Introduction to *EBImage*

Andrzej Oles, Gregoire Pau, Oleg Sklyar, Wolfgang Huber
gpau@ebi.ac.uk

March 23, 2015

Contents

1 Reading, displaying and writing images

The package *EBImage* is loaded by the following command.

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

The function `readImage` is able to read images from files or URLs. Currently supported image formats are JPEG, PNG and TIFF[1].

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

Images can be displayed using the function `display`. Pixel intensities should range from 0 (black) to 1 (white).

```
> display(img)
```

Color images or images with multiple frames can also be read with `readImage`.

[1] See the RBioFormats package for support of a much wider range of formats.

Figure 1: `img`, `imgc`

```
> imgc = readImage(system.file("images", "sample-color.png", package="EBImage"))
> display(imgc)
> nuc = readImage(system.file('images', 'nuclei.tif', package='EBImage'))
> display(nuc)
```

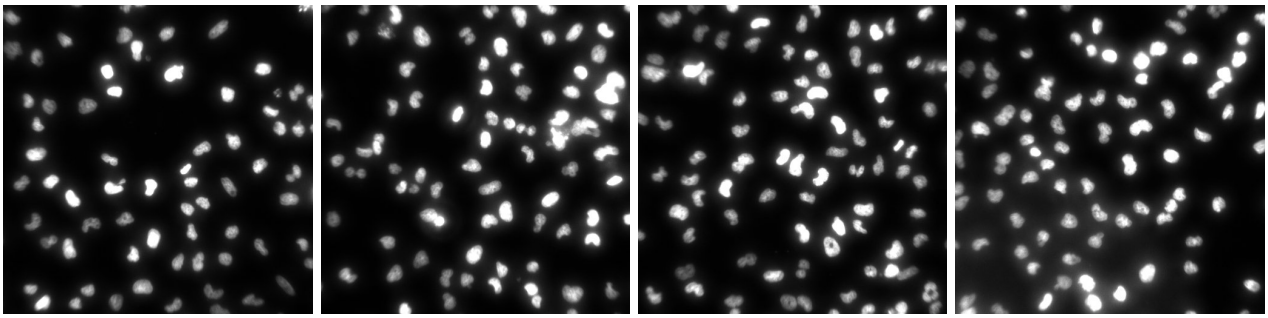

Figure 2: `nuc`

Images can be written with `writeImage`. The file format is deduced from the file name extension. The format need not be the same as the format of the file from which the image was read.

```
> writeImage(img,  'img.jpeg', quality=85)
> writeImage(imgc, 'imgc.jpeg', quality=85)
```

2 Image objects and matrices

The package *EBImage* uses the class `Image` to store and process images. Images are stored as multi-dimensional arrays containing the pixel intensities. All EBImage functions are also able to work with matrices and arrays.

```
> print(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
```

As matrices, images can be manipulated with all R mathematical operators. This includes + to control the brightness of an image, * to control the contrast of an image or ^ to control the gamma correction parameter.

```
> img1 = img+0.5
> img2 = 3*img
> img3 = (0.2+img)^3
```

Figure 3: `img`, `img1`, `img2`, `img3`

Others operators include [to crop images, < to threshold images or `t` to transpose images.

```
> img4 = img[299:376, 224:301]
> img5 = img>0.5
> img6 = t(img)
> print(median(img))
```

```
[1] 0.3843137
```

Images with multiple frames are created using `combine` which merges images.

```
> imgcomb = combine(img, img*2, img*3, img*4)
> display(imgcomb)
```

Figure 4: `img`, `img4`, `img5`, `img6`

Figure 5: `imgcomb`

3 Spatial transformations

Specific spatial image transformations are done with the functions `resize`, `rotate`, `translate` and the functions `flip` and `flop` to reflect images.

```
> img7 = rotate(img, 30)
> img8 = translate(img, c(40, 70))
> img9 = flip(img)
```

