Chapter 11 Examples

Brooke Anderson

4/28/2020

```
library(tidyverse)
```

First part—working with image data

First, you'll need the `EBImage` package from Bioconductor. This is a package with lots of tools for processing images and then analyzing the resulting data.

```
# Uncomment and run the following line if you need the package
# BiocManager::install("EBImage")
library("EBImage")

# To access its vignette, uncomment and run the following line:
# vignette("EBImage-introduction")
```

It has a function called `readImage` that lets you read in an image file, like a file that ends ".png" or ".jpeg". There's an example file that comes with the book's datasets called "mosquito.png", so you can try reading that in:

```
mosq <- "data/mosquito.png" %>%
  readImage()
```

Here's what that data looks like:

```
mosq
```

```
## Image
##   colorMode    : Color
##   storage.mode : double
##   dim          : 1400 952 4
##   frames.total : 4
##   frames.render: 1
##
## imageData(object)[1:5,1:6,1]
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
## [1,] 0.1490196 0.1490196 0.1490196 0.1490196 0.1490196 0.1490196
## [2,] 0.1490196 0.1490196 0.1490196 0.1490196 0.1490196 0.1490196
## [3,] 0.1490196 0.1490196 0.1490196 0.1529412 0.1490196 0.1490196
## [4,] 0.1490196 0.1490196 0.1529412 0.1568627 0.1490196 0.1490196
## [5,] 0.1490196 0.1490196 0.1607843 0.1647059 0.1490196 0.1450980
```

It's in a special class called "Image":

```
mosq %>%
  class()
```

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

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

It includes a matrix where every cell in the matrix represents a pixel in the image, and each value gives the color value. If you use `str`, you can figure out the "slot" in the R object that's storing this data. When you use `class`, you can see that it's an array, which is kind of like a matrix, but it allows you to have extra dimensions. For example, if you wanted to represent data in a matrix across different time points, you could use a third dimension for time—an array would be a great object class for that, because it would just let you "stack up" the matrices for each timepoint into one R object. (One way to think of an array is like a Rubic's cube, where each "slice" is a 3x3 matrix, and there are three of those in the array, so your total dimension would be 3x3x3.)

I think that in this case, they use those extra dimensions when the image is in color, so they can give the values for red, green, and blue to represent the color.

```
mosq %>%
  str()
```

```
## Formal class 'Image' [package "EBImage"] with 2 slots
##   ..@ .Data    : num [1:1400, 1:952, 1:4] 0.149 0.149 0.149 0.149 0.149 ...
##   ..@ colormode: int 2
```

```
mosq@.Data %>%
  class()
```

```
## [1] "array"
```

```
mosq@.Data %>%
  dim()
```

```
## [1] 1400  952    4
```

```
# Peak at a corner of the array
mosq@.Data %>%
  `[`(1:3, 1:3, 1)
```

```
##           [,1]      [,2]      [,3]
## [1,] 0.1490196 0.1490196 0.1490196
## [2,] 0.1490196 0.1490196 0.1490196
## [3,] 0.1490196 0.1490196 0.1490196
```

There's also an *accessor* function for this object class that lets you pull out the data:

```
mosq %>%
  imageData() %>%
  `[`(1:3, 1:3, 1)
```

```
##           [,1]      [,2]      [,3]
## [1,] 0.1490196 0.1490196 0.1490196
## [2,] 0.1490196 0.1490196 0.1490196
## [3,] 0.1490196 0.1490196 0.1490196
```

If you want, you can check out the four "layers" in this array by making a tile plot of each:

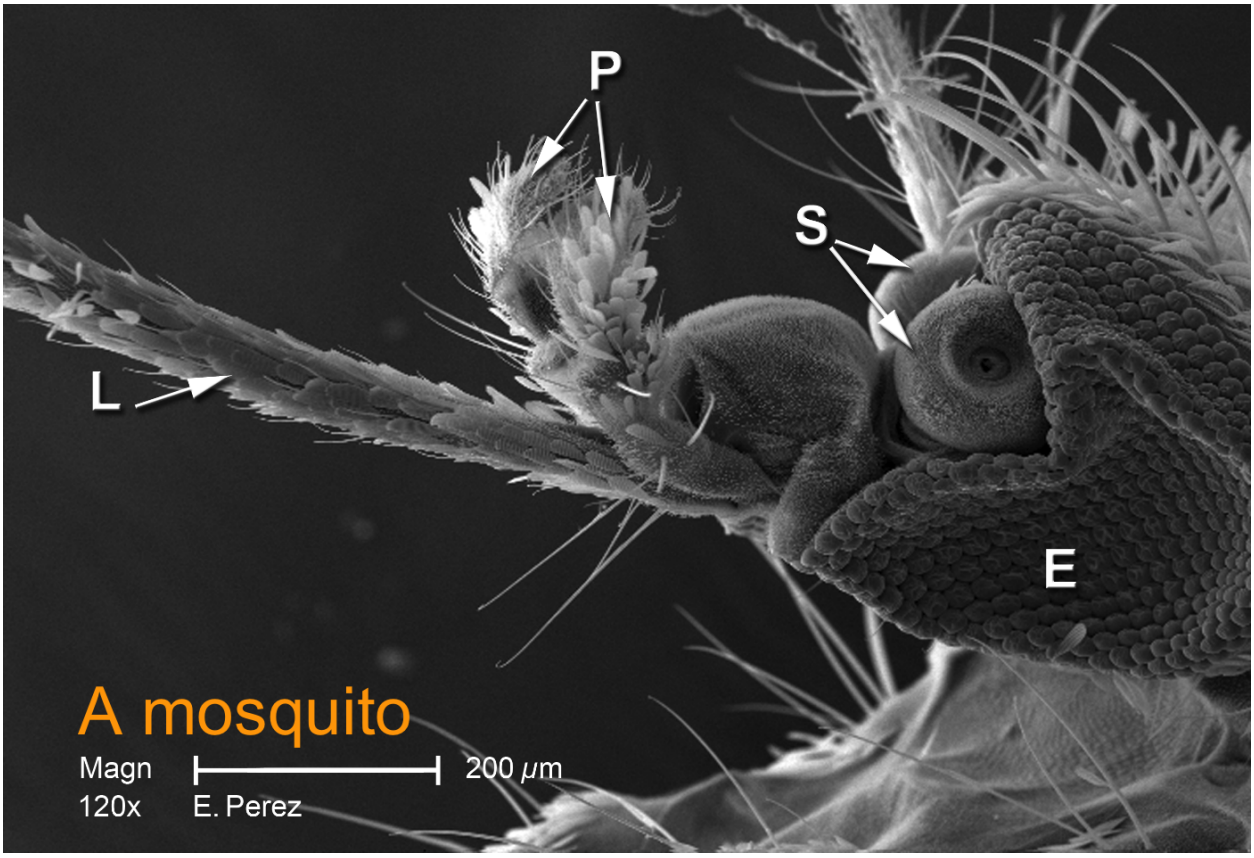
```
# I've currently set this code to not evaluate, but you can try it out on your
# own computer.
mosq@.Data %>%
  `[`(, , 1) %>%
  as_tibble() %>%
  mutate(row_n = 1:n()) %>%
  pivot_longer(-row_n) %>%
```

```
  ggplot(aes(x = name, y = row_n, fill = value)) +
  geom_tile() +
  theme_void()

mosq@.Data %>%
  `[`(, , 2) %>%
  as_tibble() %>%
  mutate(row_n = 1:n()) %>%
  pivot_longer(-row_n) %>%
  ggplot(aes(x = name, y = row_n, fill = value)) +
  geom_tile() +
  theme_void()

mosq@.Data %>%
  `[`(, , 3) %>%
  as_tibble() %>%
  mutate(row_n = 1:n()) %>%
  pivot_longer(-row_n) %>%
  ggplot(aes(x = name, y = row_n, fill = value)) +
  geom_tile() +
  theme_void()

mosq@.Data %>%
  `[`(, , 4) %>%
  as_tibble() %>%
  mutate(row_n = 1:n()) %>%
  pivot_longer(-row_n) %>%
  ggplot(aes(x = name, y = row_n, fill = value)) +
  geom_tile() +
  theme_void()
```

It looks like the content of the first three is really similar (maybe just the different color channels—red, blue, and green?), and then there's really nothing stored in the fourth dimension of the array in this example.

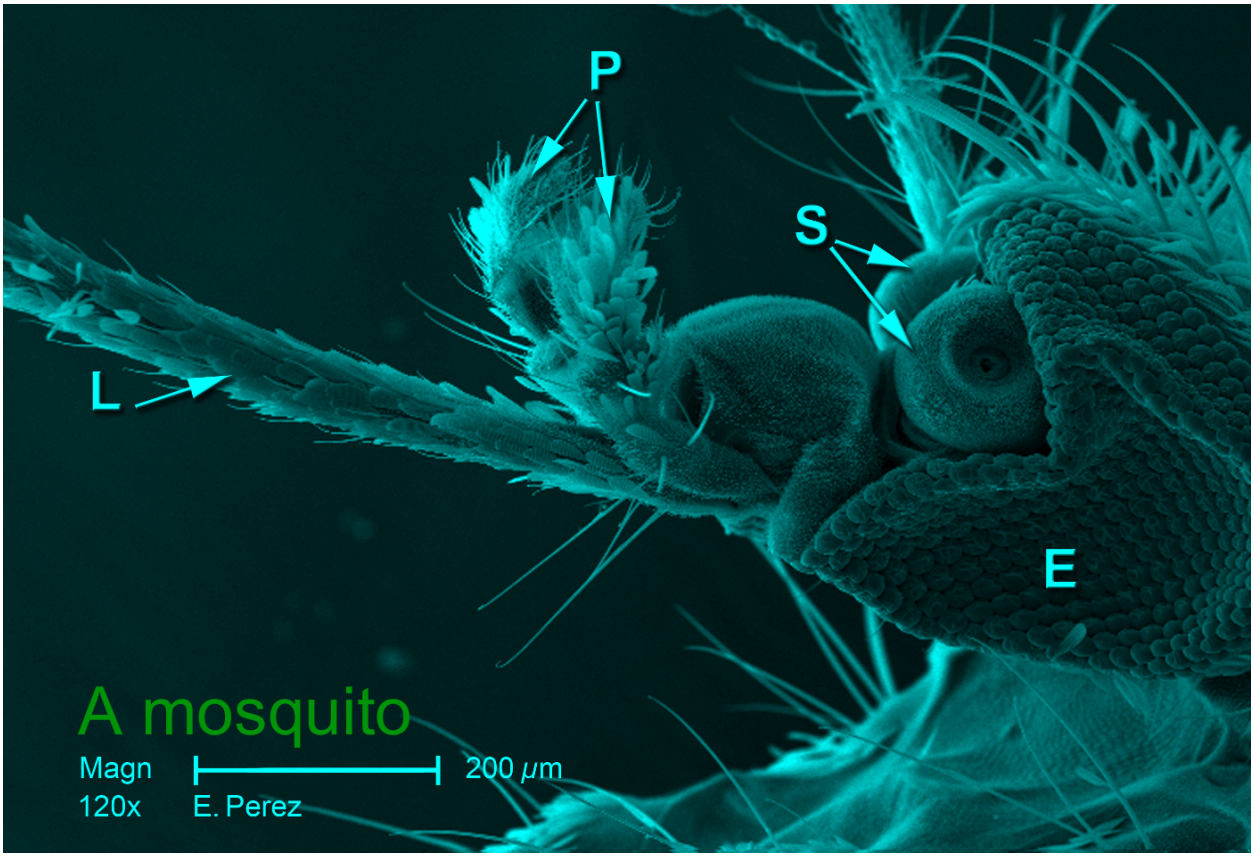
You can also use the `display` method for this object class to show the picture:

```
mosq %>%
  # Use `raster` to print out as static image, rather than show in web browser
  display(method = "raster")
```



