Modern Statistics for Modern Biology

Susan Holmes, Wolfgang Huber, Trevor Martin

June 27, 2014

Contents

Chapter 13

Image data

Images are a rich source of data. In this chapter, we will study examples of how data can be extracted from images, and how we can use statistical methods to summarize and understand the data. The goal of this chapter is to show that getting started working with image data is easy – if you are able to handle the basic R environment, you are ready to start working with images. That said, this chapter is not a general introduction to image analysis. The field is extensive, it touches many areas of advanced mathematics, engineering and computer science, and there are many excellent books about it.

We will study two-dimensional images, and in a case study that makes up the main part of the chapter we will look at images of cells, learn how to identify their positions in the images, and how to quantitatively measure characteristics of the objects that we detected, like their size, shape, color or texture. Such information can then be used for down-stream analyses: for instance, we could compare cells between different conditions, say under the effect of different drugs, or in different stages of differentiation and growth; or we could measure how the objects in the image relate to each other, e. g., whether they like to cluster or rather repel each other, or whether certain characteristics tend to be more shared between neighboring objects, indicative of cell-cell communication. In the language of genetics, what this means is that we can use images as complex phenotypes or as multivariate quantitative traits.

We will not touch here many important areas of image analysis in more than two dimensions: we do not consider 3D segmentation and registration, nor temporal tracking. These are sophisticated tasks for which specialized software would likely perform better than what we could assemble in the scope of this chapter.

There are similarities between data from high-throughput imaging and other high-throughput data in genomics. Batch effects tend to play a role, for instance because of changes in staining efficiency, lamp aging or many other factors. Experimental design and analysis need to make the appropriate precautions. In principle, the intensity values in an image can be calibrated in physical units, corresponding, say to radiant energy or fluorophore concentration; however this is not always done in practice. Somewhat easier to achieve and clearly valuable is a calibration of the spatial dimensions of the image, i. e., the conversion factor between pixel distance and metric distance.

13.1 Goals for this chapter

- Learn how to read and write images into and out of R.
- Learn how to manipulate images in R.
- Understand how to apply filters and transformations to images.
- Combine these skills to do segmentation and feature extraction; we will use cell segmentation as an example.

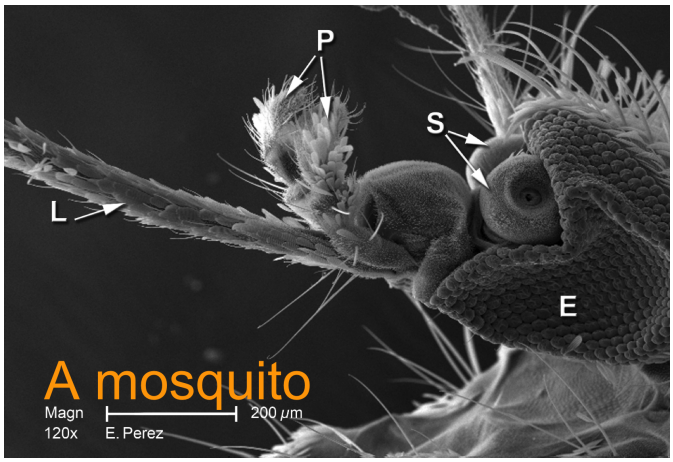

Figure 13.1: "Mosquito discovered deceased in the suburbs of Decatur, Georgia (credit: CDC / Janice Haney Carr).

13.2 Loading images

A useful toolkit for handling images in R is the Bioconductor package *EBImage* [7]. We start out by reading in a simple picture to demonstrate the basic functions.

```
library("EBImage")
imagefile = system.file("images", "mosquito.png", package = "MSBdata")
mosq = readImage(imagefile)
```

EBImage currently supports three image file formats: `jpeg`, `png` and `tiff`. Above, we loaded a sample image from the *MSBdata* package. When you are working with your own data, you do not need that package, just provide the name(s) of your file(s) to the `readImage` function. As you will see later in this chapter, `readImage` can read multiple images in one go, which are then all assembled into a single image data object.

13.3 Visualizing

Now we want to visualize the image that we just read in. The basic function to do so is `display`.

```
display(mosq)
```

The above command opens the image in a window of your web browser (as set by `getOption("browser")`). Using the mouse or keyboard shortcuts, you can zoom in and out of the image, pan and cycle through multiple image frames.

Alternatively, we can also display the image using R's built-in plotting by calling `display` with the argument `method = "raster"`. The image then goes to the current device. In this way, we can combine image data with other plotting functionality, for instance, to add text labels.

```
display(mosq, method = "raster")
text(x = 85, y = 800, label = "A mosquito", adj = 0, col = "orange", cex = 1.5)
```

The resulting plot is shown in Figure 13.1. As usual, the graphics displayed in an R device can be saved using the *base* R functions `dev.print` or `dev.copy`.

Note that we can also read and view color images, see Figure 13.2.

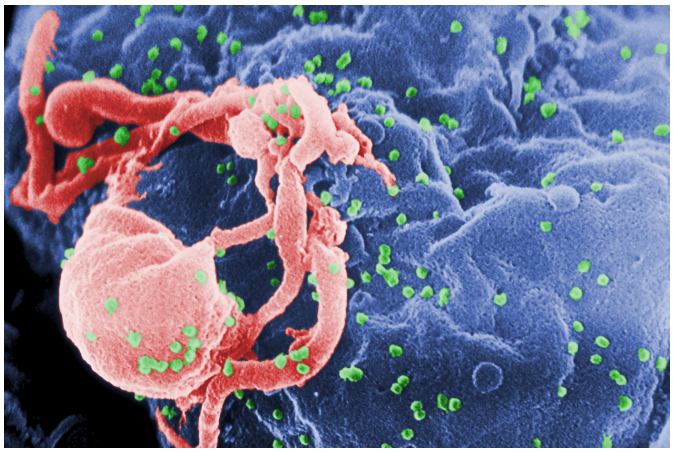

Figure 13.2: Scanning electron micrograph of HIV-1 virions budding from a cultured lymphocyte (credit: CDC / C. Goldsmith, P. Feorino, E.L. Palmer, W.R. McManus).

```
imagefile = system.file("images", "hiv.png", package = "MSBdata")
hivc = readImage(imagefile)
display(hivc)
```

Furthermore, if an image has multiple frames, they can be displayed all at once in a grid arrangement by specifying the function argument `all = TRUE` (Figure 13.3),

```
nuc = readImage(system.file("images", "nuclei.tif", package = "EBImage"))
display(1 - nuc, method = "raster", all = TRUE)
```

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

```
display(nuc, method = "raster", frame = 2)
```

13.4 Writing images to file

EBImage also allows us to save images to file. The image `hiv.png` that we read in was a `png` file, so now we will write out the same image as a `jpeg` file. The `jpeg` format allows setting 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 stronger compression at the cost of some reduction in image quality.

```
writeImage(hivc, "hivc.jpeg", quality = 85)
```

Similarly, we could have written the image out as a `tiff` file and set which compression algorithm we want to use (see `?writeImage`).