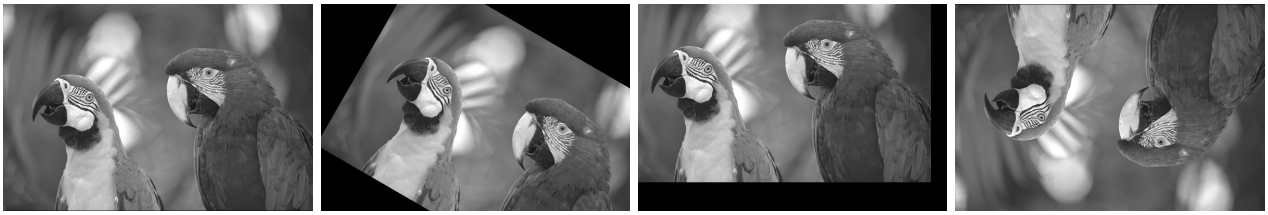

Figure 6: `img`, `img7`, `img8`, `img9`

4 Color management

The class `Image` extends the base class `array` and uses the `colormode` slot to store how the color information of the multi-dimensional data should be handled.

As an example, the color image `imgc` is a 512x512x3 array, with a `colormode` slot equals to `Color`. The object is understood as a color image by *EBImage* functions.

```
> print(imgc)
```

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

imageData(object)[1:5,1:6,1]
          [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
[1,] 0.4549020 0.4784314 0.4941176 0.5137255 0.5294118 0.5529412
[2,] 0.4588235 0.4784314 0.4941176 0.4980392 0.5215686 0.5490196
[3,] 0.4705882 0.4823529 0.4980392 0.5137255 0.5294118 0.5372549
[4,] 0.4666667 0.4745098 0.5058824 0.5137255 0.5450980 0.5647059
[5,] 0.4705882 0.4705882 0.4901961 0.5137255 0.5372549 0.5647059
```

The function `colorMode` can access and change the value of the slot `colormode`, modifying the rendering mode of an image. In the next example, the `Color` image `imgc` with one frame is changed into a `Grayscale` image with 3 frames, corresponding to the red, green and blue channels. The function `colorMode` does not change the content of the image but changes only the way the image is rendered by *EBImage*.

```
> colorMode(imgc) = Grayscale
> display(imgc)
```

Figure 7: `imgc`, rendered as a `Color` image and as a `Grayscale` image with 3 frames (red channel, green channel, blue channel)

The color mode of image `imgc` is reverted back to `Color`.

```
> colorMode(imgc) = Color
```

The function `channel` performs colorspace conversion and can convert `Grayscale` images into `Color` ones both ways and can extract color channels from `Color` images. Unlike `colorMode`, `channel` changes the pixel intensity values of the image. The function `rgbImage` is able to combine 3 `Grayscale` images into a `Color` one.

```
> imgk = channel(img, 'rgb')
> imgk[236:276, 106:146, 1] = 1
> imgk[236:276, 156:196, 2] = 1
> imgk[236:276, 206:246, 3] = 1
> imgb = rgbImage(red=img, green=flip(img), blue=flop(img))
```

Figure 8: `imgk`, `imgb`

5 Image filtering

Images can be linearly filtered using `filter2`. `filter2` convolves the image with a matrix filter. Linear filtering is useful to perform low-pass filtering (to blur images, remove noise, ...) and high-pass filtering (to detect edges, sharpen images, ...). Various filter shapes can be generated using `makeBrush`.