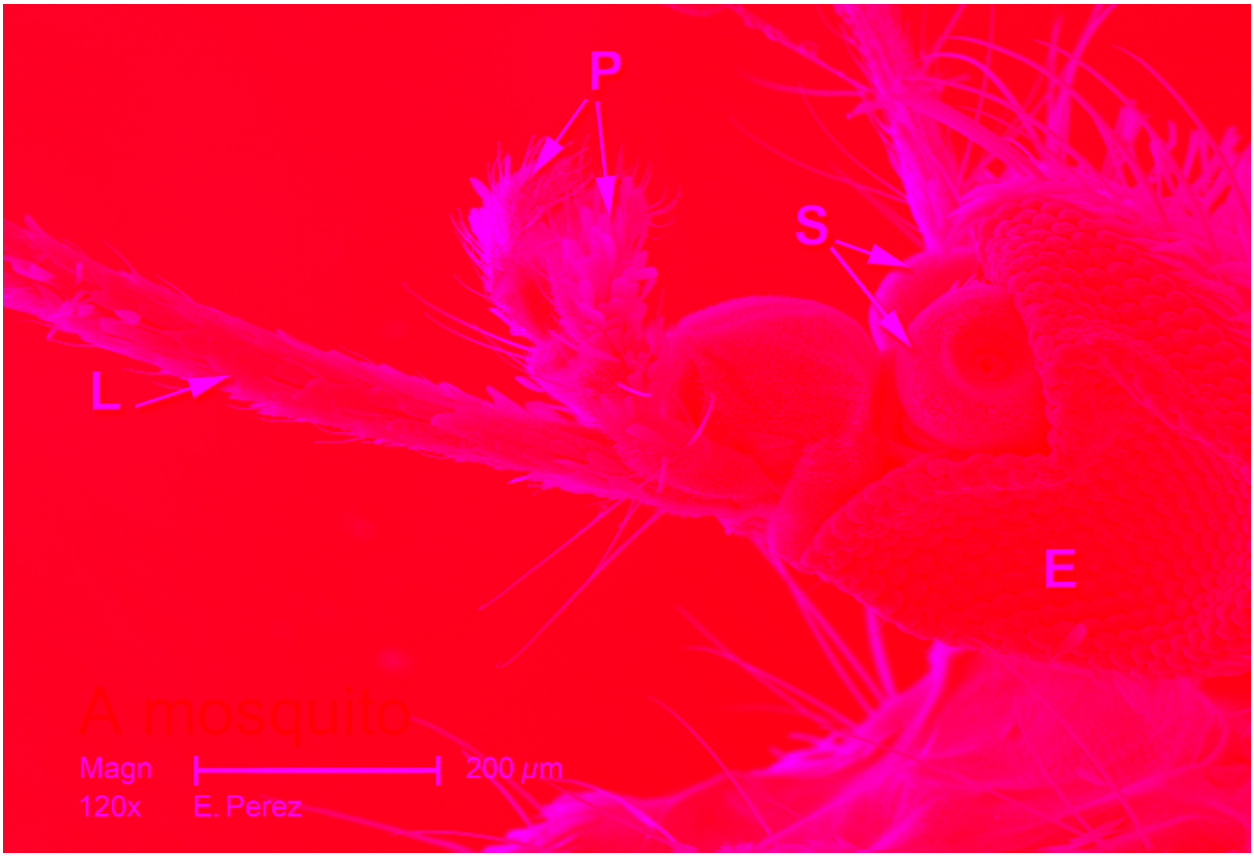
Play around some.... what if we replace all of the values in the first channel with "0"s?

```
mosq2 <- mosq
mosq2@.Data[ , , 1] <- 0
mosq2 %>%
  display(method = "raster")
```



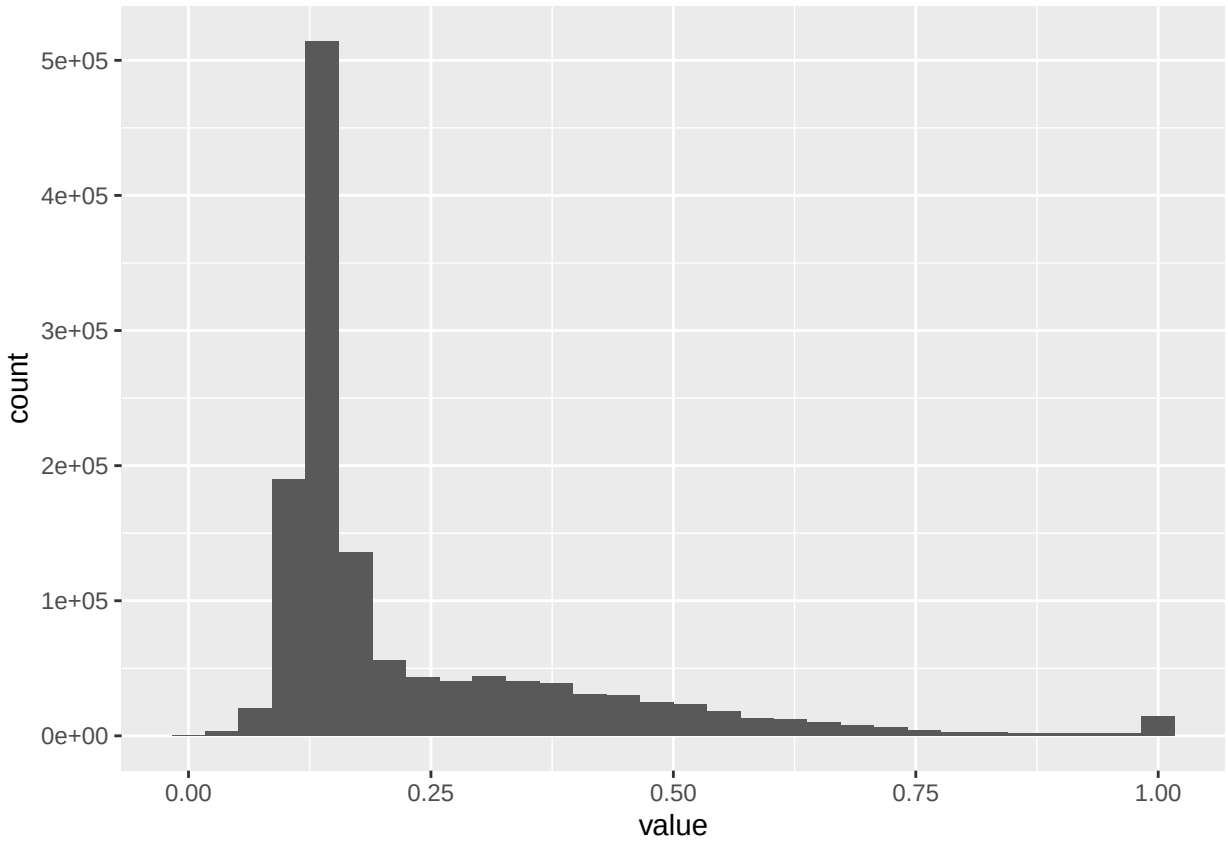
What if we replace them all with "1"s and replace everything in the second channel with "0"s?

```
mosq3 <- mosq
mosq3@.Data[ , , 1] <- 1
mosq3@.Data[ , , 2] <- 0
mosq3 %>%
  display(method = "raster")
```



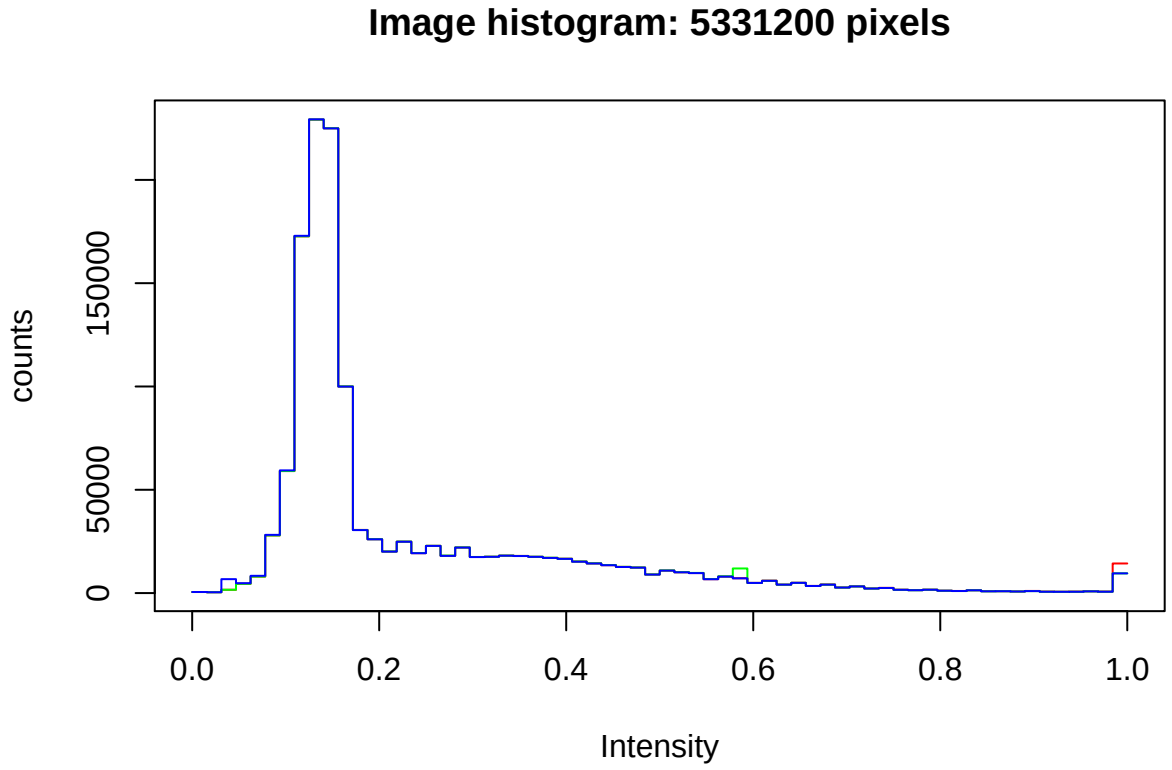
You can use histograms to see the intensity of each pixel in each channel of the image's data. For example, the following code pulls all the data for the first channel (maybe red color?)

```
mosq %>%
  # Pull out the data from the object
  imageData() %>%
  # Pull out just the first "slice" of the array
  `[`(, , 1) %>%
  # Convert this from an array to a single vector
  as.vector() %>%
  # Put back in a tibble to make ggploting easier
  # (the default will be to have one column named "value" in the tibble)
  as_tibble() %>%
  # Create the histogram
  ggplot(aes(x = value)) +
  geom_histogram()
```



There's also a `hist` method for this object class that lets you create this histogram (it looks like it does it for all three channels at once):

```
mosq %>%
  hist()
```

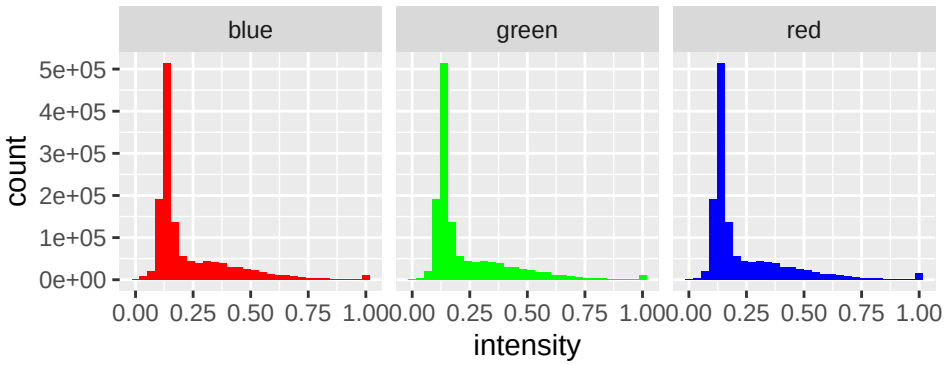


For help on this function, you can call:

```
?`hist,Image-method`
```

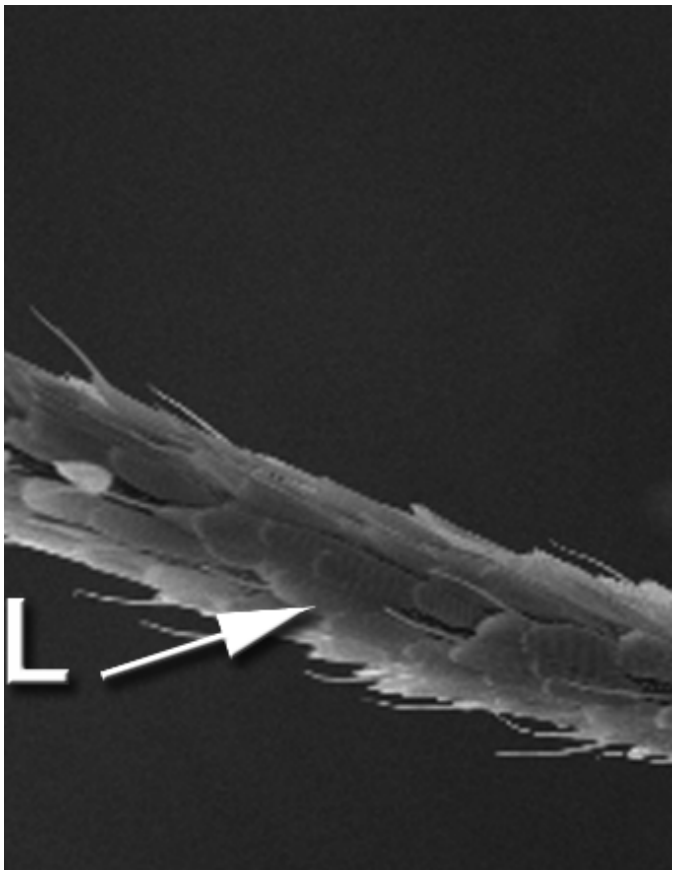
Another way you could try creating a histogram with the color intensities (in this case, with panels for each color):

```
# Put each color channel's values into a column in a tibble (pull each
# as a vector)
tibble(red = as.vector(imageData(mosq)[ , , 1]),
       green = as.vector(imageData(mosq)[ , , 2]),
       blue = as.vector(imageData(mosq)[ , , 3])) %>%
  # Pivot the data to make it easier to plot with faceting
  pivot_longer(everything(), names_to = "color_channel", values_to = "intensity") %>%
  # Create the plot
  ggplot(aes(x = intensity, fill = color_channel)) +
  geom_histogram() +
  facet_wrap(~ color_channel, nrow = 1) +
  scale_fill_manual(values = c("red", "green", "blue")) +
  # No need for the fill legend, so you can take it off
  theme(legend.position = "none")
```