13.5 How are images stored in R?

Let's dig into what's going on by first identifying the class of the images we are plotting:

```
class(mosq)
```

```
## [1] "Image"
```

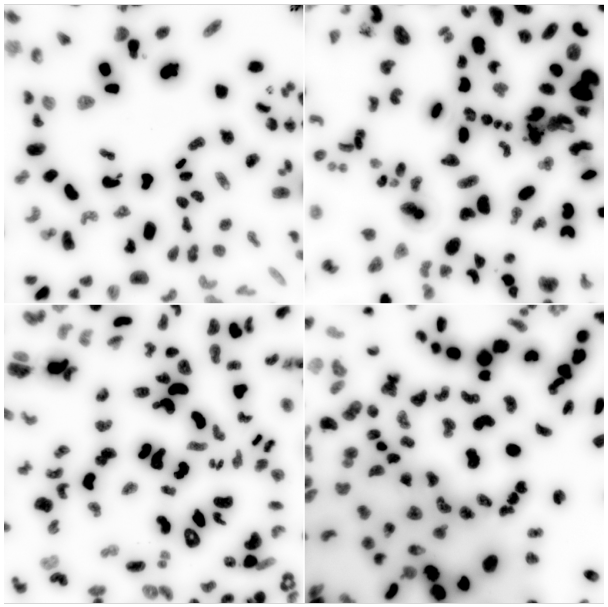

Figure 13.3: Tiled display of four images of cell nuclei from the *EBImage* package.

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

So we see that we are using a package-specific class called *Image*. We can find out more about this class by typing `class ?Image`. It is derived from the R base class *array*. Let's peek into the internal structure of the object.

```
str(mosq)
```

```
## Formal class 'Image' [package "EBImage"] with 2 slots
##   ..@ .Data    : num [1:1400, 1:952] 0.196 0.196 0.196 0.196 0.196 ...
##   ..@ colormode: int 0
```

The first slot, `.Data`, holds a numeric array with the pixel intensities. We can see that the array is two-dimensional, with 1400 times 952 elements. Note that these dimensions can be extracted using the `dim` function, just like for regular arrays.

```
dim(mosq)
```

```
## [1] 1400  952
```

We can use the method `hist` to plot a histogram of the numeric pixel intensities.

```
hist(mosq)
```

The histogram in Figure 13.4 indicates that the range of the intensities is between 0 and 1.

If we want to directly access the data matrix as an R *array*, we can use the accessor function `imageData`.

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

```
##       [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
```

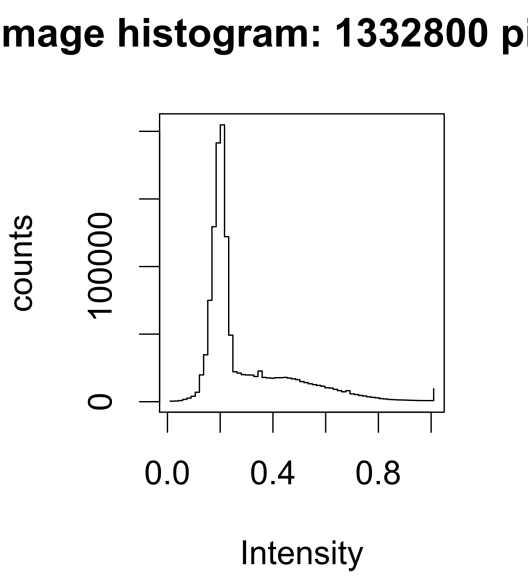

Figure 13.4: Histogram of the pixel intensities in `mosq`.

```
## [1,] 0.196 0.196 0.196 0.196 0.196 0.196
## [2,] 0.196 0.196 0.196 0.196 0.196 0.196
## [3,] 0.196 0.196 0.200 0.204 0.200 0.196
```

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

```
mosq
```

```
## Image
##   colormode: Grayscale
##   storage.mode: double
##   dim: 1400 952
##   nb.total.frames: 1
##   nb.render.frames: 1
##
## imageData(object)[1:5,1:6]:
##       [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
## [1,] 0.196 0.196 0.196 0.196 0.196 0.196
## [2,] 0.196 0.196 0.196 0.196 0.196 0.196
## [3,] 0.196 0.196 0.200 0.204 0.200 0.196
## [4,] 0.196 0.196 0.204 0.208 0.200 0.196
## [5,] 0.196 0.200 0.212 0.216 0.200 0.192
```

Now let us look at the color image.

```
hivc
```

```
## Image
##   colormode: Color
##   storage.mode: double
##   dim: 1400 930 3
```

```
##   nb.total.frames: 3
##   nb.render.frames: 1
##
## imageData(object)[1:5,1:6,1]:
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    0    0    0    0    0    0
## [2,]    0    0    0    0    0    0
## [3,]    0    0    0    0    0    0
## [4,]    0    0    0    0    0    0
## [5,]    0    0    0    0    0    0
```

The two images differ by their property `colorMode`, which is `Grayscale` for `mosq` and `Color` for `hivc`. What is the point of this property? It turns out to be convenient when we are dealing with stacks of images. If `colorMode` is `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. In `hivc`, there are three color channels, which correspond to the red, green and blue intensities of our photograph. However, this does not necessarily need to be the case, the number of color channels can be any number ≥ 1.

Finally, if we look at our cell data,

```
nuc
```

```
## Image
##   colormode: Grayscale
##   storage.mode: double
##   dim: 510 510 4
##   nb.total.frames: 4
##   nb.render.frames: 4
##
## imageData(object)[1:5,1:6,1]:
##        [,1]   [,2]   [,3]   [,4]   [,5]   [,6]
## [1,] 0.0627 0.0745 0.0706 0.0824 0.1059 0.0980
## [2,] 0.0627 0.0588 0.0784 0.0902 0.0902 0.1059
## [3,] 0.0667 0.0667 0.0824 0.0784 0.0941 0.0941
## [4,] 0.0667 0.0667 0.0706 0.0863 0.0863 0.0980
## [5,] 0.0588 0.0667 0.0706 0.0824 0.0941 0.1059
```

```
dim(imageData(nuc))
```

```
## [1] 510 510   4
```

we see that we have 4 total frames that correspond to the 4 separate images (`nb.render.frames`).

13.6 Manipulating images

Now that we know that images are stored as arrays of numbers in R, our method of manipulating images becomes clear – simple algebra!

For example, we can take our original image, shown again in the top left of Figure 13.5, and flip the bright areas to dark and vice versa by simply subtracting the image from its maximum value, 1..

```
mosqinv = 1 - mosq
```