```
> flo = makeBrush(21, shape='disc', step=FALSE)^2
> flo = flo/sum(flo)
> imgflo = filter2(imgc, flo)
> fhi =  matrix(1, nc=3, nr=3)
> fhi[2,2] = -8
> imgfhi = filter2(imgc, fhi)
```

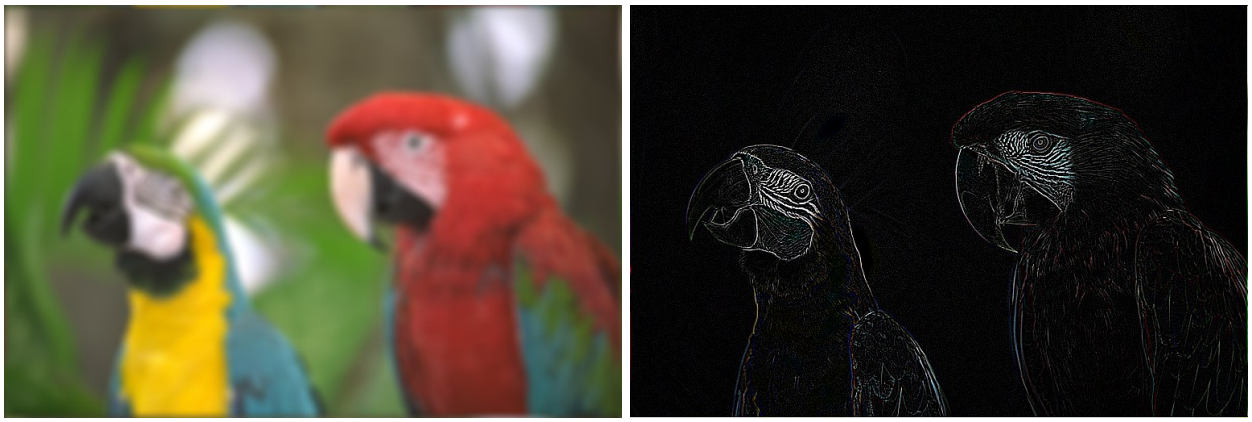

Figure 9: Low-pass filtered `imgflo` and high-pass filtered `imgfhi`

6 Morphological operations

Binary images are images where the pixels of value 0 constitute the background and the other ones constitute the foreground. These images are subject to several non-linear mathematical operators called morphological operators, able to `erode` and `dilate` an image.

```
> ei = readImage(system.file('images', 'shapes.png', package='EBImage'))
> ei = ei[110:512,1:130]
> display(ei)
> kern = makeBrush(5, shape='diamond')
> eierode = erode(ei, kern)
> eidilat = dilate(ei, kern)
```

Figure 10: `ei` ; `eierode` ; `eidilat`

7 Segmentation

Segmentation consists in extracting objects from an image. The function `bwlabel` is a simple function able to extract every connected sets of pixels from an image and relabel these sets with a unique increasing integer. `bwlabel` can be used on binary images and is useful after thresholding.

```
> eilabel = bwlabel(ei)
> cat('Number of objects=', max(eilabel),'\n')
```

```
Number of objects= 7
```

```
> nuct = nuc[,,1]>0.2
> nuclabel = bwlabel(nuct)
> cat('Number of nuclei=', max(nuclabel),'\n')
```