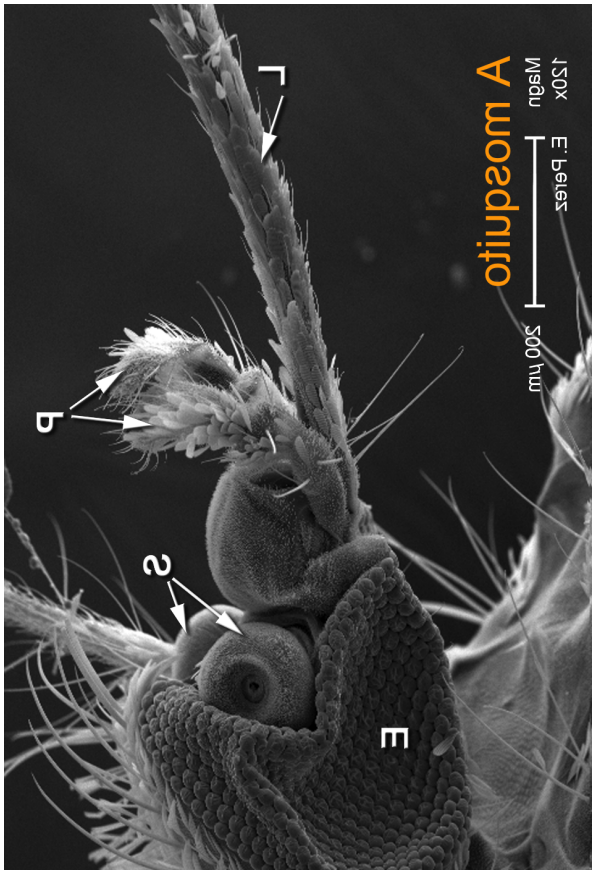
Use index selection to "crop" the image:

```
mosq %>%
  `[`(100:438, 112:550, ) %>%
  display(method = "raster")
```



Transpose the image:

```
mosq %>%
  transpose() %>%
  display(method = "raster")
```



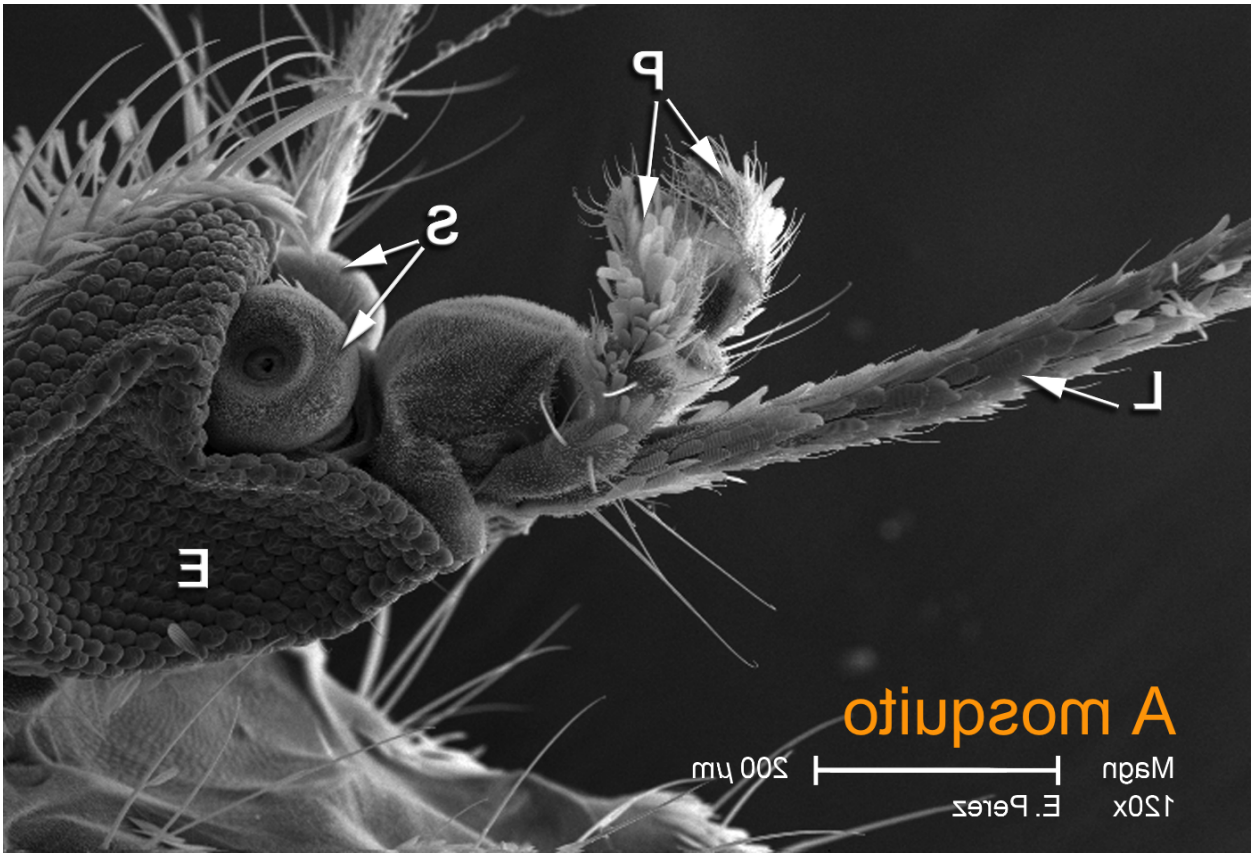
Rotate the image with `rotate`:

```
mosq %>%
  rotate(angle = 30) %>%
  display(method = "raster")
```

Flip and flop the image:

```
mosq %>%
  flip() %>%
  display(method = "raster")
```

```
mosq %>%
  flop() %>%
  display(method = "raster")
```



Working with cell image data

For the cell image data, you'll need to pull it from the "MSMB" R package, because it's not available in the book's data files.

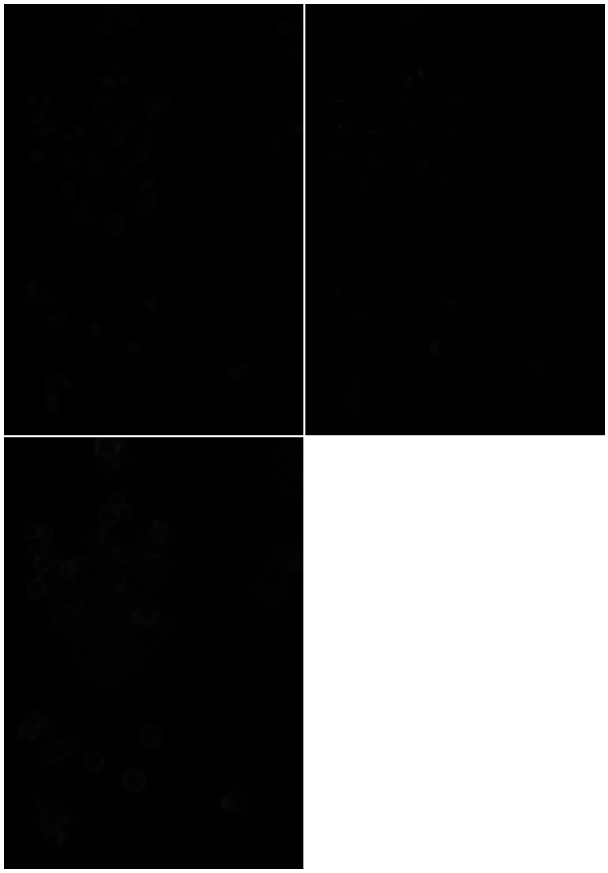
```
# If you need the package, uncomment and run the following line
# BiocManager::install("MSMB")

# `system.file` finds the filepaths for these files on your computer
# (they cam when you installed the MSMB package)
system.file("images",
            c("image-DAPI.tif", "image-FITC.tif", "image-Cy3.tif"),
            package = "MSMB") %>%
  # Read these files in as an Image object and save to an object
  # named "cells"
  readImage() -> cells
cells
```

```
## Image
##   colorMode    : Grayscale
##   storage.mode : double
##   dim          : 340 490 3
##   frames.total : 3
##   frames.render: 3
##
## imageData(object)[1:5,1:6,1]
##             [,1]        [,2]        [,3]        [,4]        [,5]        [,6]
```

```
## [1,] 0.001724269 0.001693751 0.001739528 0.001678492 0.001724269 0.001663233
## [2,] 0.001739528 0.001724269 0.001693751 0.001693751 0.001739528 0.001709010
## [3,] 0.001678492 0.001754788 0.001724269 0.001724269 0.001678492 0.001663233
## [4,] 0.001724269 0.001709010 0.001693751 0.001739528 0.001693751 0.001647974
## [5,] 0.001739528 0.001709010 0.001770047 0.001678492 0.001724269 0.001678492
```

Check out this image:

```
cells %>%
  display(method = "raster", all = TRUE)
```

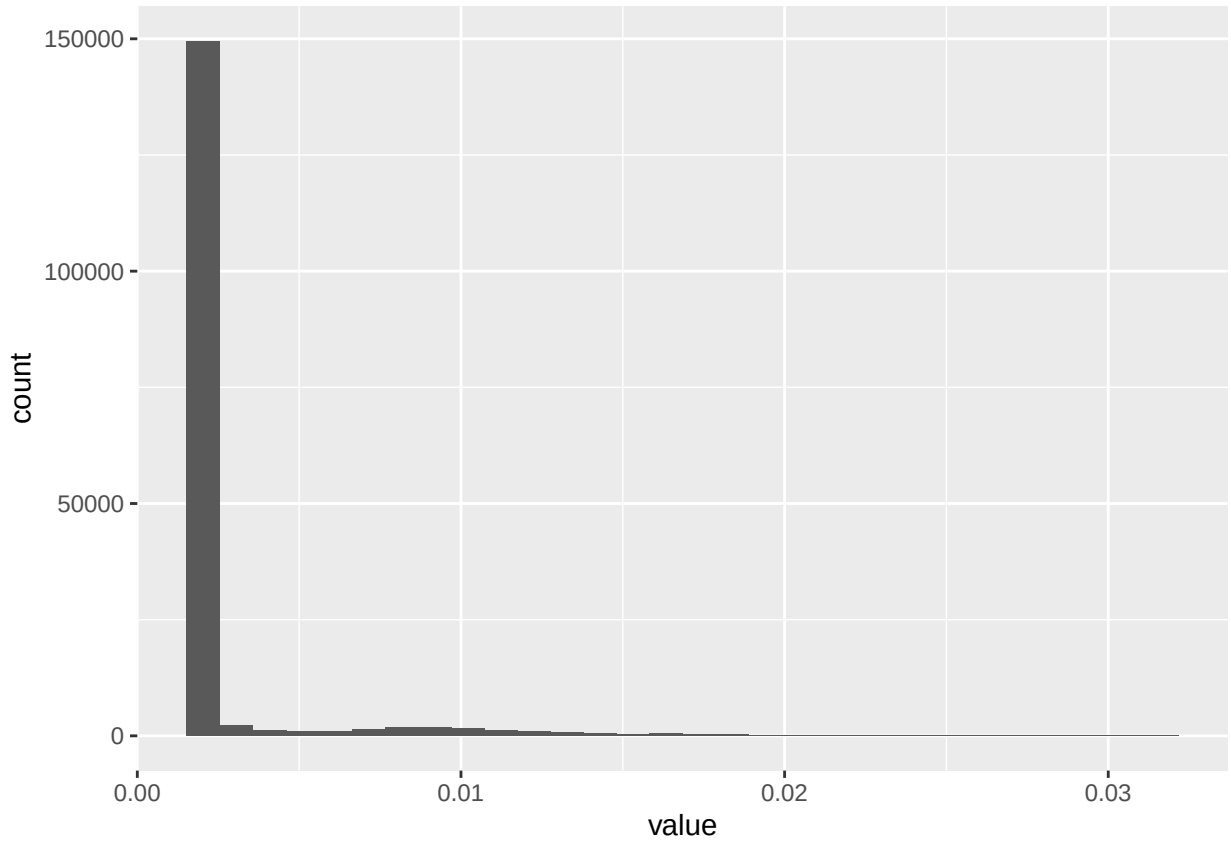


Unfortunately, it looks like these are just blank?

```
cells %>%
  dim()
```

```
## [1] 340 490   3
```

```
cells %>%
  imageData() %>%
  `[`(, , 1) %>%
  as.vector() %>%
  as_tibble() %>%
  ggplot(aes(x = value)) +
  geom_histogram()
```



Maybe not.... it looks like there is a bit of variation....