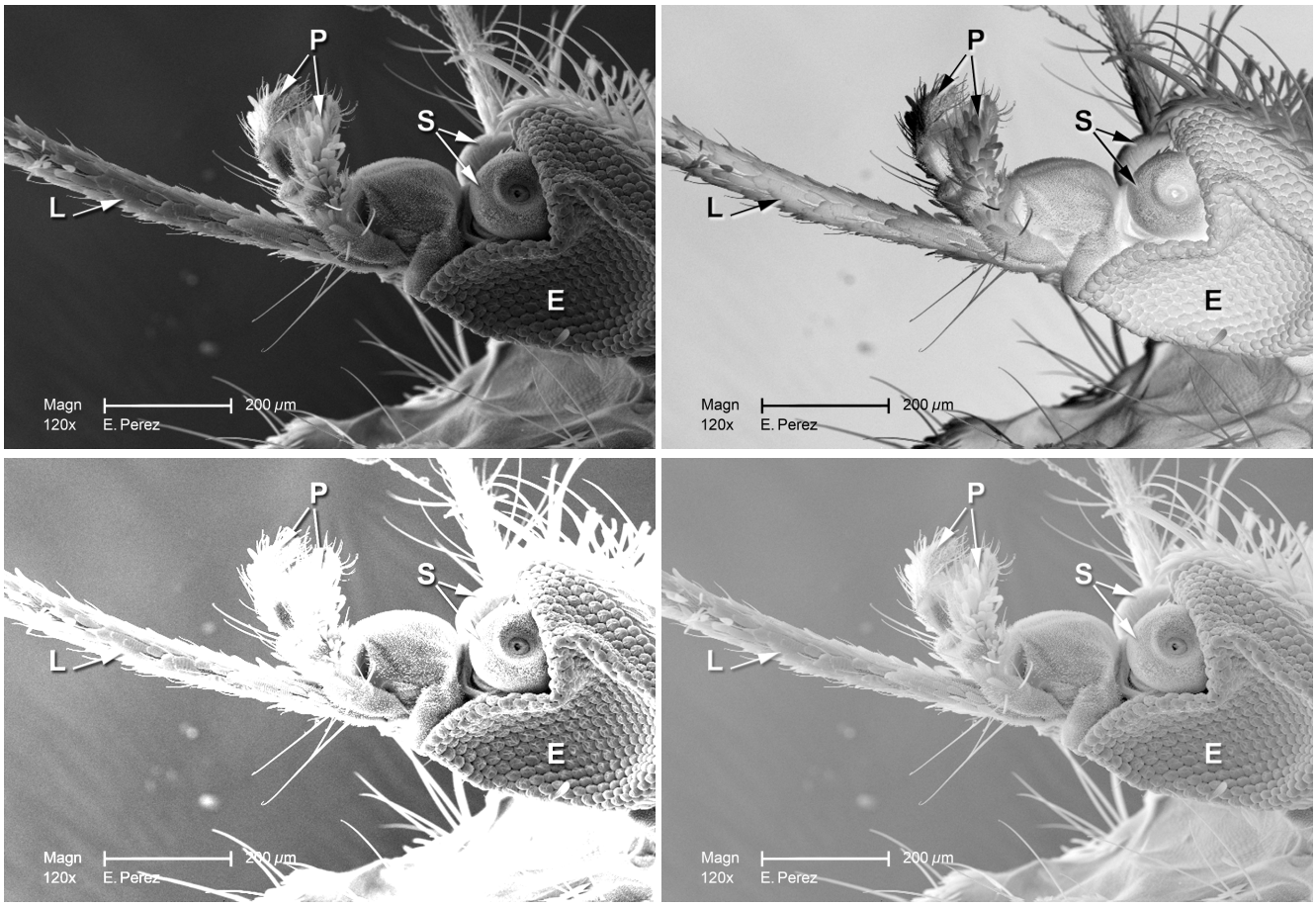

Figure 13.5: The original mosquito image (top left) and three different image transformations (subtraction, multiplication, power transformation).

```
display(mosqinv)
```

The resulting image is displayed in the top right of Figure 13.5. We could also adjust the contrast through multiplication (Figure 13.5, bottom left) and the gamma-factor through exponentiation (Figure 13.5, bottom right).

```
mosqcont = mosq * 3
mosqexp = mosq ^ (1/3)
```

Furthermore, we can crop, threshold, and transpose images with matrix operations (see Figure 13.6).

```
mosqcrop   = mosq[100:438, 112:550]
mosqthresh = mosq > 0.5
mosqtransp = transpose(mosq)
```

The thresholding operation returns an *Image* object whose pixels are binary values. The R data type used for these is *logical*.

```
mosqthresh
```

```
## Image
##   colormode: Grayscale
##   storage.mode: logical
##   dim: 1400 952
```

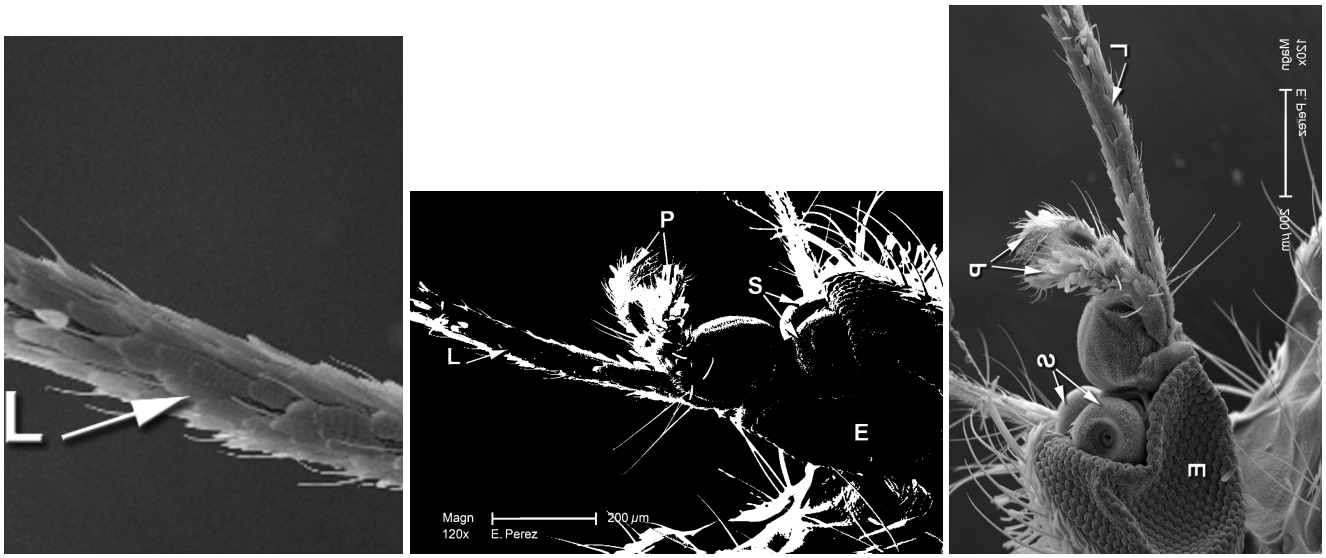

Figure 13.6: Three more image transformations: cropping, thresholding, transposition.

```
##   nb.total.frames: 1
##   nb.render.frames: 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
```

In principle, we could also have used R's *base* function `t` for transposition, however, the method `transpose` is preferable since it also works with color and multiframe images.

13.7 Spatial transformations

We just saw one type of spatial transformation, transposition, but there are many more – here are some examples:

```
mosqrot   = rotate(mosq, angle = 30)
mosqshift = translate(mosq, v = c(40, 70))
mosqflip  = flip(mosq)
mosqflop  = flop(mosq)
```

`rotate` rotates the image clockwise with the given angle, `translate` moves the image by the specified two-dimensional vector (pixels that end up outside the image region are cropped, and pixels that enter into the image region are set to zero). The functions `flip` and `flop` reflect the image around the central horizontal and vertical axis, respectively. The results of these operations are shown in Figure 13.7.