```
Number of nuclei= 74
```

Figure 11: `ei`, `eilabel/max(eilabel)`

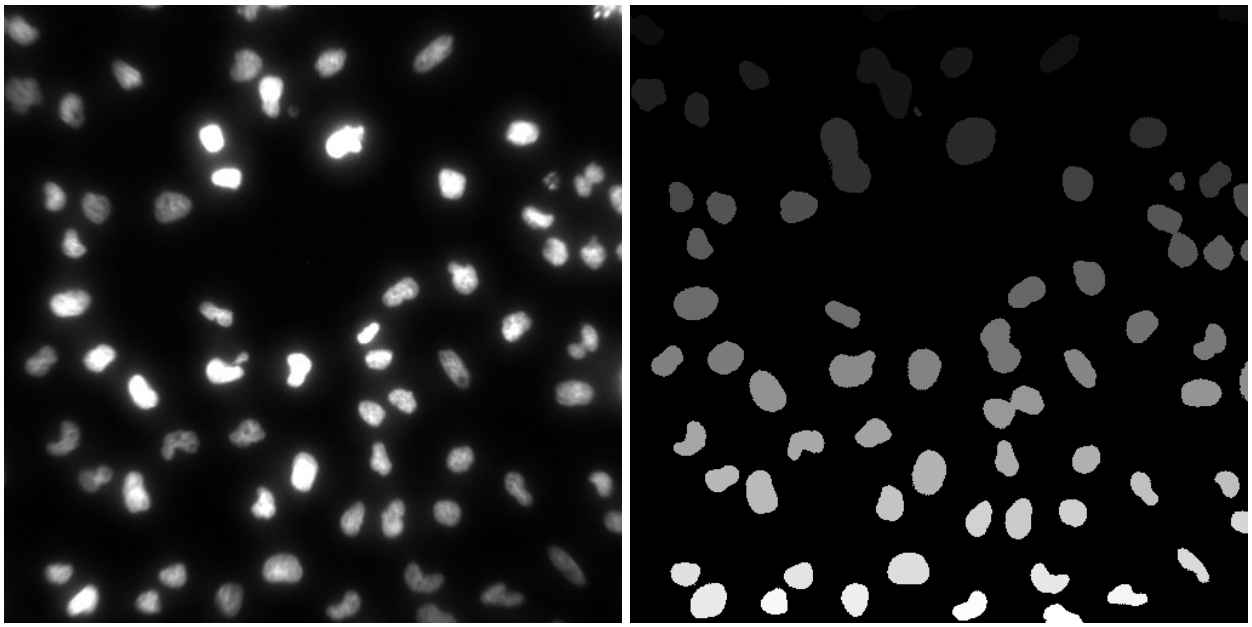

Figure 12: `nuc[ , ,1]`, `nuclabel/max(nuclabel)`

Since the images `eilabel` and `nuclabel` range from 0 to the number of object they contain (given by `max(eilabel)` and `max(nucabel)`), they have to be divided by these number before displaying, in order to fit the [0,1] range needed by `display`.

The grayscale top-bottom gradient observable in `eilabel` and `nuclabel` is due to the way `bwlabel` labels the connected sets, from top-left to bottom-right.

Adaptive thresholding consists in comparing the intensity of pixels with their neighbors, where the neighborhood is specified by a filter matrix. The function `thresh` performs a fast adaptive thresholding of an image with a rectangular window while the combination of `filter2` and < allows a finer control. Adaptive thresholding allows a better segmentation when objects are close together.

```
> nuct2 =  thresh(nuc[,,1], w=10, h=10, offset=0.05)
> kern = makeBrush(5, shape='disc')
> nuct2 = dilate(erode(nuct2, kern), kern)
> nuclabel2 = bwlabel(nuct2)
> cat('Number of nuclei=', max(nuclabel2),'\n')
```

```
Number of nuclei= 76
```

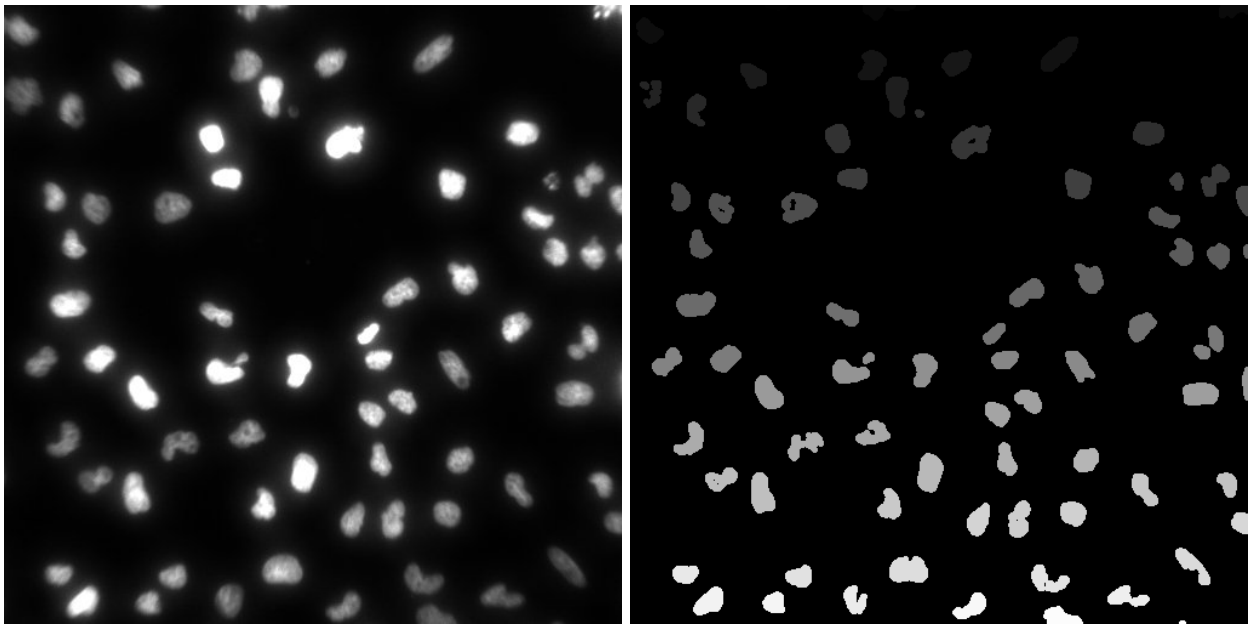

Figure 13: `nuc[ , ,1]`, `nuclabel2/max(nuclabel)`

8 Object manipulation

Objects, defined as sets of pixels with the same unique integer value can be outlined and painted using `paintObjects`. Some holes are present in objects of `nuclabel2` which can be filled using `fillHull`.