You can use `apply` to check the range of values in each matrix over one of the array's dimensions (in this case, "3", so you're getting the range for each staining channel, maybe?):

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

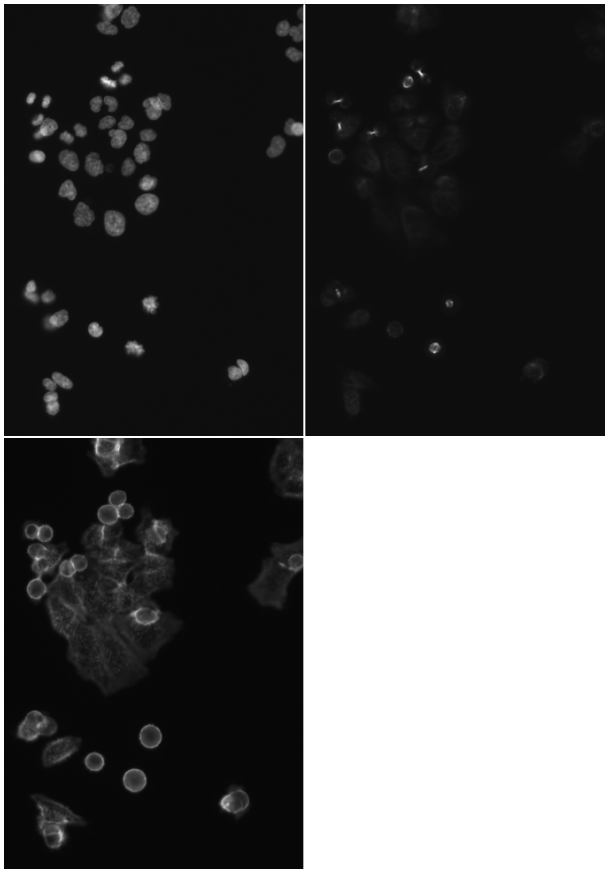
```
##       image-DAPI  image-FITC   image-Cy3
## [1,] 0.001586938 0.002899214 0.001663233
## [2,] 0.031204700 0.062485695 0.055710689
```

So you can see something, you need to pump up the intensities in all three channels (more in the first, DNA channel, but also some in the tubulin and actin channels):

```
# This is reassigning the values in the first channel to be 32 times
# their original value (and the values in channels 2 and 3 to 16 times their
# original values)
cells[ , , 1] <- 32 * cells[ , , 1]
cells[ , , 2:3] <- 16 * cells[ , , 2:3]
```

Try visualizing again:

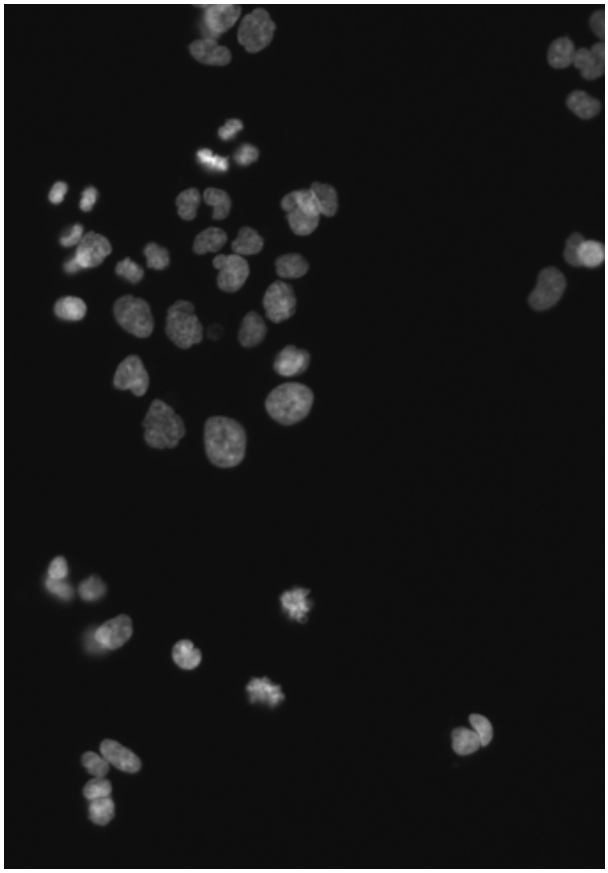
```
cells %>%
  display(method = "raster", all = TRUE)
```



There we go...

When you have multiple channels like this, you can also use the "frame" option to show just one. For example, to show just the first channel (the DNA one), you can run:

```
cells %>%
  display(method = "raster", frame = 1)
```



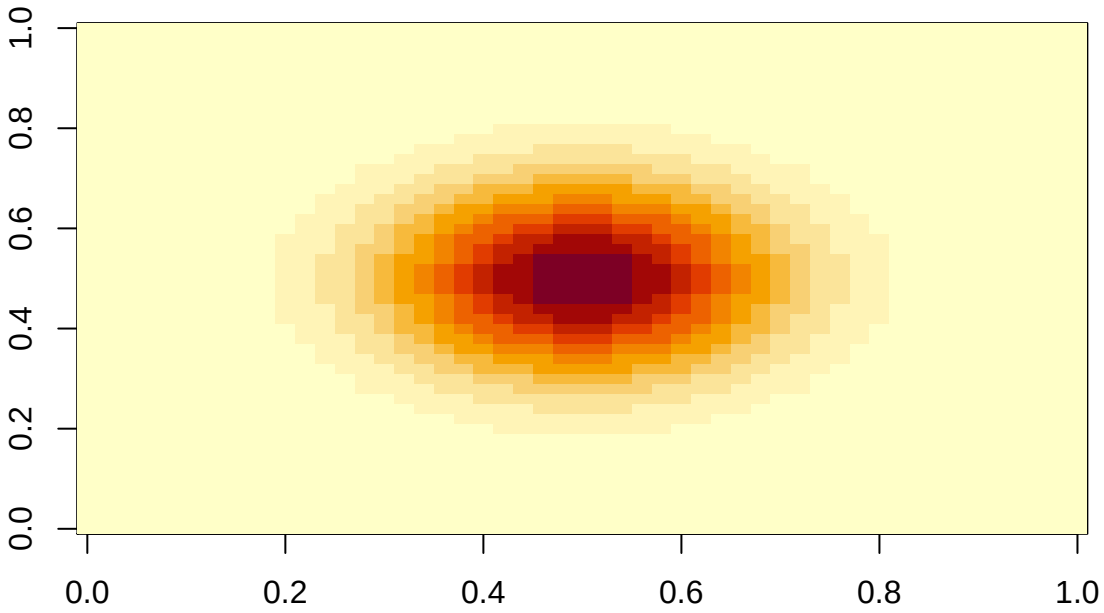
Next, we want to be able to tease apart the different cells in this image (*segmentation*) and then measure some things about each cell (*feature extraction*).

First, we evidently want to smooth out some of the noise from the image. We can use a "filter" with a "brush" to do that. First, make the brush:

```
w <- makeBrush(size = 51, shape = "gaussian", sigma = 7)

# It's 51 x 51, since we asked for size = 51:
w %>%
  dim()
```

```
## [1] 51 51
```

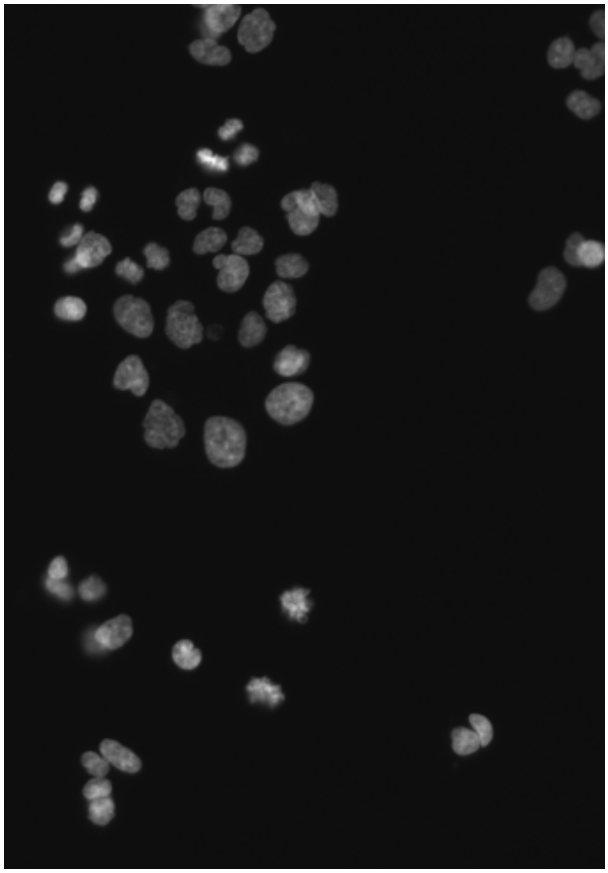
```
# Look at an image (you could do this with a geom_tile in
# ggplot, too, after some reshaping of the data)
w %>%
  image()
```



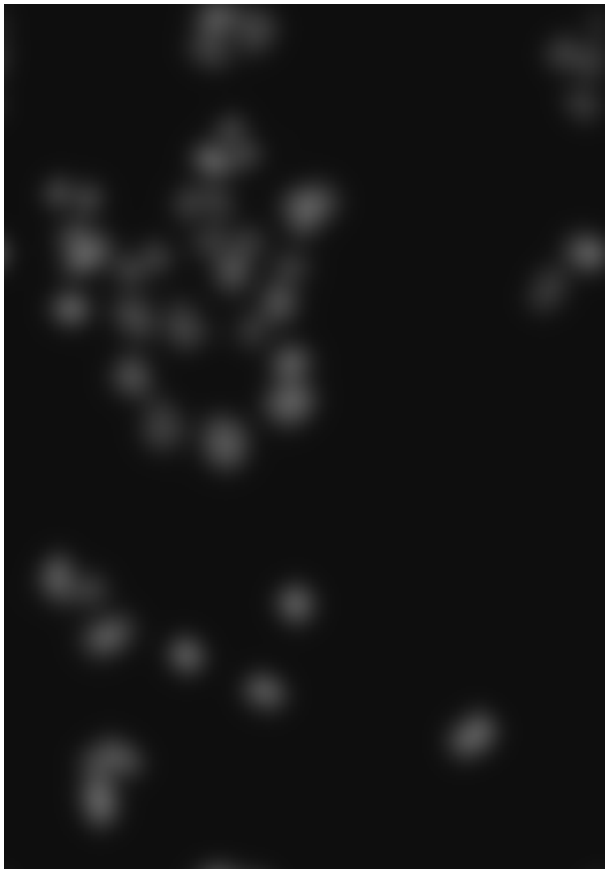
You can see that this has a two-dimensional "bell curve" (gaussian) shape.

