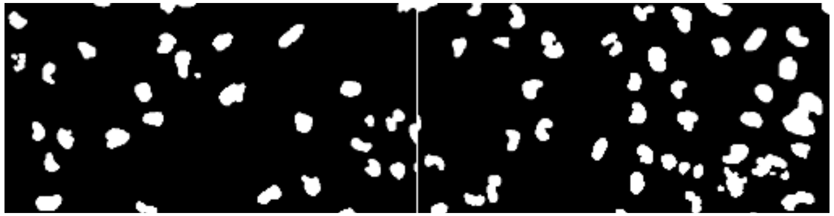
```
nuc = readImage(system.file('images', 'nuclei.tif', package='EBImage'))
cel = readImage(system.file('images', 'cells.tif', package='EBImage'))

cells = rgbImage(green=1.5*cel, blue=nuc)
display(cells, all = TRUE)
```

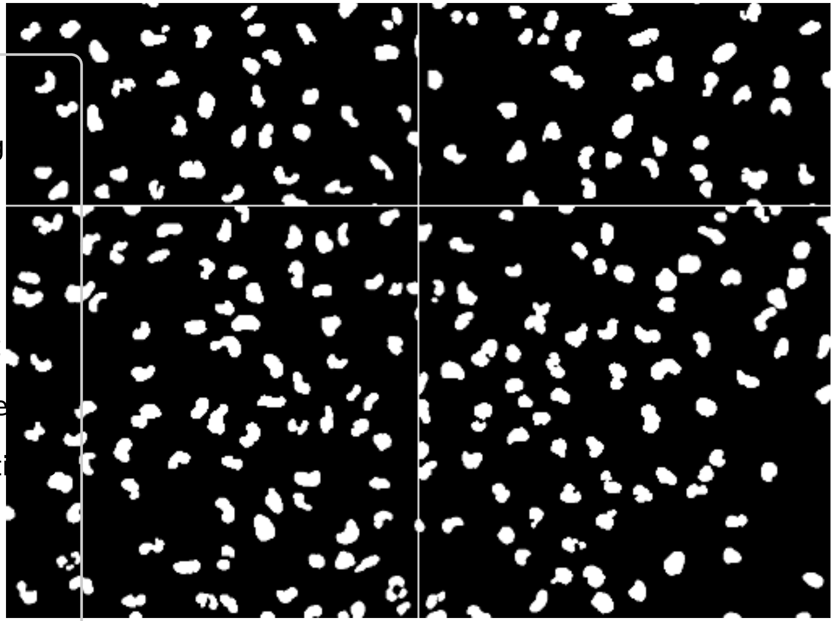




First, we segment the nuclei using `thresh`, `fillHull`, `bwlabel` and `opening`.

```
nmask = thresh(nuc, w=10, h=10, offset=0.05)
nmask = opening(nmask, makeBrush(5, shape='disc'))
nmask = fillHull(nmask)
nmask = bwlabel(nmask)

display(nmask, all=TRUE)
```





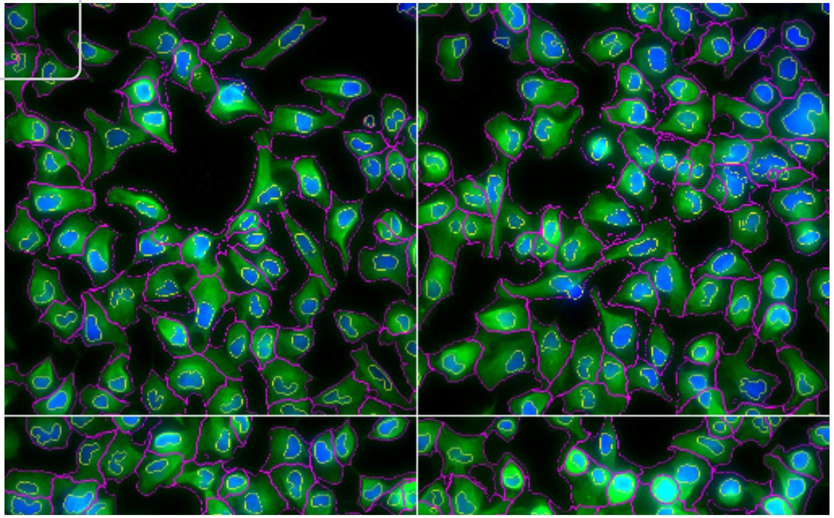
Next, we use the segmented nuclei as seeds in the Voronoi segmentation of the cytoplasm.

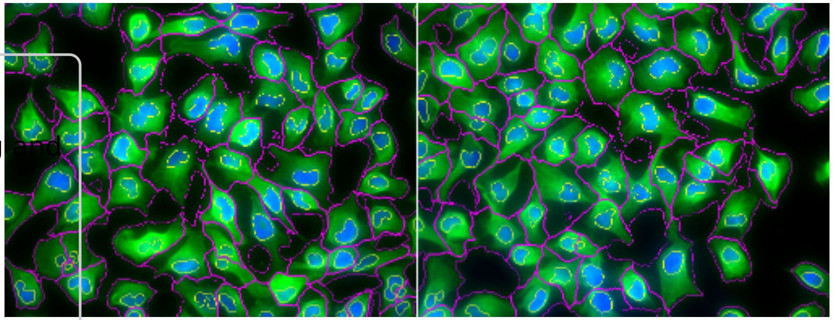
```
ctmask = opening(cel>0.1, makeBrush(5, shape='disc'))
cmask = propagate(cel, seeds=nmask, mask=ctmask)

display(ctmask)
```

```
Only the first frame of the image stack is displayed.
To display all frames use 'all = TRUE'.
```

To visualize our segmentation on the we use `paintObject`.

```
segmented = paintObjects(cmask, cells, col='#ff00ff')
segmented = paintObjects(nmask, segmented, col='#ffff00')

display(segmented, all=TRUE)
```





12 Session Info

```
sessionInfo()
```

```
R Under development (unstable) (2017-02-13 r72168)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 16.04.2 LTS

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods
base

other attached packages:
[1] EBImage_4.17.10  knitr_1.15.1     BiocStyle_2.3.30

loaded via a namespace (and not attached):
 [1] locfit_1.5-9.1      Rcpp_0.12.9         bookdown_0.3
 [4] lattice_0.20-34     fftwtools_0.9-7     png_0.1-7
 [7] digest_0.6.12       rprojroot_1.2       tiff_0.1-5
[10] grid_3.4.0          backports_1.0.5     magrittr_1.5
[13] evaluate_0.10       stringi_1.1.2       rmarkdown_1.3
[16] tools_3.4.0         stringr_1.1.0       jpeg_0.1-8
[19] abind_1.4-5         parallel_3.4.0      yaml_2.1.14
[22] compiler_3.4.0      BiocGenerics_0.21.3 htmltools_0.3.5
```