```
> nucgray = channel(nuc[,,1], 'rgb')
> nuch1 = paintObjects(nuclabel2, nucgray, col='#ff00ff')
> nuclabel3 = fillHull(nuclabel2)
> nuch2 = paintObjects(nuclabel3, nucgray, col='#ff00ff')
```

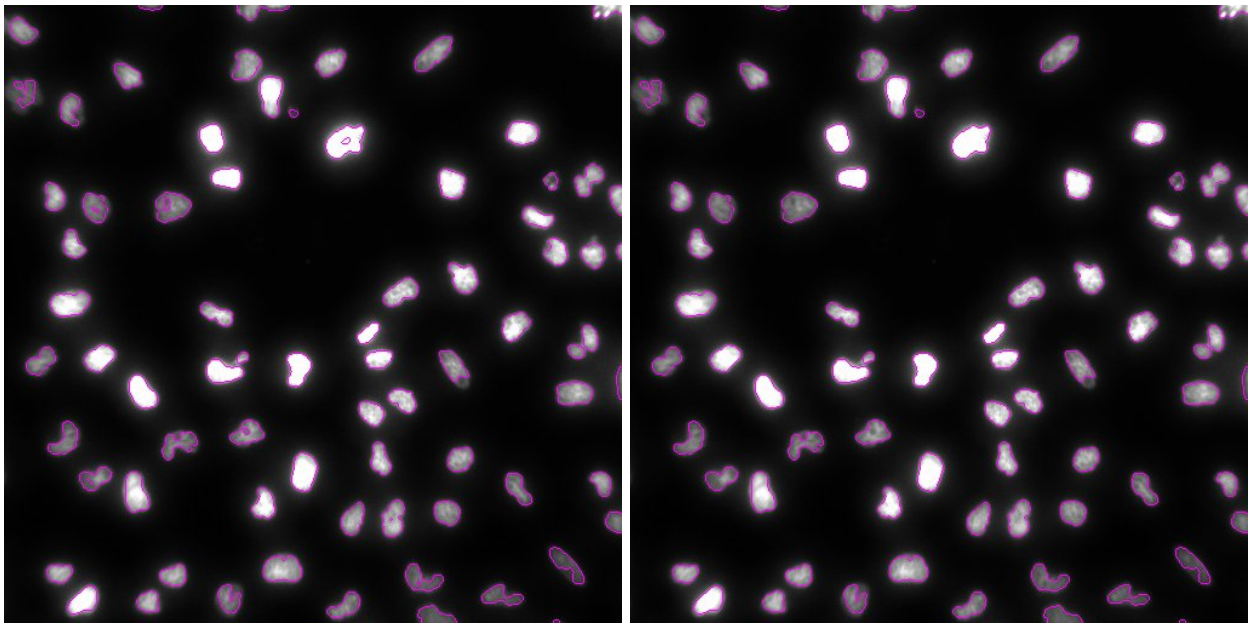

Figure 14: `nuch1`, `nuch2`

A broad variety of objects features (basic, image moments, shape, Haralick features) can be computed using `computeFeatures`. In particular, object coordinates are computed with the function `computeFeatures.moment`.