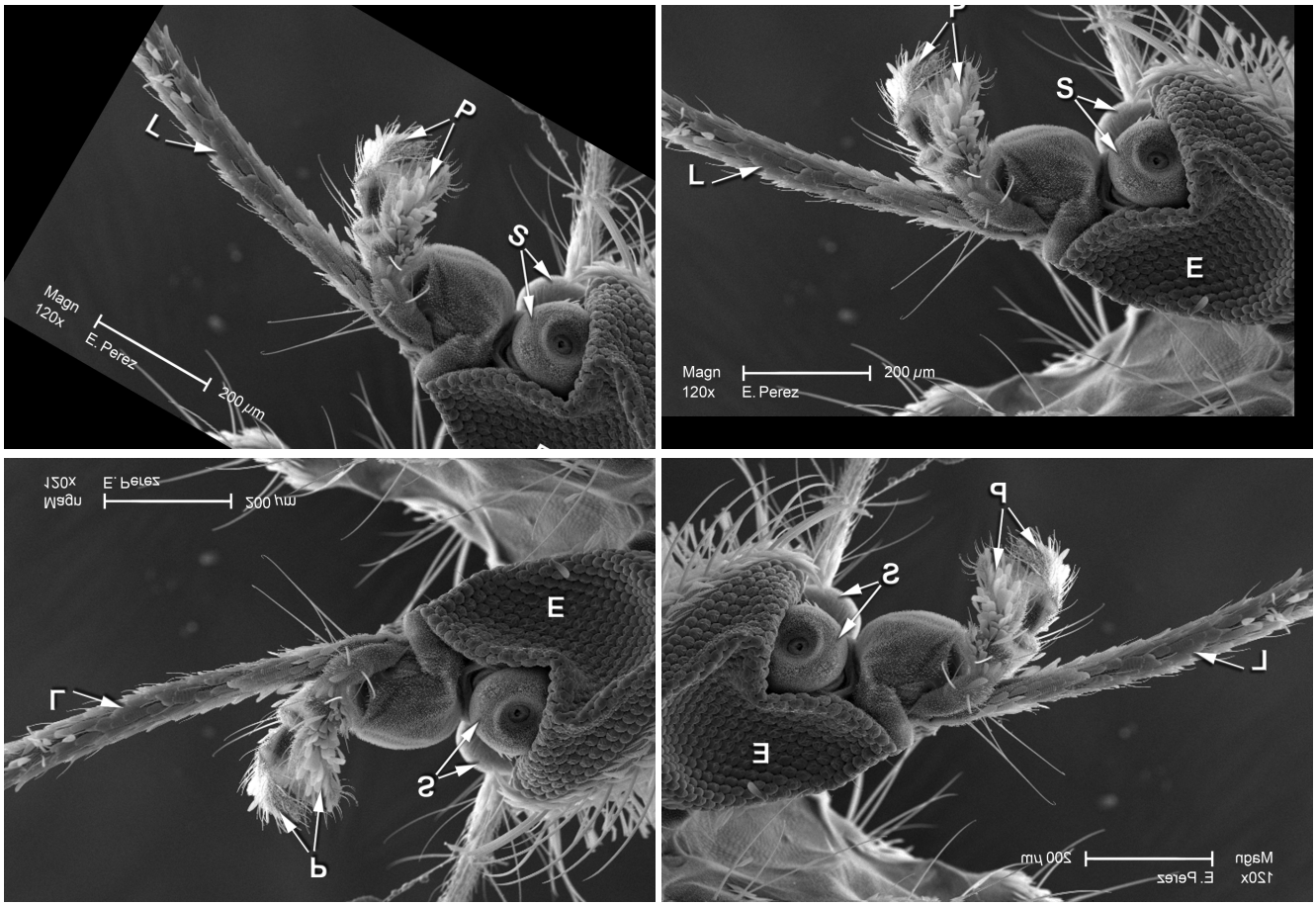

Figure 13.7: Spatial transformations: rotation (top left), translation (top right), reflection about the central horizontal axis (`flip`, bottom left), reflection about the central vertical axis (`flop`, bottom rightt).

13.8 Linear filters

Let's now switch to an application in cell biology. We load images of human cancer cells that were studied by Laufer, Fischer and co-workers [6]. They are shown in Figure 13.8.

```
imagefiles = system.file("images", c("image-DAPI.tif", "image-FITC.tif", "image-Cy3.tif"),
                         package="MSBdata")
cells = readImage(imagefiles)
```

The *Image* object `cells` is now a three-dimensional array of size 340 × 490 × 3, where the last dimension indicates that there are three individual grayscale frames. Our goal now is to computationally identify and quantitatively characterize the cells in these images. That by itself may seem a modest goal, but we can note that the dataset of Laufer et al. contains over 690,000 images, each of which has 2,048 × 2,048 pixels. Here, we are looking at three of these, out of which a small region was cropped. Once we know how to achieve our stated goal, we may apply our abilities to such large image collections, and that need no longer be a modest aim!

However, before we can start with real work, we need to deal with a slightly mundane data conversion issue. This is, of course, not unusual. Let us inspect the dynamic range (the minimum and the maximum value) of the images.

```
apply(cells, 3, range)
```

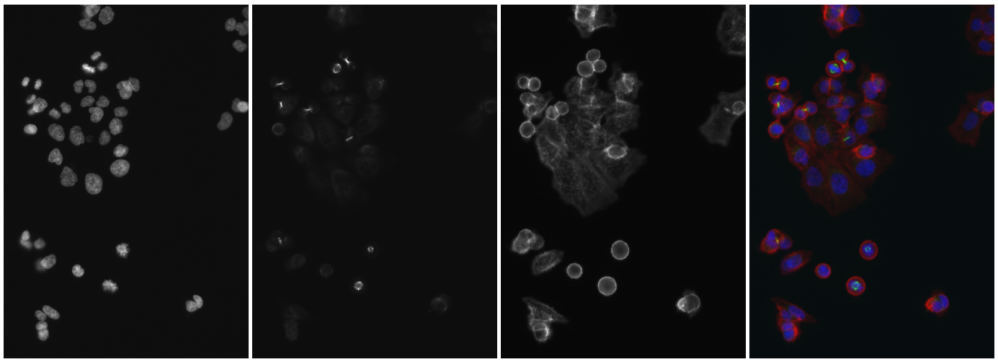

Figure 13.8: Human colon cancer cells (HCT116). The four images show the same cells: the leftmost image corresponds to DAPI staining of the cells' DNA, the second to immunostaining against α-tubulin, the third to actin. They are displayed as gray-scale images. The rightmost image is obtained by overlaying the three images as color channels of an RGB image (red: actin, green: α-tubulin, blue: DNA) [6].

```
##         [,1]   [,2]    [,3]
## [1,] 0.00159 0.0029 0.00166
## [2,] 0.03120 0.0625 0.05571
```

We see that the maximum values are small numbers well below 1. The reason for this is that the `readImage` function recognizes that the `TIFF` images uses 16 bit integers to represent each pixel, and it returns the data – as is common for numeric variables in R – in an array of double precision floating point numbers, with the integer values (whose theoretical range is from 0 to $2^{16} - 1 = 65535$) stored in the mantissa of the floating point representation and the exponents chosen so that the theoretical range is mapped to the interval $[0, 1]$. However, the scanner that was used to create these images only used the lower 11 or 12 bits, and this explains the small maximum values in the images. We can rescale the data to cover the range $[0, 1]$ as follows.

```
cells[,,1] = 32 * cells[,,1]
cells[,,2:3] = 16 * cells[,,2:3]
apply(cells, 3, range)
```

```
##        [,1]   [,2]   [,3]
## [1,] 0.0508 0.0464 0.0266
## [2,] 0.9986 0.9998 0.8914
```

We can keep in mind that these multiplications by powers of 2 have no impact on the underlying precision of the stored data. We also observe that the image representation in *EBImage* is not memory-efficient, as it uses 8 bytes per pixel, even if the real precision of the input data is much lower (in our case, 11 or 12 bits). However, for many biological images, their memory footprint is still small compared to the memory in modern computers, and the benefit is that we can use ordinary R arithmetic.