Here's what the first channel (the DNA channel) looks like before and after you smooth some of the noise with this brush:

```
# Before
cells %>%
  # Get just the first channel (the DNA one)
  getFrame(1) %>%
  display(method = "raster")
```



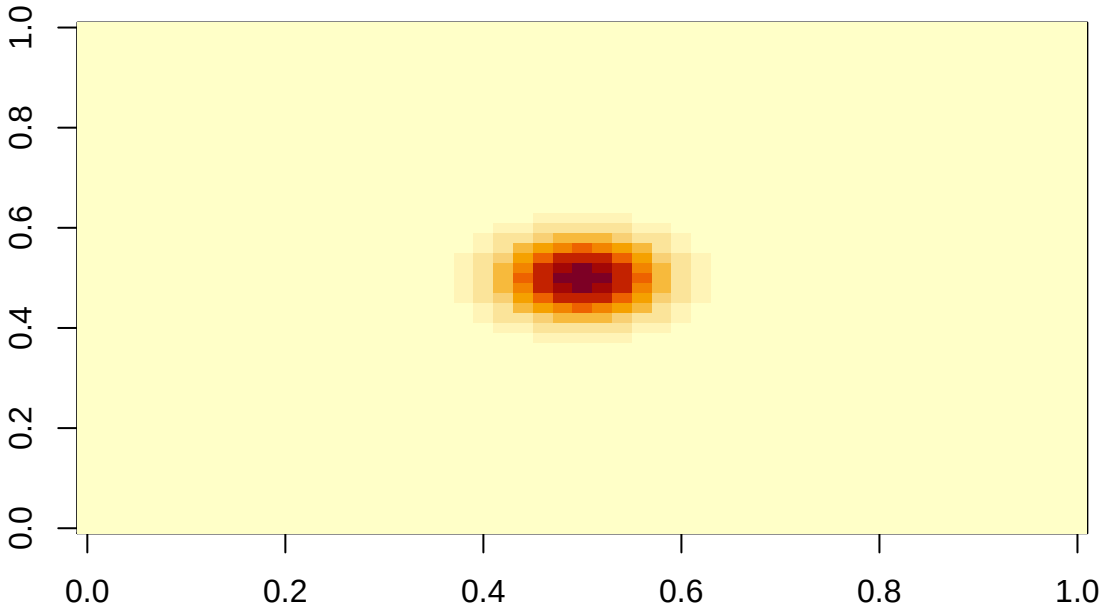
```
# After
cells %>%
  # Get just the first channel (the DNA one)
  getFrame(1) %>%
  # Apply the brush, w, that we made in the last code chunk
  filter2(w) %>%
  display(method = "raster")
```



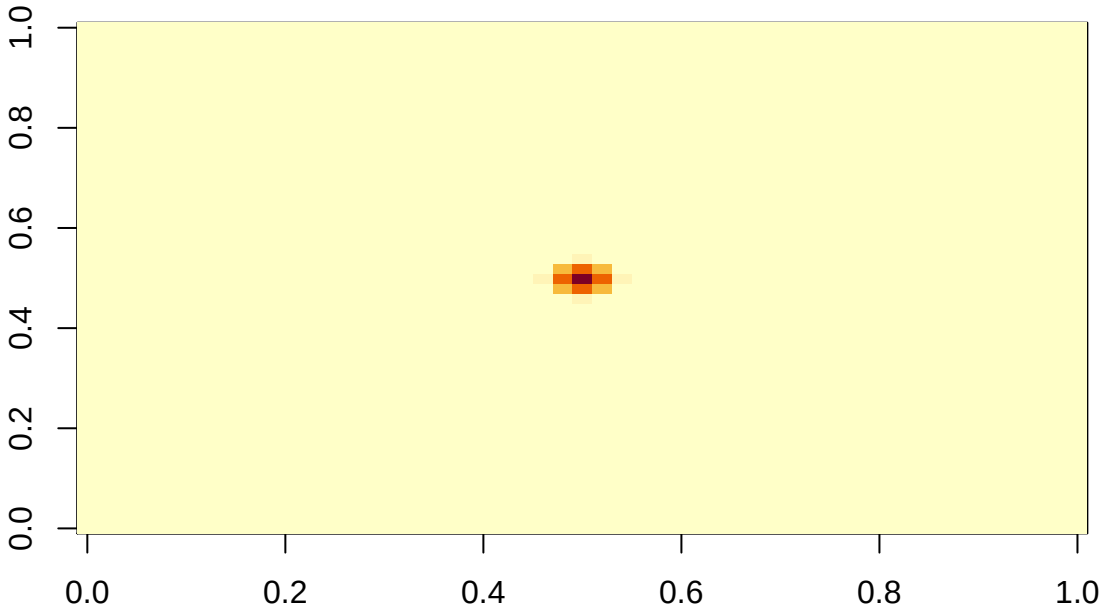
Everything in the image gets smoothed by this process, so you end up with blurry shapes for the cells.

They note that this is too smooth (too blurry), so you can try lower values of sigma when you make the brush. For example, here's what a brush would look like with sigmas of 3 and 1:

```
# sigma of 3
makeBrush(size = 51, shape = "gaussian", sigma = 3) %>%
  image()
```

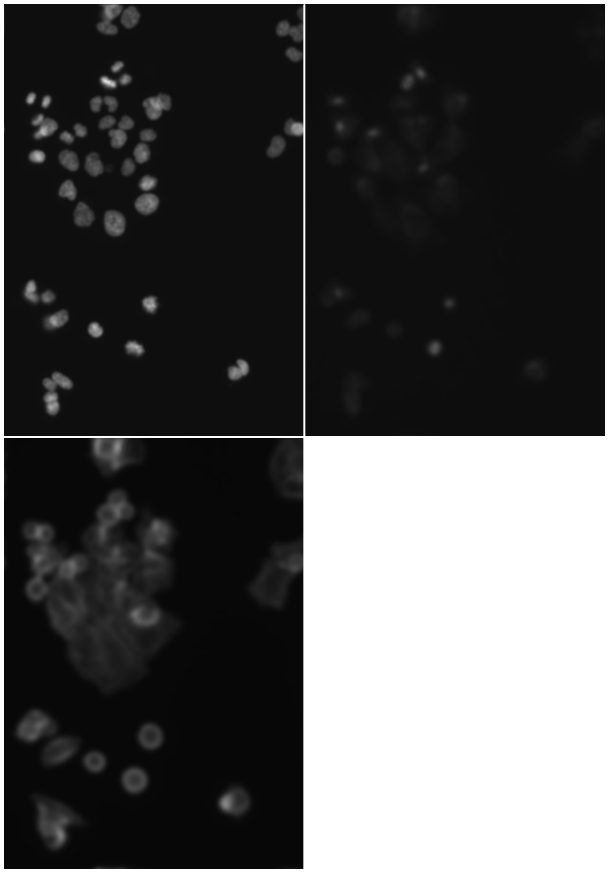


```r
# sigma of 1
makeBrush(size = 51, shape = "gaussian", sigma = 1) %>%
  image()
```



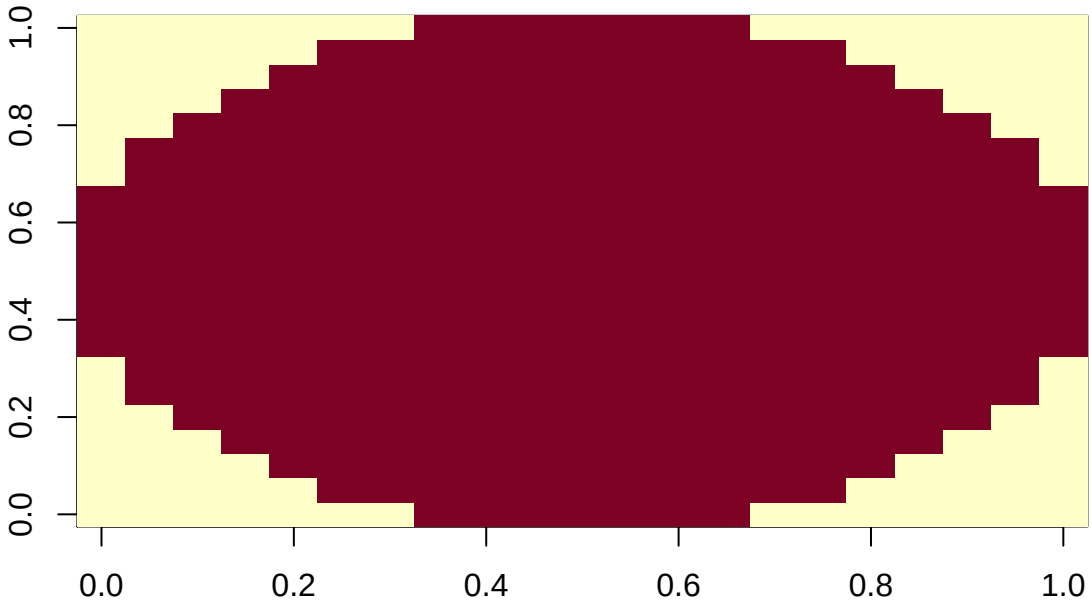
They decide to smooth the DNA channel (the first channel) with a brush with sigma of 1 and the other two channels (tubulin and actin) with a brush with sigma 3. Here's an alternative of how to do that, rather than the code in the book:

```r
# Make the two brushes
brush_sigma1 <- makeBrush(size = 51, shape = "gaussian", sigma = 1)
brush_sigma3 <- makeBrush(size = 51, shape = "gaussian", sigma = 3)

# Copy the original image into a new object
cellsSmooth <- cells

# Replace each channel with the smoothed version of that channel
cellsSmooth[, , 1] <- cellsSmooth[, , 1] %>%
  filter2(brush_sigma1)
cellsSmooth[, , 2] <- cellsSmooth[, , 2] %>%
  filter2(brush_sigma3)
cellsSmooth[, , 3] <- cellsSmooth[, , 3] %>%
  filter2(brush_sigma3)

# Check out the results
cellsSmooth %>%
  display(method = "raster", all = TRUE)
```



Next, they use adaptive thresholding on the image, to clean it up while taking advantage of the fact that we expect there to be spatial dependencies in the data. They show some examples of bad choices, but here's the code for the one they finally pick:

```
# Make a brush for this adaptive thresholding
disc <- makeBrush(size = 21, shape = "disc")
disc <- disc / sum(disc) # Normalize the values in the brush
# Here's what this brush looks like
disc %>%
  image()
```




```
# Apply the brush to get a thresholded value---a new matrix where values
# are logical for whether the difference between smoothed values in the image
# and the smoothed values after filtering with the disc are higher than
# some offset value (0.02 is chosen here)
nucThres <- (cellsSmooth[ , , 1] - filter2(cellsSmooth[ , , 1], disc)) > 0.02

nucThres[1:10, 1:10]
```

```
## Image
##   colorMode    : Grayscale
##   storage.mode : logical
##   dim          : 10 10
##   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
```

You can see that all the values in the image are now TRUE or FALSE (binary, so pure black and white).

You can take a look at the image now (we only did this for the first channel, so that's all you'll see):

```
nucThres %>%
  display(method = "raster")
```

Next, you can use an `opening` operation on this first channel of the image. There is a function with this name in both the `EBImage` and the `spatstat` libraries, so be sure to specify the `EBImage::` at the start of the function so your R session will pick the right one.

```
nucOpened <- nucThres %>%
  EBImage::opening(kern = makeBrush(5, shape = "disc"))

# Again, this is all TRUE / FALSE
nucOpened[1:5, 1:5]
```

```
## Image
##   colorMode    : Grayscale
##   storage.mode : logical
##   dim          : 5 5
##   frames.total : 1
##   frames.render: 1
##
## imageData(object)[1:5,1:5]
##       [,1]  [,2]  [,3]  [,4]  [,5]
## [1,] FALSE FALSE FALSE FALSE FALSE
## [2,] FALSE FALSE FALSE FALSE FALSE
## [3,] FALSE FALSE FALSE FALSE FALSE
## [4,] FALSE FALSE FALSE FALSE FALSE
## [5,] FALSE FALSE FALSE FALSE FALSE
```

```
# Check the image
nucOpened %>%
```

```
  display(method = "raster")
```

Now that you've cleaned up the object some (removed some of the noise), you want to segment it, to pick out the individual cells. You can use the `bwlabel` function to label connected sets of pixels in the image at this point:

```
nucSeed <- nucOpened %>%
  bwlabel()

nucSeed
```

```
## Image
##   colorMode    : Grayscale
##   storage.mode : integer
##   dim          : 340 490
##   frames.total : 1
##   frames.render: 1
##
## imageData(object)[1:5,1:6]
##      [,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
```