```
> xy = computeFeatures.moment(nuclabel3)[, c("m.cx", "m.cy")]
> xy[1:4,]
```

```
       m.cx      m.cy
1 121.74667  2.466667
2 210.19231  4.611888
3 497.44550  5.165877
4  15.99688 22.140187
```

9 Cell segmentation example

This is a complete example of segmentation of cells (nucleus + cell bodies) using the functions described before and the function `propagate`, able to perform Voronoi-based region segmentation.

Images of nuclei and cell bodies are first loaded:

```
> nuc = readImage(system.file('images', 'nuclei.tif', package='EBImage'))
> cel = readImage(system.file('images', 'cells.tif', package='EBImage'))
> img = rgbImage(green=1.5*cel, blue=nuc)
```

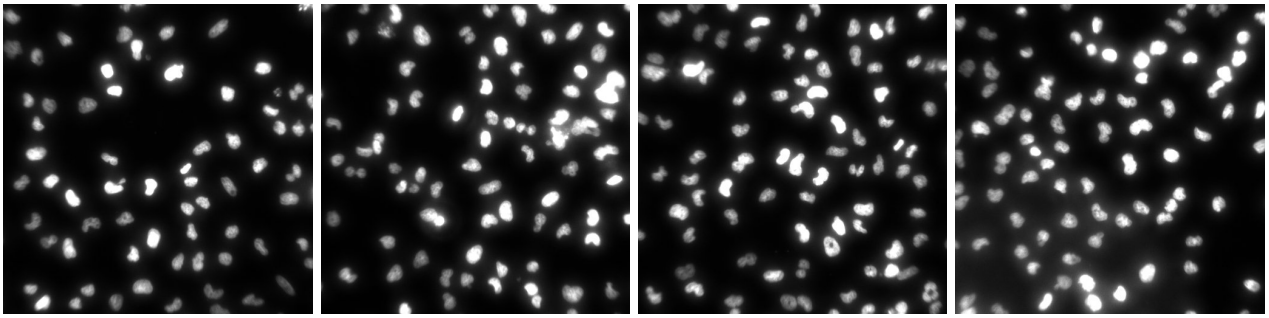

Figure 15: `nuc`

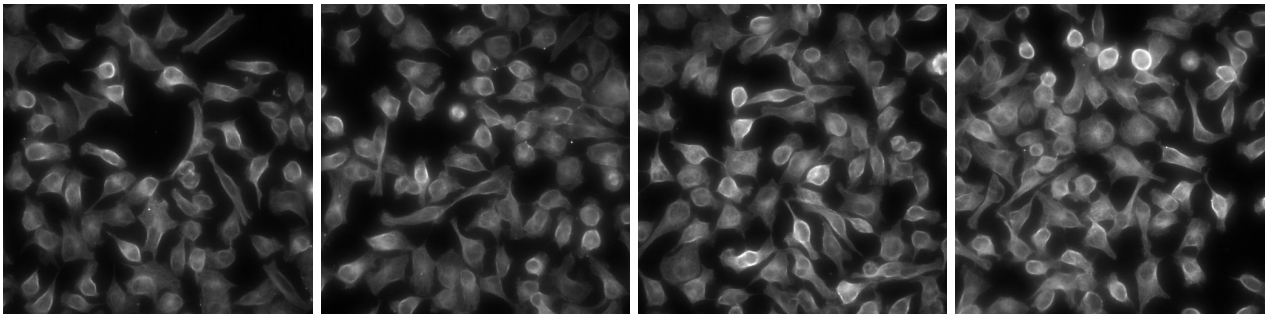

Figure 16: `cel`

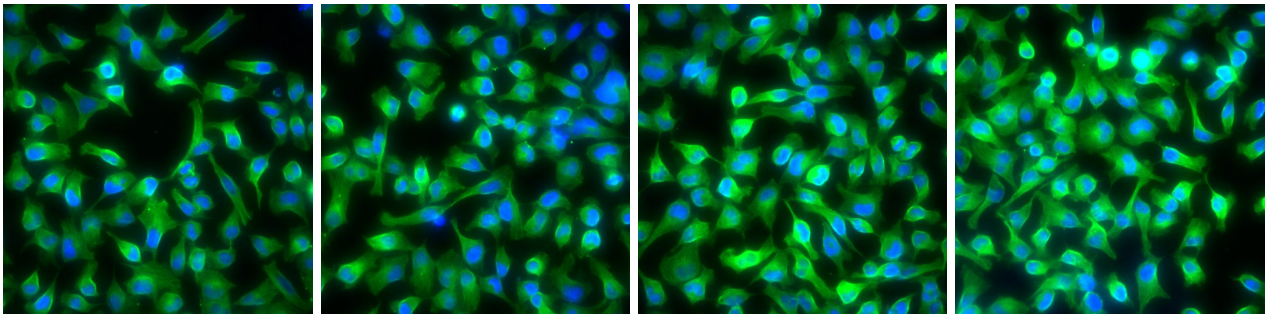

Figure 17: `img`

Nuclei are first segmented using `thresh`, `fillHull`, `bwlabel` and `opening`, which is an `erosion` followed by a `dilatation`.

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

Cell bodies are segmented using `propagate`.

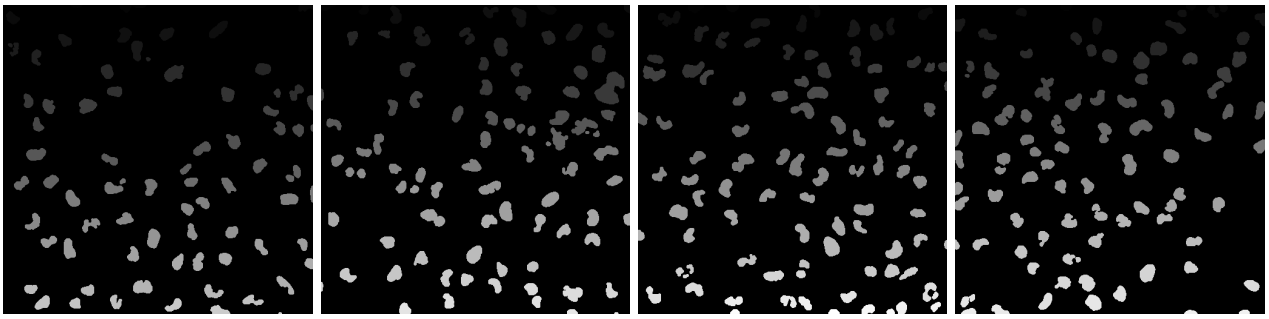

Figure 18: nmask/max(nmask)

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

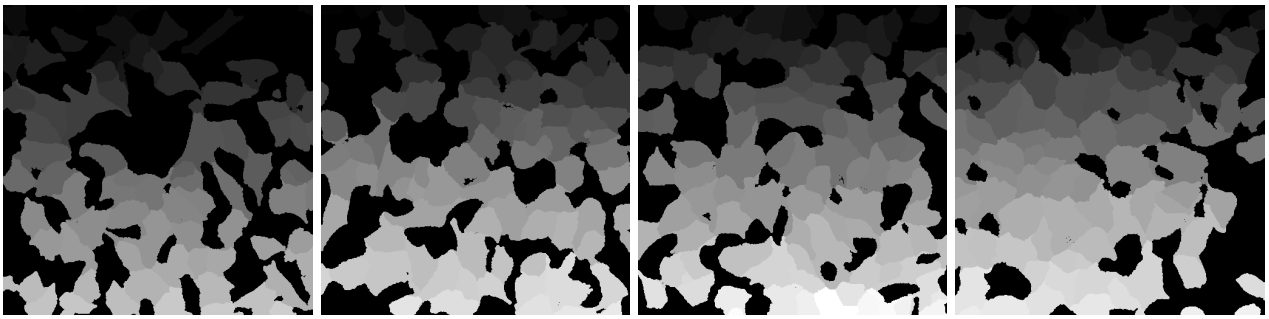

Figure 19: cmask/max(cmask)

Cells are outlined using paintObjects.