Now we are ready to get going with analyzing the images. As our first goal is segmentation of the images to identify the individual cells, we can start by removing local artifacts or noise from the images through smoothing. An intuitive approach is to define a window of a selected size around each pixel and average the values within that window. After applying this procedure to all pixels, the new, smoothed image is obtained. Mathematically, we can express this as

$$f'(x, y) = \frac{1}{N} \sum_{s=-a}^{a} \sum_{t=-a}^{a} f(x + s, y + t), \tag{13.1}$$

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

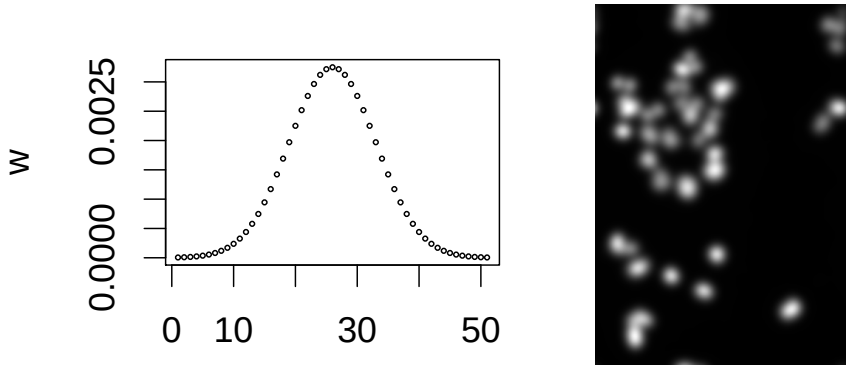

Figure 13.9: Left: the middle row of the weight matrix, `w[26, ]`. Right: `nucSmooth`, a smoothed version of the DNA channel in the image object `cells` (the original version is shown in the leftmost panel of Figure 13.8).

typically has highest weight at the window midpoint ($s = t = 0$) and then decreases towards the edges.

$$(w * f)(x, y) = \sum_{s=-\infty}^{+\infty} \sum_{t=-\infty}^{+\infty} w(s, t)\, f(x + s, y + s) \tag{13.2}$$

For notational convenience, we let the summations range from $-\infty$ to ∞, even if in practice the sums are finite since 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 $*$. In *EBImage*, the 2-dimensional convolution is implemented by the function `filter2`, and the auxiliary function `makeBrush` can be used to generate the weight function w.

```
w = makeBrush(size = 51, shape = "gaussian", sigma = 7)
nucSmooth = filter2(getFrame(cells, 1), w)
plot(w[(nrow(w)+1)/2, ], cex = 0.3, ylab = "w", xlab = "")
```

The result is shown in Figure 13.9. In fact, the `filter2` function does not directly perform the summation indicated in Equation(13.2). Instead, it uses the Fast Fourier Transformation [11], in a way that is mathematically equivalent and computationally more efficient.

The convolution in Equation(13.2) is a *linear* operation, in the sense that $w * (c_1 f_1 + c_2 f_2) = c_1 w * f_1 + c_2 w * f_2$ for any two images f_1, f_2 and numbers c_1, c_2. There is beautiful and powerful theory underlying linear filters [11].

To proceed, we now, however, use smaller smoothing bandwidths than what we displayed in Figure 13.9 for demonstration. Let's use a `sigma` of 1 pixel for the DNA channel and 3 pixels for actin and tubulin.

```
cellsSmooth = Image(dim = dim(cells))
sigma = c(1, 3, 3)
for(i in seq_along(sigma))
  cellsSmooth[,,i] = filter2( cells[,,i],
         filter = makeBrush(size = 51, shape = "gaussian", sigma = sigma[i]) )
```

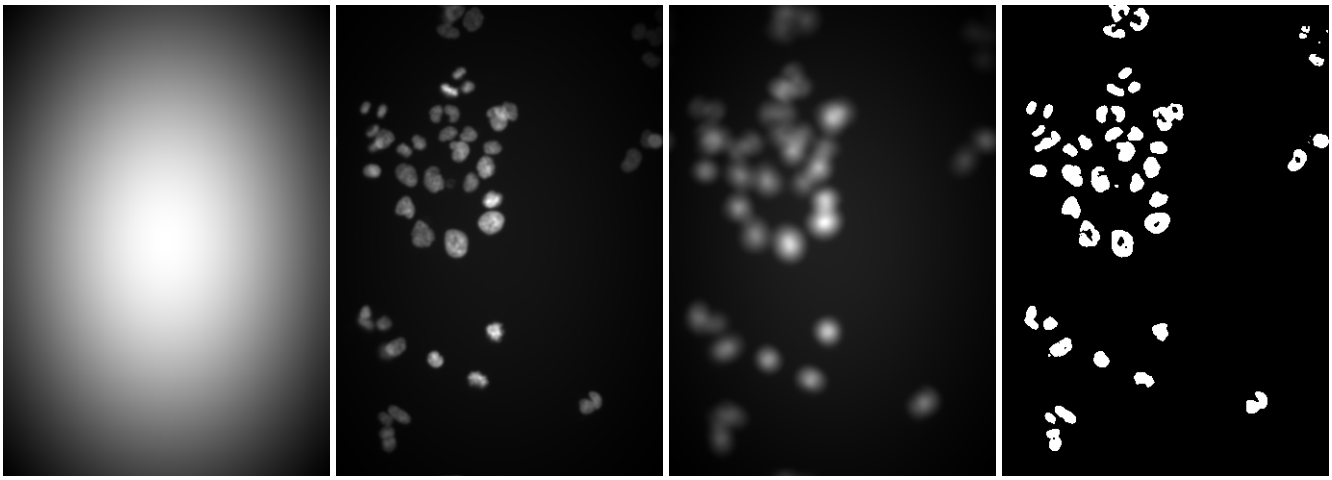

Figure 13.10: From left to right: `illuminationMask`, a function that has its maximum at the center and falls off towards the sides, and which simulates uneven illumination sometimes seen in images; `nucBadlyIlluminated`, the image that results from multiplying the DNA channel in `cellsSmooth` with `illuminationMask`; `localBackground`, the result of applying a linear filter with a bandwidth that is larger than the objects to be detected; `nucBadThresh`, the result of adaptive thresholding. The nuclei at the periphery of the image are reasonably well identified, despite the drop off in signal strength.

13.9 Adaptive thresholding

The idea of adaptive thresholding is that, compared to straightforward thresholding as we did for Figure 13.6, 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. In fact, we have already seen an example for uneven background in the bottom right image of Figure 13.3.

Our colon cancer images (Figure 13.8) do not have such artefacts, but for demonstration, let's simulate uneven illumination by multiplying the image with a mask, `illuminationMask`, that has highest value in the middle and falls off to the sides (Figure 13.10).

```
py = seq(-1, +1, length.out = dim(cellsSmooth)[1])
px = seq(-1, +1, length.out = dim(cellsSmooth)[2])
illuminationMask = Image(outer(py, px, function(x, y) exp(-(x^2+y^2))))
nucBadlyIlluminated = cellsSmooth[,,1] * illuminationMask
```

We now define a smoothing window, `disc`, whose size is 21 pixels, and therefore bigger than the nuclei we want to detect, but small compared to the length scales of the illumination artifact. We use it to compute the image `localBackground` (shown in Figure 13.10) and the thresholded image `nucBadThresh`.

```
disc = makeBrush(21, "disc")
disc = disc / sum(disc)
localBackground = filter2( nucBadlyIlluminated, disc )
offset = 0.02
nucBadThresh = (nucBadlyIlluminated - localBackground > offset)
```

After having seen that this may work, let's do the same again for the actual (not artificially degraded) image, as we need this for the next steps.

```
nucThresh = (cellsSmooth[,,1] - filter2( cellsSmooth[,,1], disc ) > offset)
```

Critique: By comparing each pixel's intensity to a background determined from the values in a local neighborhood,