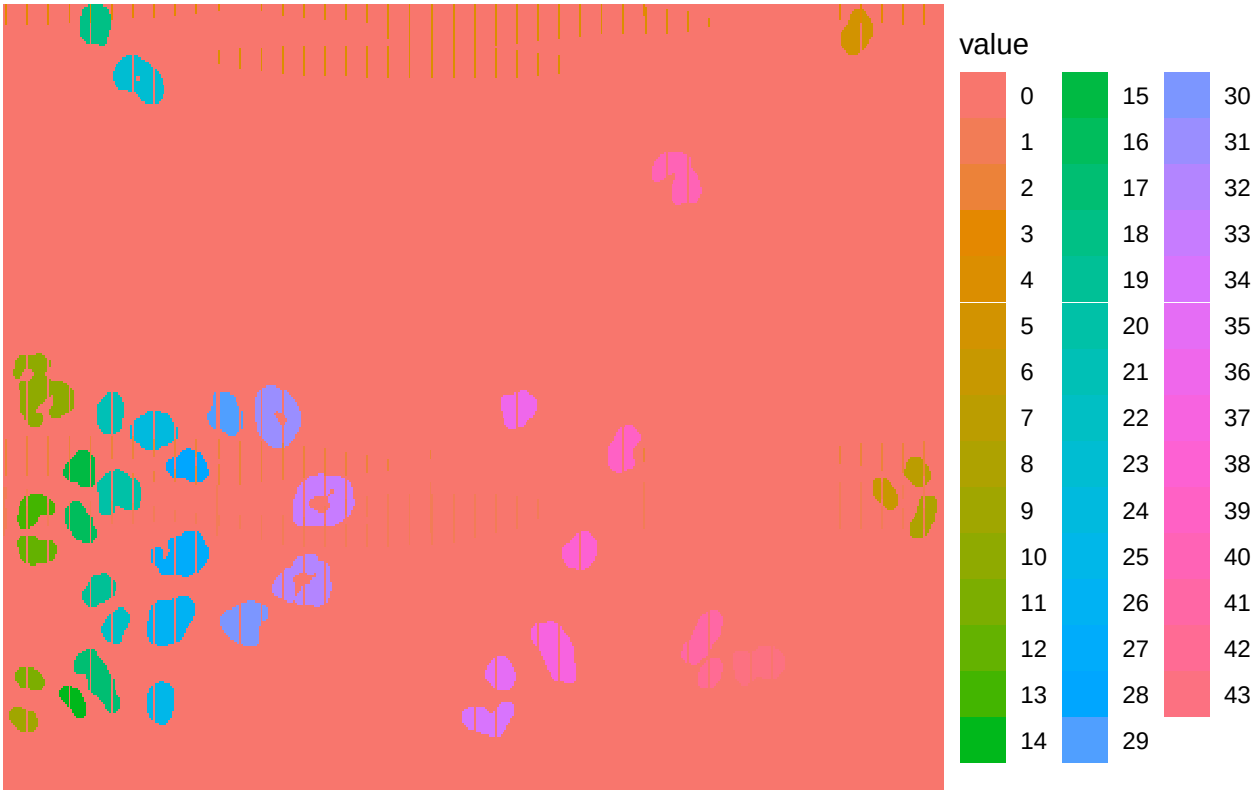
```r
# Get a summary of the values that are filling the image's data matrix
nucSeed %>%
  imageData() %>%
  # Stretch into a single vector so you can summarize
  # everything at once
  as.vector() %>%
  summary()
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   0.000   0.000   1.526   0.000  43.000
```

```r
# Count up the number of values with each group label
nucSeed %>%
  imageData() %>%
  as.vector() %>%
  as_tibble() %>%
  group_by(value) %>%
  count()
```

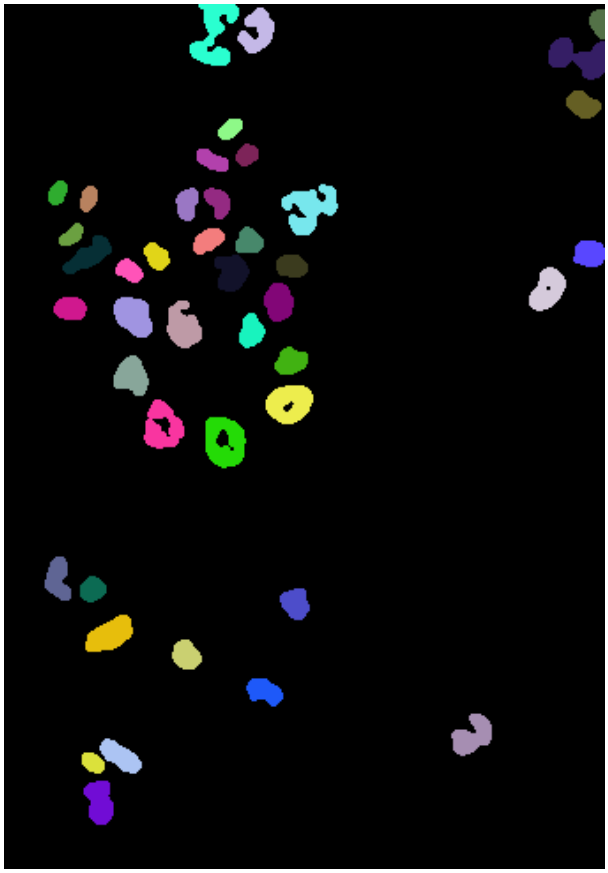
```
## # A tibble: 44 x 2
## # Groups:   value [44]
##    value      n
##    <int>  <int>
##  1     0 155408
##  2     1    511
##  3     2    330
##  4     3    120
##  5     4    468
##  6     5    222
##  7     6    121
##  8     7    125
##  9     8    159
## 10     9    116
## # ... with 34 more rows
```

```r
# Explore this data a bit with a plot
nucSeed %>%
  imageData() %>%
  as_tibble() %>%
  mutate(rownumber = 1:n()) %>%
  pivot_longer(-rownumber) %>%
  mutate(value = as_factor(value)) %>%
  ggplot(aes(x = name, y = rownumber, fill = value)) +
  geom_tile() +
  theme_void()
```



There's also a way to plot this using the `display` method from the `EBImage` package:

```
nucSeed %>%
  colorLabels() %>%
  display(method = "raster")
```



This has done a pretty good job of separating the different cells. However, there are still a few cases where two or more cells were close together and were assigned to the same cell number with this analysis.

They re-do some of the analysis from the adaptive thresholding, using a less stringent mask for that step, and then add `fillHull` to fill in holes that show up in some of the cells in the image, and then `propogate` segmented objects to fill the mask (evidently, this helps set up for the Voronoi tesselation step):

```
# Make a mask that does both adaptive thresholding and hull filling. This
# is just for the first panel (the DNA staining one) of the image. It uses
# the `disc` brush that we defined earlier
nucMask <- ((cellsSmooth[ , , 1] - filter2(cellsSmooth[ , , 1], disc)) > 0) %>%
  fillHull()

nuclei <- cellsSmooth[ , , 1] %>%
  propagate(seeds = nucSeed, mask = nucMask)

nuclei
```

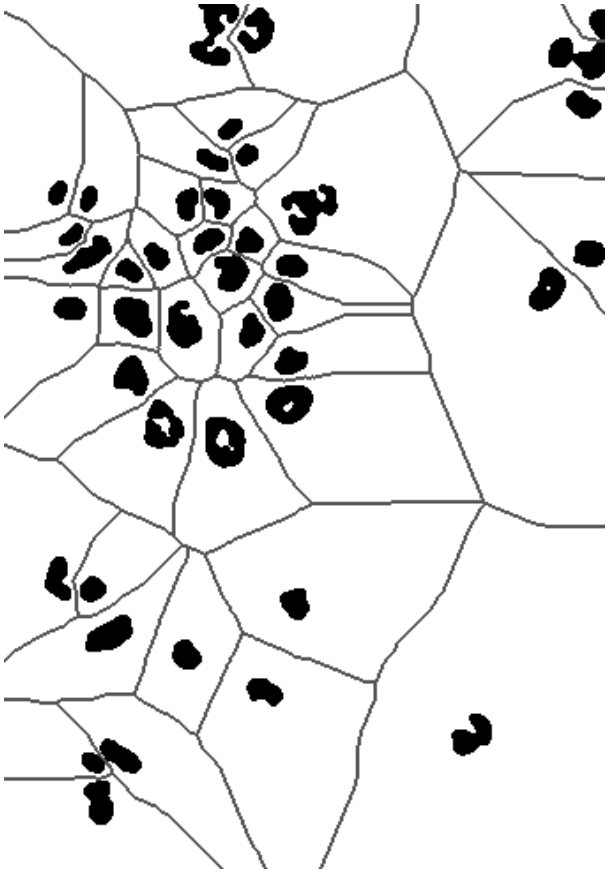
```
## Image
##   colorMode    : Grayscale
##   storage.mode : integer
##   dim          : 340 490
##   frames.total : 1
##   frames.render: 1
##
## imageData(object)[1:5,1:6]
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    0    0    0    0    0    0
```

```
## [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
```

```
nuclei %>%
  display(method = "raster")
```

You can see from this that the holes in some cells have now been filled in.

```
# Start with an image with only zeros but otherwise the same
# size as the `nuclei` image
Image(dim = dim(nuclei)) %>%
  # propagate, using the `nuclei` image as seeds, and force to rely
  # on distance in x and y directions (by using a really high lambda of 100)
  propagate(seeds = nuclei, lambda = 100) %>%
  # Visualize the divisions created by the Voronoi tesselation, while adding
  # on to the image the cells as of the `nucOpened` stage of the image processing
  paintObjects(tgt = 1 - nucOpened) %>%
  display(method = "raster")
```



Immune and cancer cells case study

For this, you want to read in all the files that start with "99_4525D" in the book's data folder

```
# You can use regular expressions in `list.files` to get these names:
list.files(path = "data", pattern = "99_4525D.+")
```

```
## [1] "99_4525D-DCs.txt"         "99_4525D-other_cells.txt"
## [3] "99_4525D-T_cells.txt"     "99_4525D-Tumor.txt"
```

These are all comma-separated flat files. We want to read them all in as dataframes and then stick them together into one long dataframe, with a column for the class of cells (e.g., "T_cells", "Tumor").

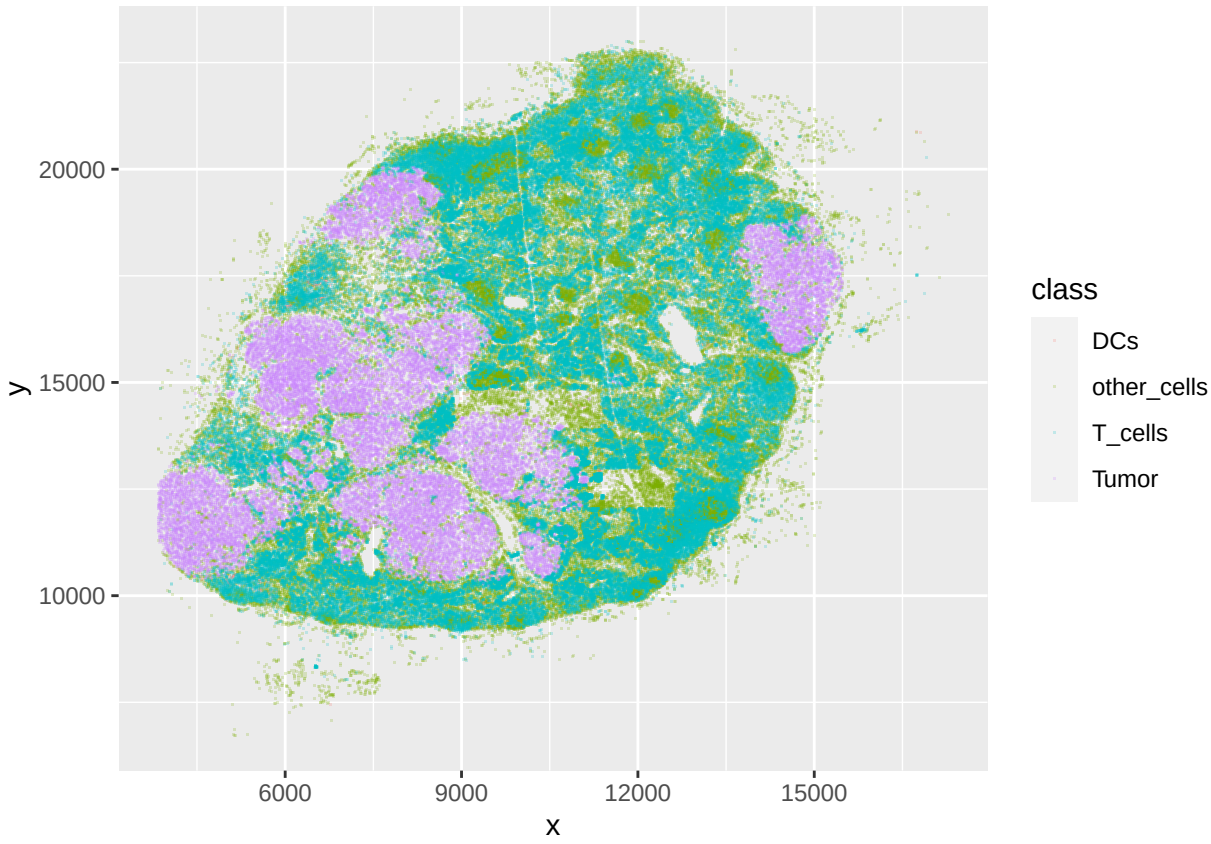
```
# Create a vector of the filepaths for everything you want to read in
# Use `full.names` since it's in a subdirectory of the current directory
list.files(path = "data", pattern = "99_4525D.+", full.names = TRUE) %>%
  # Put these into a dataframe. The default column name will be "values",
  # which you can rename to "filepath"
  as_tibble() %>%
  rename(filepath = value) %>%
  # Use `map` (from `purrr`) to apply `read_csv` to all these files
  # It will read the data in as a special column list in the tibble,
  # while keeping your filenames (so you can pull out the cell type)
  mutate(data = map(filepath, read_csv)) %>%
  # Use regular expressions to pull out the cell type from the file name
  # One way is to remove all the stuff you don't want from both the beginning
  # and the end of the file names. Make sure it's a "factor" class.
  mutate(class = filepath %>%
```

```
            str_remove("data/99_4525D-") %>%
            str_remove(".txt"),
          class = as_factor(class)) %>%
  # Get rid of the filepath column, since you don't need it anymore, and
  # then "unnest" the special list-column to unpack the data
  select(-filepath) %>%
  unnest(data) %>%
  # Clean up and rename some columns, then save as the object called "brcalymphnode
  select(globalX:class) %>%
  rename(x = globalX,
         y = globalY) -> brcalymphnode

# Here's an example of what that data looks like:
brcalymphnode %>%
  head()
```