```
> res = paintObjects(cmask, img, col='#ff00ff')
> res = paintObjects(nmask, res, col='#ffff00')
```

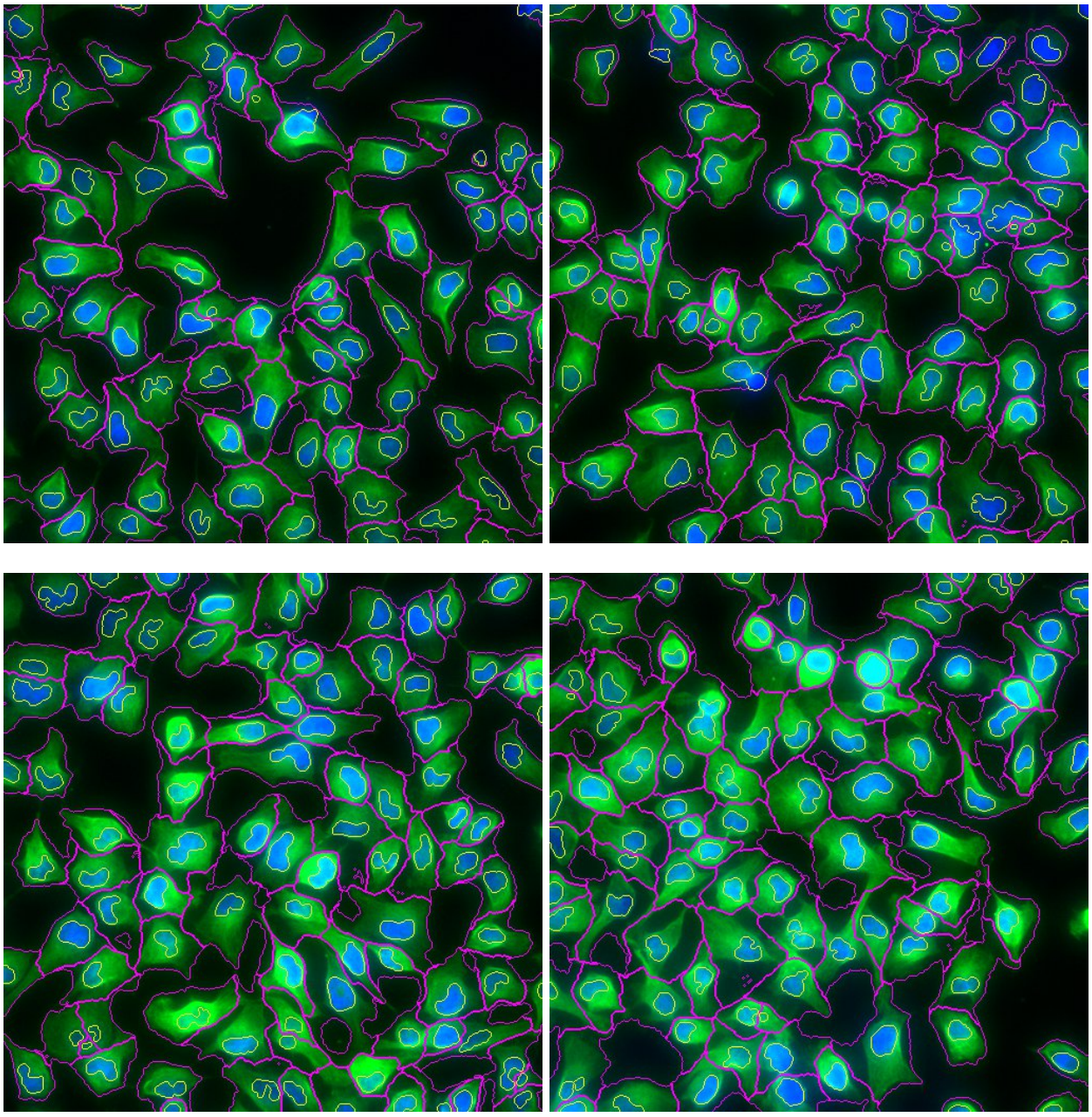

Figure 20: Final segmentation `res`