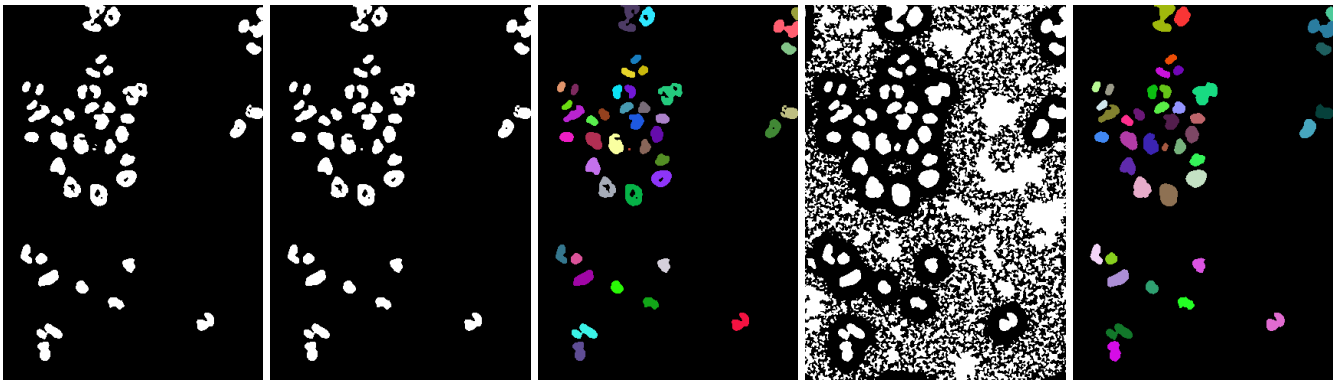

Figure 13.11: Different steps in the segmentation of the nuclei.

we assume that the objects are relatively sparse distributed in the image, so that the signal distribution in the neighborhood is dominated by background. For the nuclei in our images, this assumption makes sense, for other situations, you may need to make different assumptions. The adaptive thresholding that we have done here uses a linear filter, `filter2`, and therefore amounts to (weighted) local averaging. Other distribution summaries, e. g. the median or a low quantile, are useful in some cases, although such filters tend to be computationally more expensive. For local median filtering, *EBimage* provides the function `medianFilter`.

13.10 Morphological operations on binary images

The thresholded image `nucThresh` (shown in the left panel of Figure 13.11) is not yet satisfactory. The boundaries of the nuclei are slightly rugged, and there is noise at the single-pixel level. An effective and simple way to remove these nuisances is given by a set of morphological operations [10].

Provided a binary image (with values, say, 0 and 1, representing back- and foreground pixels), and a binary mask (which is sometimes also called the structuring element), these operations work as follows

- `erode`: For every foreground pixel, put the mask around it, and if any pixel under the mask is from the background, then set all these pixels to background.
- `dilate`: For every background pixel, put the mask around it, and if any pixel under the mask is from the foreground, then set all these pixels to foreground.
- `open`: perform `erode` followed by `dilate`.

We can also think of these operations as filters, however, in contrast to the linear filters of Section 13.8 they operate on binary images only, and there is no linearity.

Let us apply morphological opening to our image.

```
nucOpened = opening(nucThresh, kern = makeBrush(3, shape = "disc"))
```

The result of this is subtle, and you will have to zoom into the images in Figure 13.11 to spot the differences, but this operation manages to smoothen out some pixel-level features in the binary images that for our application were undesired.

13.11 Segmentation of a binary image into objects

The binary image `nucOpened` represents a segmentation of the image into foreground and background pixels, but not into individual nuclei. We can take one step further and extract individual objects defined as connected sets of pixels. In *EBImage*, there is a handy function for this purpose, `bwlabel`.

```
nucSeed = bwlabel(nucOpened)
table(nucSeed)
```

```
## nucSeed
##      0      1      2      3      4      5      6      7      8      9     10
## 155250    521    342    120    468    222    122    125    159    117    537
##     11     12     13     14     15     16     17     18     19     20     21
##    117    184    180    116    184    189    304    285    167    309    195
##     22     23     24     25     26     27     28     29     30     31     32
##    148    345    287    203    380    380    209      9    222    321    443
##     33     34     35     36     37     38     39     40     41     42     43
##    413    498    256    169    231    376    214    231    342    394    316
```

The function returns an image, `nucSeed`, of integer values, where 0 represents the background, and the numbers from 1 to 43 index the different identified objects. The numbers in the table equal the area (in pixels) of each object, and we could use this information to remove objects that are too large or too small compared to what we expect. To visualize such images, the function `colorLabels` is convenient, which converts the (grayscale) integer image into a color image, using distinct, arbitrarily chosen colors for each object.

```
display(colorLabels(nucSeed))
```

This is shown in the middle panel of Figure 13.11. The result is already encouraging, although we can spot two types of errors:

- Some neighboring objects were not properly separated.
- Some objects contain holes.

Indeed, we could change the occurences of these by playing with the disc size and the parameter `offset` in Section 13.9: making the offset higher reduces the probability that two neighboring object touch and are seen as one object by `bwlabel`; on the other hand, that leads to even more and even bigger holes. Vice versa for making it lower.

Critique. Segmentation is a rich and diverse field of research and engineering, with a large body of literature, software tools [9, 3, 2, 4] and practical experience in the image analysis and machine learning communities. What is the adequate approach to a given task depends hugely on the data and the underlying question, and there is no universally best method. It is typically even difficult to obtain a "ground truth" or "gold standards" by which to evaluate an analysis – relying on manual annotation of a modest number of selected images is not uncommon. Despite the bewildering array of choices, it is easy to get going, and we need not be afraid of starting out with a simple solution, which we can successively refine. Improvements can usually be gained from methods that allow inclusion of more prior knowledge of the expected shapes, sizes and relations between the objects to be identified.

For statistical analyses of high-throughput images, we may choose to be satisfied with a simple method that does not rely on too many parameters or assumptions and results in a perhaps sub-optimal but rapid and good enough result [8]. In this spirit, let us proceed with what we have. We generate a lenient foreground mask, which surely covers all nuclear stained regions, even though it also covers some regions between nuclei. To do so, we simply apply a second, less stringent adaptive thresholding.

```
nucMask = (cellsSmooth[,,1] - filter2( cellsSmooth[,,1], disc ) > 0)
```

and apply another morphological operation, `fillHull`, which fills holes that are surrounded by foreground pixels.

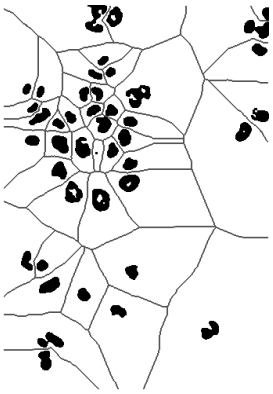

Figure 13.12: Example of a Voronoi segmentation, indicated by the gray lines, using the nuclei (indicated by black regions) as seeds.

```
nucMask = fillHull(nucMask)
```

To improve `nucSeed`, we can now *propagate* its segmented objects until they fill the mask defined by `nucMask`. Boundaries between nuclei, in those places where the mask is connected, can be drawn by Voronoi tessellation, which is implemented in the function `propagate`, and will be explained in the next section.

```
nuclei = propagate(cellsSmooth[,,1], nucSeed, mask = nucMask)
```

The result is displayed in the rightmost panel of Figure 13.11.

13.12 Voronoi tessellation

Voronoi tessellation is useful if 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. As this is an intuitive and powerful idea, we'll use this section for a short digression on it. Let us consider a basic example. We use the image `nuclei` as seeds. To call the function `propagate`, we also need to specify an image, for now we just provide a trivial image of all zeros, and we set the parameter `lambda` to a large positive value (we will come back to these choices).

```
zeroImage = Image(dim = dim(nuclei))
voronoiExamp = propagate(seeds = nuclei, x = zeroImage, lambda = 100)
voronoiPaint = paintObjects(voronoiExamp, 1 - nucOpened)
```

The result is shown in Figure 13.12. This looks interesting, but perhaps not yet as useful as the image `nuclei` in Figure 13.11. We note that 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 shortest distances not in a simple planar 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` parameter.
- By default, we think of the space as flat – but in fact it could have "hills" and "valleys", so that the distance between two points in the landscape not only depends on their x- and y-positions but also on the slopes that lie in between. You can specify such a landscape to `propagate` through the `x` argument.