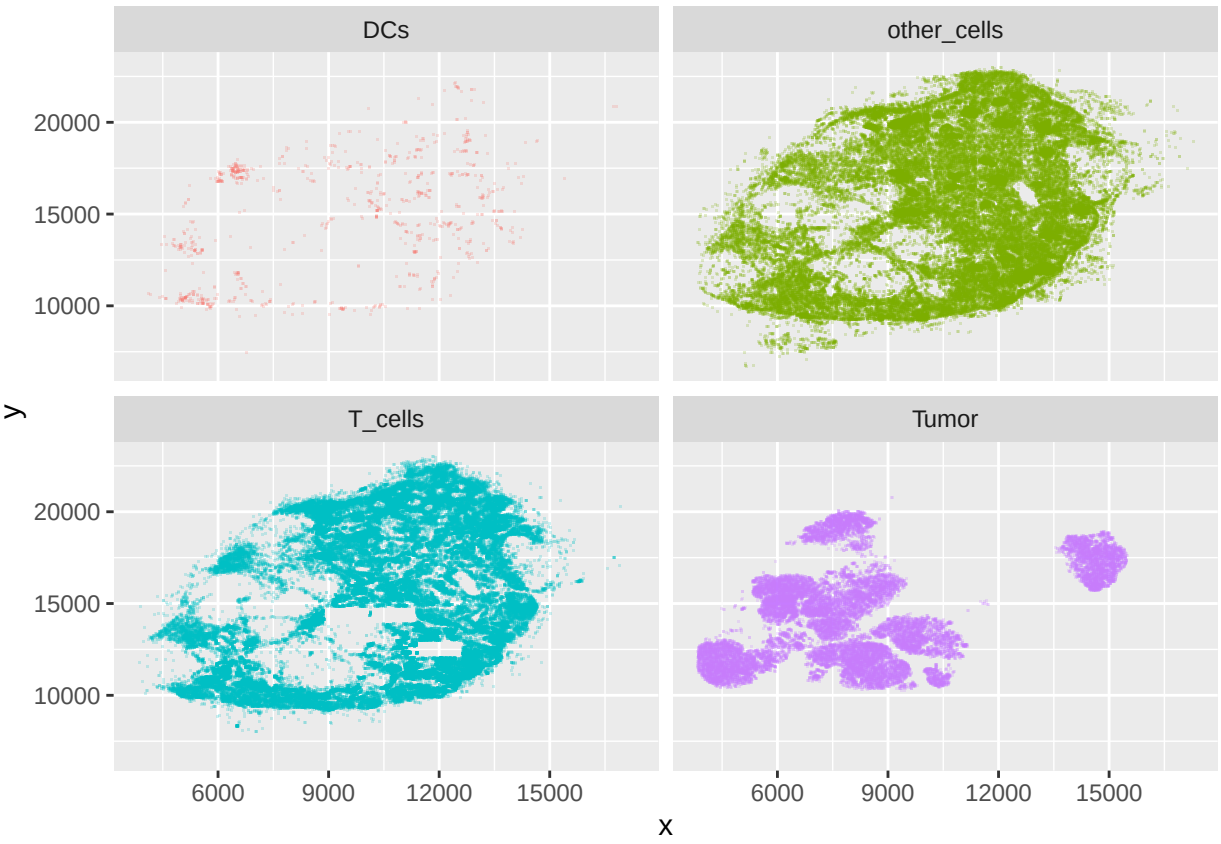
```
## # A tibble: 6 x 3
##       x     y class
##   <dbl> <dbl> <fct>
## 1  6416 10263 DCs
## 2  6445 10336 DCs
## 3  6448 10457 DCs
## 4  6462 10457 DCs
## 5  6600 10933 DCs
## 6  6629 10971 DCs
```

I think that each row represents a cell, with "x" and "y" giving the cell's position in the image.

```
# Without facetting
brcalymphnode %>%
  ggplot(aes(x = x, y = y, color = class)) +
  geom_point(shape = ".", alpha = 0.2)
```




```
# With facetting
brcalymphnode %>%
  ggplot(aes(x = x, y = y, color = class)) +
  geom_point(shape = ".", alpha = 0.2) +
  facet_wrap(~ class) +
  theme(legend.position = "none")
```



You can count the number of cells by each type:

```
brcalymphnode %>%
  group_by(class) %>%
  count()
```

```
## # A tibble: 4 x 2
## # Groups:   class [4]
##   class            n
##   <fct>        <int>
## 1 DCs            878
## 2 other_cells  77081
## 3 T_cells     103681
## 4 Tumor        27822
```

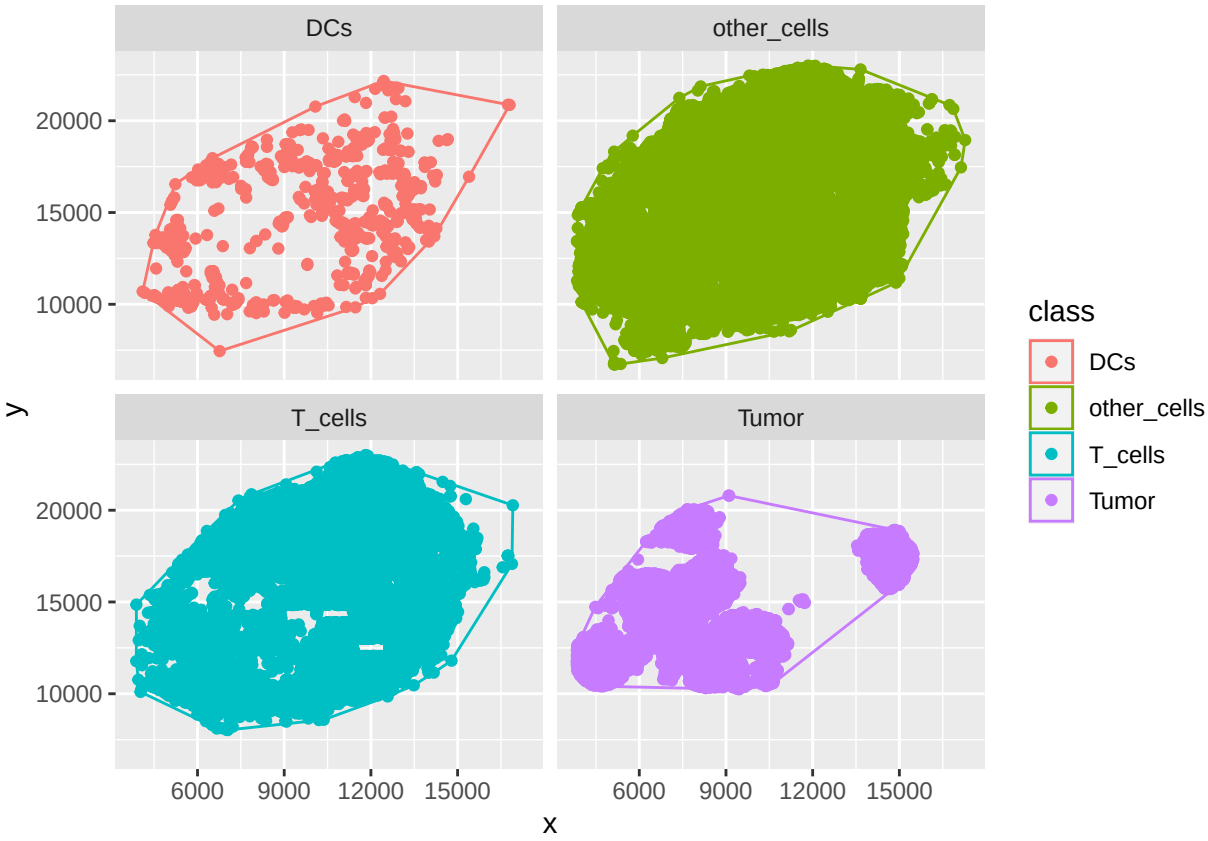
The book shows one way to get a convex hull for the data, using the `convhulln` package from the `geometry` library. You can also use the function `chull`, which comes with base R. For example, you could get (and plot) separate hulls for each class of cells with:

```
hull <- brcalymphnode %>%
  group_by(class) %>%
  slice(chull(x = x, y = y))
hull
```

```
## # A tibble: 92 x 3
## # Groups:   class [4]
##        x     y class
##    <dbl> <dbl> <fct>
##  1 14195 13706 DCs
```

```
##  2 12318 10568 DCs
##  3 12049 10332 DCs
##  4 11465  9834 DCs
##  5  6766  7449 DCs
##  6  4103 10698 DCs
##  7  4473 13342 DCs
##  8  4545 13780 DCs
##  9  5234 16544 DCs
## 10  6517 17963 DCs
## # ... with 82 more rows
```

```
ggplot() +
  geom_point(data = brcalymphnode,
             aes(x = x, y = y, color = class)) +
  geom_polygon(data = hull,
               aes(x = x, y = y, color = class),
               fill = NA) +
  facet_wrap(~ class)
```



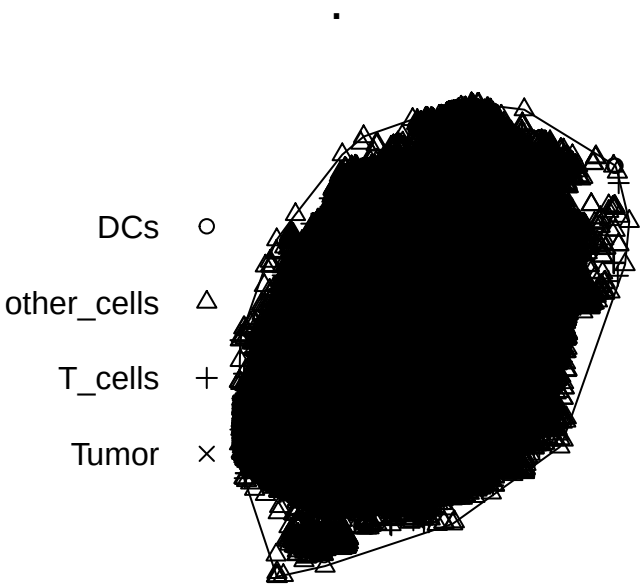
For some of the rest of the analysis in this chapter, they ask you to put the data into a object with a "ppp" class, which is defined in the `spatstat` package. You can do that with:

```
# install.packages("spatstat")
library("spatstat")

# Make a hull for all the cells
full_hull <- brcalymphnode %>%
  slice(chull(x = x, y = y)) %>%
```

```
  # For ppp, it needs to be in counterclockwise order, so we
  # need to flip the order of the rows
  mutate(n = 1:n()) %>%
  arrange(desc(n)) %>%
  select(x, y) %>%
  as.matrix()

ln <- ppp(x = brcalymphnode$x, y = brcalymphnode$y,
          marks = brcalymphnode$class,
          xrange = range(brcalymphnode$x),
          yrange = range(brcalymphnode$y),
          # Add in the hull
          poly = full_hull)

ln
```

```
## Marked planar point pattern: 209462 points
## Multitype, with levels = DCs, other_cells, T_cells, Tumor
## window: polygonal boundary
## enclosing rectangle: [3839, 17276] x [6713, 23006] units
```