Mathematically, we can say that instead of the simple default case (rectangular space with 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 thus not simply

$$ds = \sqrt{dx^2 + dy^2} \tag{13.3}$$

but instead

$$ds = \sqrt{\frac{1}{\lambda+1}\left[\lambda\left(dx^2+dy^2\right)+dz^2\right]}. \tag{13.4}$$

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 (13.4) becomes equivalent to (13.3), i. e., the importance of dz becomes negligible. This is what we did when we used `lambda = 100` in our call to `propagate` at the begin of this section.

For the purpose of cell segmentation, these ideas were put forward by Thouis Jones et al. [5, 2], who also wrote the efficient algorithm that is used by `propagate`.

fixme: Possible exercise: try out the effect of different λs.

13.13 Segmenting the cell bodies

To determine a mask of cytoplasmic area in the images, let us explore a different way of thresholding, this time using a global threshold which we find by fitting a mixture model to the data. Let us assume that the signal in the cytoplasmic channels of the *Image* `cells` is a mixture of two distributions: a log-Normal background and a foreground with another, unspecified, but mostly non-overlapping distribution; and that the majority of pixels are from the background. We can then find robust estimates for the location and width parameters of the log-Normal component from the half range mode (implemented in the package *genefilter*) and from the variance of the values that lie left of the mode.

```
library("genefilter")
muSigmaEstimator = function(x) {
  x    = log(x)
  mu   = half.range.mode( x )
  left = (x - mu)[ x < mu ]
  sd   = sqrt( sum(left ^ 2) / (length(left)-1) )
  c(mu = mu, sigma = sd)
}
logNormPars = apply(cellsSmooth, MARGIN = 3, FUN = muSigmaEstimator)
logNormPars
```

```
##           [,1]    [,2]    [,3]
## mu    -2.90177 -2.9443 -3.5219
## sigma  0.00635  0.0112  0.0153
```

Let us take as a threshold the mean plus 5 standard deviations[1]. *fixme: Alternatively, would phrasing this part as an example of Efron's* empirical null *be instructive?* We also need to exponentiate the result to bring it back to the scale of the image data.

```
thresholds = exp( logNormPars["mu", ] + 5 * logNormPars["sigma", ] )
```

[1] The choice of the number 5 here is arbitrary, we could formalize it by estimating the weight of the mixture components and assigning each pixel to either fore- or background based on its posterior probability according to the mixture distribution.

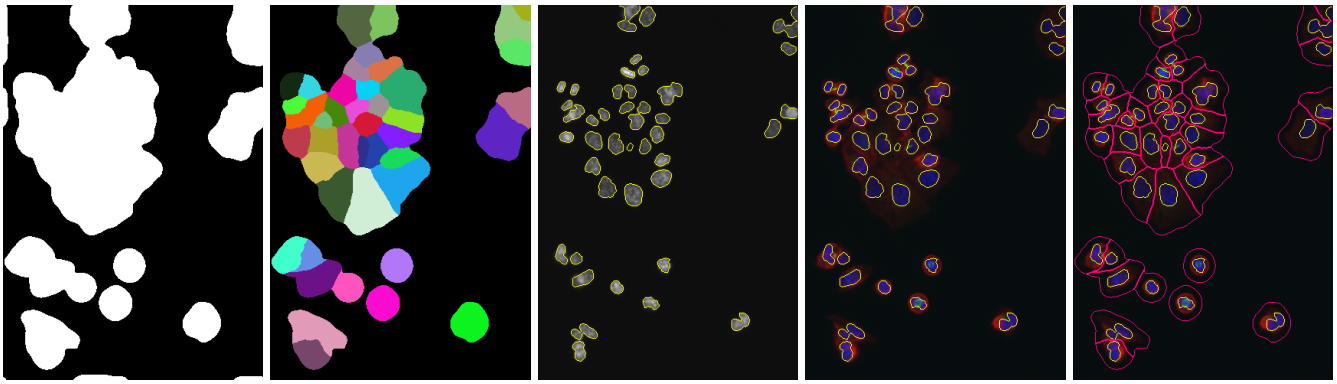

Figure 13.13: Steps in the segmentation of the cell bodies.

We can now define `cytoplasmMask` by the union of all those pixels that are above the threshold in the actin or tubulin image, or that we have already classified as nuclear in the image `nuclei`.

```
cytoplasmMask = (cellsSmooth[,,2] > thresholds[2]) | (cellsSmooth[,,3] > thresholds[3]) | nuclei
```

The result is shown in the left panel of Figure 13.13. To define the cellular bodies, we can now simply extend the nucleus segmentation within this mask by the Voronoi tessellation based propagation algorithm of Section 13.12. This method makes sure that there is exactly one cell body for each nucleus, and the cell bodies are delineated in such a way that a compromise is reached between compactness of cell shape and following the actin intensity signal in the images. In the terminology of the `propagate` algorithm, cell shape is kept compact by the x and y components of the distance metric (13.4), and the actin signal is used for the z component. λ controls the trade-off.

```
cellbodies = propagate(x = cellsSmooth[,,3], seeds = nuclei, lambda = 1.0e-2, mask = cytoplasmMask)
```

As an alternative representation to the `colorLabel` plots, we can also display the segmentations of nuclei and cell bodies on top of the original images using the `paintObjects` function; the Images `nucSegOnNuc`, `nucSegOnAll` and `cellSegOnAll` that are computed below are show in the middle to right panels of Figure 13.13

```
cellsColor = rgbImage(red = cells[,,3], green = cells[,,2], blue = cells[,,1])

nucSegOnNuc  = paintObjects(nuclei,     tgt = toRGB(cells[,,1]), col = "#ffff00")
nucSegOnAll  = paintObjects(nuclei,     tgt = cellsColor,        col = "#ffff00")
cellSegOnAll = paintObjects(cellbodies, tgt = nucSegOnAll,       col = "#ff0080")
```

13.14 Feature extraction

Now that we have the segmentations `nuclei` and `cellbodies` together with the original image data `cells`, we can compute various descriptors, or features, for each cell. We already saw in the beginning of Section 13.11 how to use the base R function `table` to determine the total number and sizes of the objects. Let us now take this further and compute the mean intensity of the DAPI signal (`cells[,,1]`) in the segmented nuclei, the mean actin intensity (`cells[,,3]`) in the segmented nuclei and the mean actin intensity in the cell bodies.

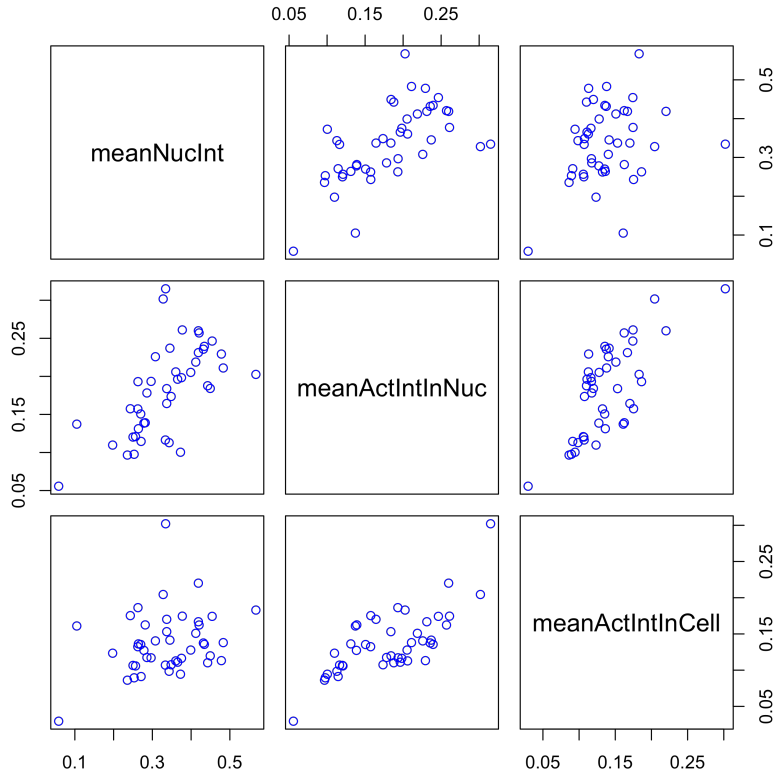

Figure 13.14: Pairwise scatterplots of per-cell intensity descriptors.

```
meanNucInt       = tapply(cells[,,1], nuclei, mean)
meanActIntInNuc  = tapply(cells[,,3], nuclei, mean)
meanActIntInCell = tapply(cells[,,3], cellbodies, mean)
```

We can visualize the features in pairwise scatterplots (Figure 13.14). We see that they are correlated with each other, although each feature also carries independent information.

```
pairs(cbind(meanNucInt, meanActIntInNuc, meanActIntInCell), col="blue")
```

With a little more work, we could also compute more sophisticated summary statistics – e. g., the ratio of nuclei area to cell body area; or entropies, mutual information and correlation of the different fluorescent signals in each cell body, as more or less abstract measures of cellular morphology. Such measures can be used, for instance, to detect subtle drug induced changes of cellular architecture.

While it is easy and intuitive to perform these computations using basic R idioms like in the `tapply` expressions above, the package *EBImage* also provides the function `computeFeatures` which efficiently computes a large collection of features that have been commonly used in the literature (a pioneering reference is [1]). Details about this function are described in its manual page, and an example application is worked through in the *HD2013SGI* vignette. Below, we compute features for intensity, shape and texture for each cell from the DAPI channel using the nucleus segmentation (`nuclei`) and from the actin and tubulin channels using the cell body segmentation (`cytoplasmRegions`).

```
F1 = computeFeatures(nuclei,     cells[,,1], xname = "nuc",  refnames = "nuc")
F2 = computeFeatures(cellbodies, cells[,,2], xname = "cell", refnames = "tub")
```

```
F3 = computeFeatures(cellbodies, cells[,,3], xname = "cell", refnames = "act")
dim(F1)
```

```
## [1] 43 89
```

`F1` is a *matrix* with 43 rows (one for each cell) and 89 columns, one for each of the computed features.

```
 F1[1:3, 1:5]
```

```
##   nuc.0.m.cx nuc.0.m.cy nuc.0.m.majoraxis nuc.0.m.eccentricity nuc.0.m.theta
## 1        120       17.5              44.9                0.837         -1.31
## 2        143       15.8              26.2                0.663         -1.21
## 3        337       11.5              19.0                0.856          1.47
```

The column names encode the type of feature, as well the color channel(s) and segmentation mask on which it was computed.

13.15 Recap of the chapter

We learned to work with image data in R – they're basically just arrays, and we can use familiar idioms to manipulate them. We can extract data from images, and then many of the analytical questions are not unlike those with other high-throughput data: say, summarisation of population behavior into relevant statistics, testing for differences between treatment conditions, robust fitting of models, clustering and classification.

We have learned how to take two-dimensional images of cell populations – acquired with multiple fluorescence markers for different cellular components, in our case, DNA, actin and tubulin – to segment the individual cells and extract quantitative features relating to their size, shape and numbers.

13.16 Acknowledgments

Bernd Fischer: package *HD2013SGI*; Andrzej Oleś: package *MSBdata*, useful feedback on this chapter.

13.17 Session Info

- R Under development (unstable) (2014-06-25 r66030), `x86_64-apple-darwin13.2.0`
- Locale: `en_GB.UTF-8/en_GB.UTF-8/en_GB.UTF-8/C/en_GB.UTF-8/en_GB.UTF-8`
- Base packages: base, datasets, graphics, grDevices, methods, stats, utils
- Other packages: EBImage 4.6.0, fortunes 1.5-2, genefilter 1.46.1, knitr 1.6, MSBdata 0.2.0
- Loaded via a namespace (and not attached): abind 1.4-0, annotate 1.42.0, AnnotationDbi 1.26.0, Biobase 2.24.0, BiocGenerics 0.10.0, DBI 0.2-7, evaluate 0.5.5, formatR 0.10, GenomeInfoDb 1.0.2, grid 3.2.0, highr 0.3, IRanges 1.22.9, jpeg 0.1-8, lattice 0.20-29, locfit 1.5-9.1, parallel 3.2.0, png 0.1-7, RSQLite 0.11.4, splines 3.2.0, stats4 3.2.0, stringr 0.6.2, survival 2.37-7, tiff 0.1-5, tools 3.2.0, XML 3.98-1.1, xtable 1.7-3

Bibliography

[1] M. V. Boland and R. F. Murphy. A neural network classifier capable of recognizing the patterns of all major subcellular structures in fluorescence microscope images of HeLa cells. *Bioinformatics*, 17(12):1213–1223, 2001.

[2] A.E. Carpenter, T.R. Jones, M.R. Lamprecht, C. Clarke, I.H. Kang, O. Friman, D.A. Guertin, J.H. Chang, R.A. Lindquist, J. Moffat, et al. CellProfiler: image analysis software for identifying and quantifying cell phenotypes. *Genome Biology*, 7:R100, 2006.

[3] Fabrice de Chaumont, Stéphane Dallongeville, Nicolas Chenouard, Nicolas Hervé, Sorin Pop, Thomas Provoost, Vannary Meas-Yedid, Praveen Pankajakshan, Timothé Lecomte, Yoann Le Montagner, Thibault Lagache, Alexandre Dufour, and Jean-Christophe Olivo-Marin. Icy: an open bioimage informatics platform for extended reproducible research. *Nature Methods*, 9:690–696, 2012.

[4] M. Held, M.H.A. Schmitz, B. Fischer, T. Walter, B. Neumann, M.H. Olma, M. Peter, J. Ellenberg, and D.W. Gerlich. CellCognition: time-resolved phenotype annotation in high-throughput live cell imaging. *Nature Methods*, 7:747, 2010.

[5] T. Jones, A. Carpenter, and P. Golland. Voronoi-based segmentation of cells on image manifolds. *Computer Vision for Biomedical Image Applications*, page 535, 2005.

[6] Christina Laufer, Bernd Fischer, Maximilian Billmann, Wolfgang Huber, and Michael Boutros. Mapping genetic interactions in human cancer cells with RNAi and multiparametric phenotyping. *Nature Methods*, 10:427–431, 2013.

[7] G. Pau, F. Fuchs, O. Sklyar, M. Boutros, and W. Huber. EBImage - an R package for image processing with applications to cellular phenotypes. *Bioinformatics*, 26:979, 2010.

[8] S. Rajaram, B. Pavie, L. F. Wu, and S. J. Altschuler. PhenoRipper: software for rapidly profiling microscopy images. *Nature Methods*, 9:635–637, 2012.

[9] Johannes Schindelin, Ignacio Arganda-Carreras, Erwin Frise, Verena Kaynig, Mark Longair, Tobias Pietzsch, Stephan Preibisch, Curtis Rueden, Stephan Saalfeld, Benjamin Schmid, Jean-Yves Tinevez, Daniel James White, Volker Hartenstein, Kevin Eliceiri, Pavel Tomancak, and Albert Cardona. Fiji: an open-source platform for biological-image analysis. *Nature Methods*, 9:676–682, 2012.

[10] Jean Serra. *Image Analysis and Mathematical Morphology*. Academic Press, 1983.

[11] Martin Vetterli, Jelena Kovačević, and Vivek Goyal. *Foundations of Signal Processing*. Cambridge University Press, 2014.