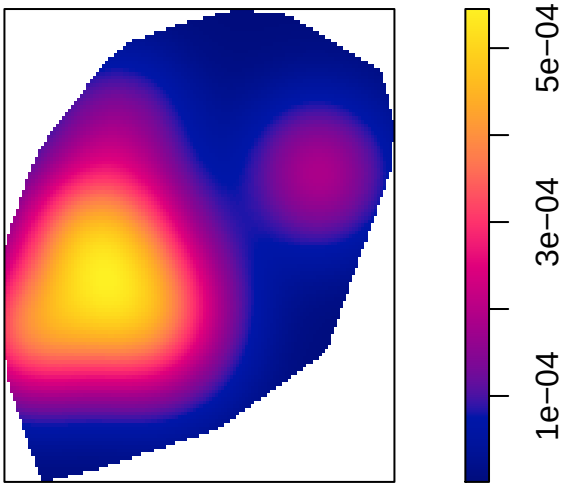
```
ln %>%
  plot()
```



You can use the `density` method (check `?density.ppp` for the helpfile) to compute a kernal smoothed intensity function from the patterns of points that we observed:

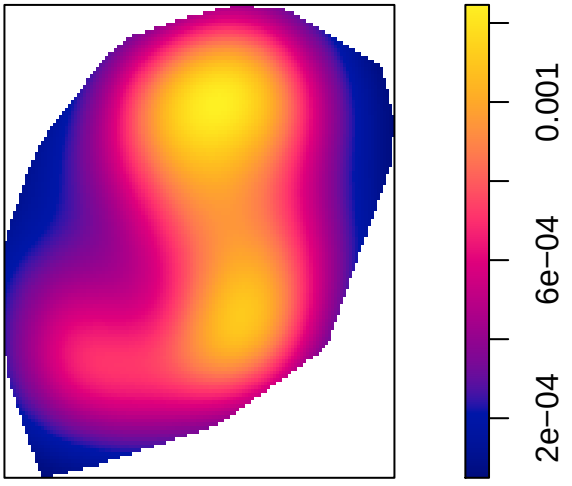
```
ln %>%
  # Limit to the cells that are tumor cells
  subset(marks == "Tumor") %>%
  density(edge = TRUE, diggle = TRUE) %>%
  plot()
```

.




```
# Now check T cells
ln %>%
  # Limit to the cells that are T cells
  subset(marks == "T_cells") %>%
  density(edge = TRUE, diggle = TRUE) %>%
  plot()
```

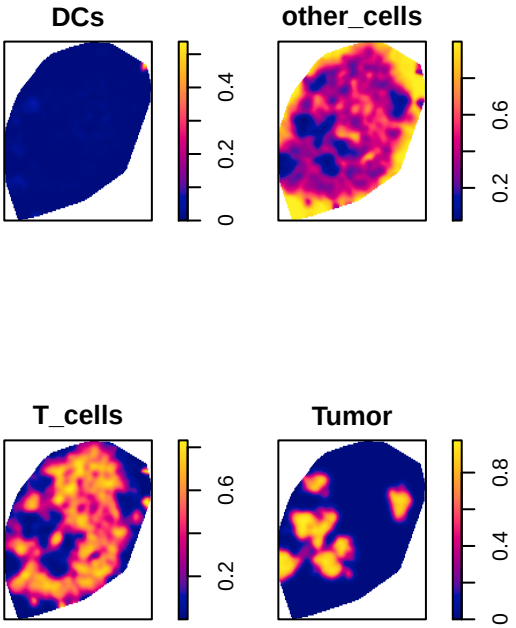
.



There is also a `relrisk` method for `ppp` objects, which estimates the probability of each type of point at each location in the image:

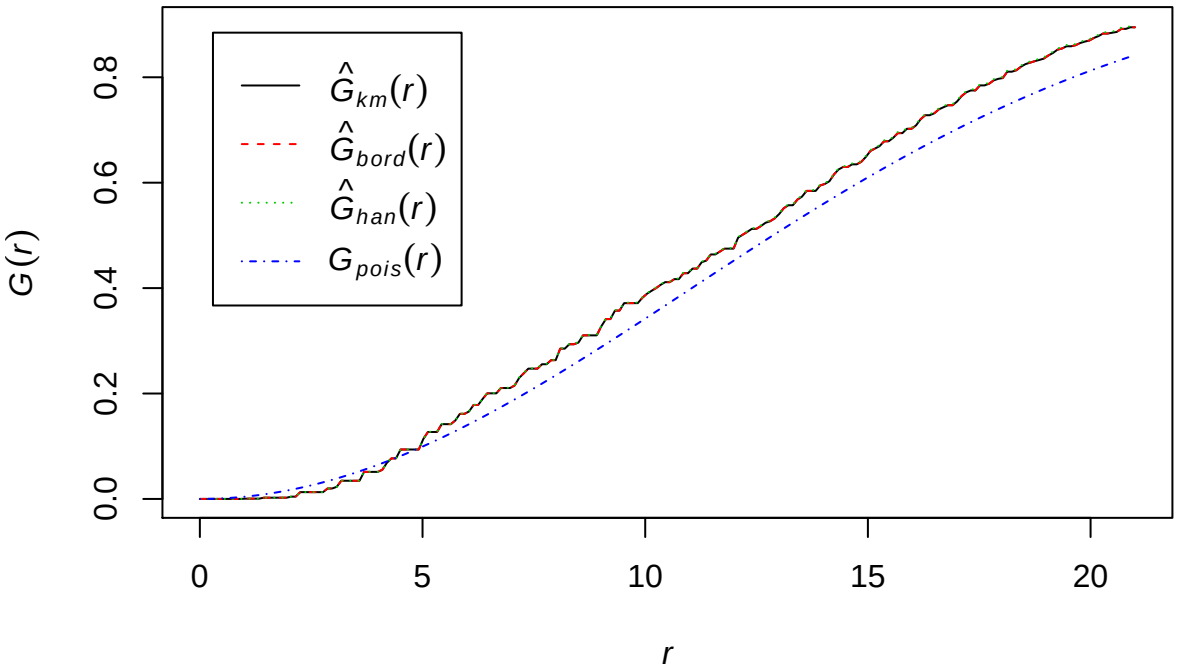
```
ln %>%
  relrisk(sigma = 250) %>%
  plot()
```

.



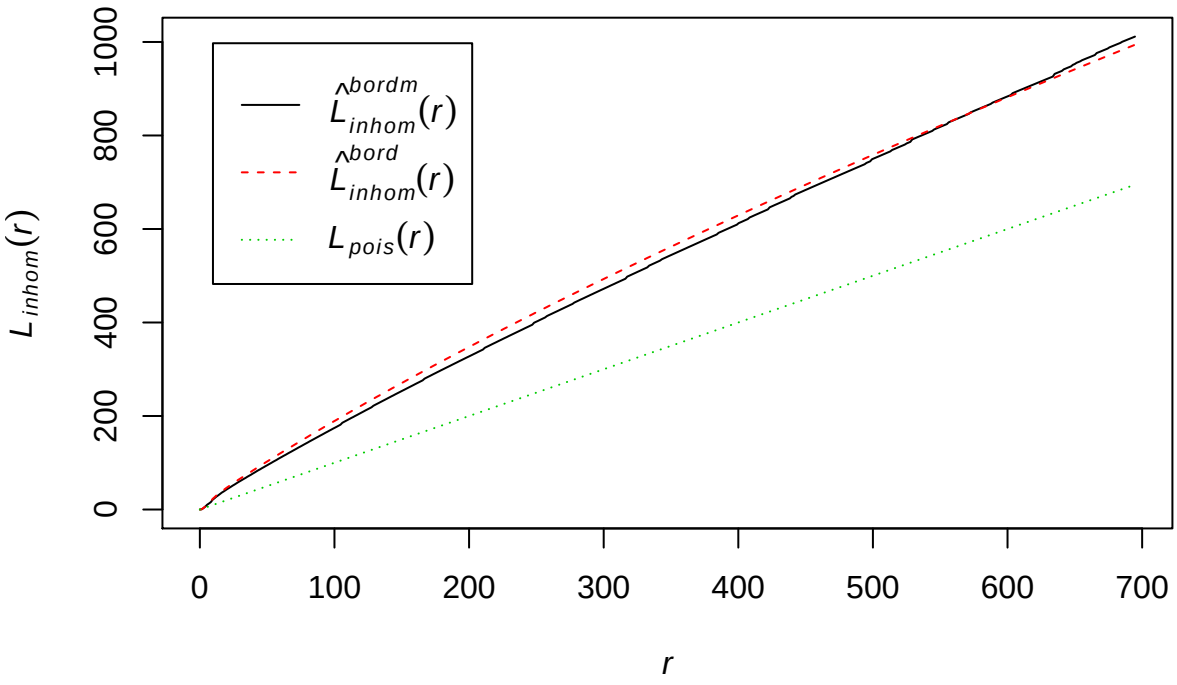
You can evidently use `Gest` to estimate the cumulative distribution function of the distance from any random point in the spatial process to its nearest neighbor. This includes fits based on the assumption that it's a homogeneous Poisson process, and so looking at the plot can help you determine if this assumption is reasonable for the data you observed.

```
ln %>%
  Gest() %>%
  plot()
```



You can also estimates the L function for the data, using `Linhom` (this takes a minute or two to run):

```
ln %>%
  subset(marks == "T_cells") %>%
  Linhom() %>%
  plot()
```



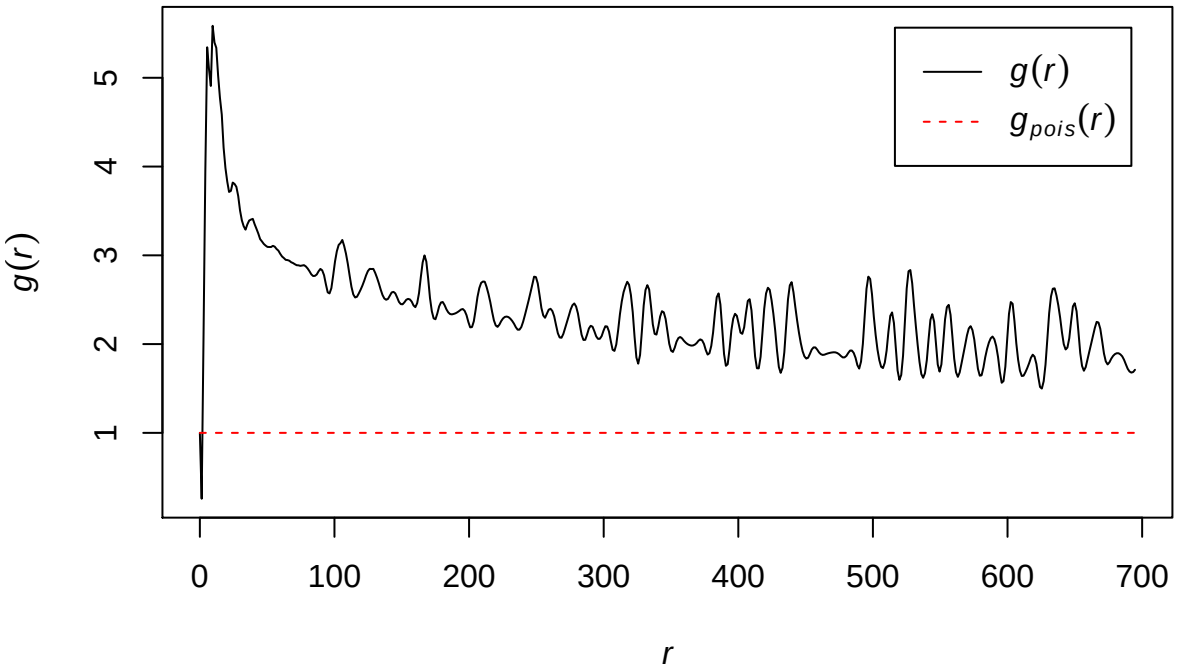
To interpret this result, you're comparing to the green line, which is for the null model that the points (T

cells) are located randomly throughout the image (at least, within the convex hull area that we identified).

Since the lines observed for this data (the red and black ones) are above the green line, it indicates that there is evidence in this data of spatial clustering—in other words, that the T cells are likely to be closer to each other than you'd expect if they were randomly spaced around the image.

Finally, you can estimate the pair correlation function for the data, using the `Kinhom` function (which also takes a minute to run):

```
ln %>%
  subset(marks == "T_cells") %>%
  Kinhom() %>%
  pcf() %>%
  plot()
```



Since the line estimated from the observed data (the black line) is lower than the null model line (red line) at very low r values (distances between cells), it means that cells are rarely *super* close together compared to in a Poisson process. This makes sense, because cells have some physical space they need. However, once you get past these really small r values, the black line is much higher than the red line—this shows evidence that T cells tend to cluster together with each other more than you'd expect if they were following a random Poisson process in how they were placed around the image.