Bioimaging1

March 26, 2020

1 Lecture 19: Bioimaging I

CBIO (CSCI) 4835/6835: Introduction to Computational Biology

1.1 Overview and Objectives

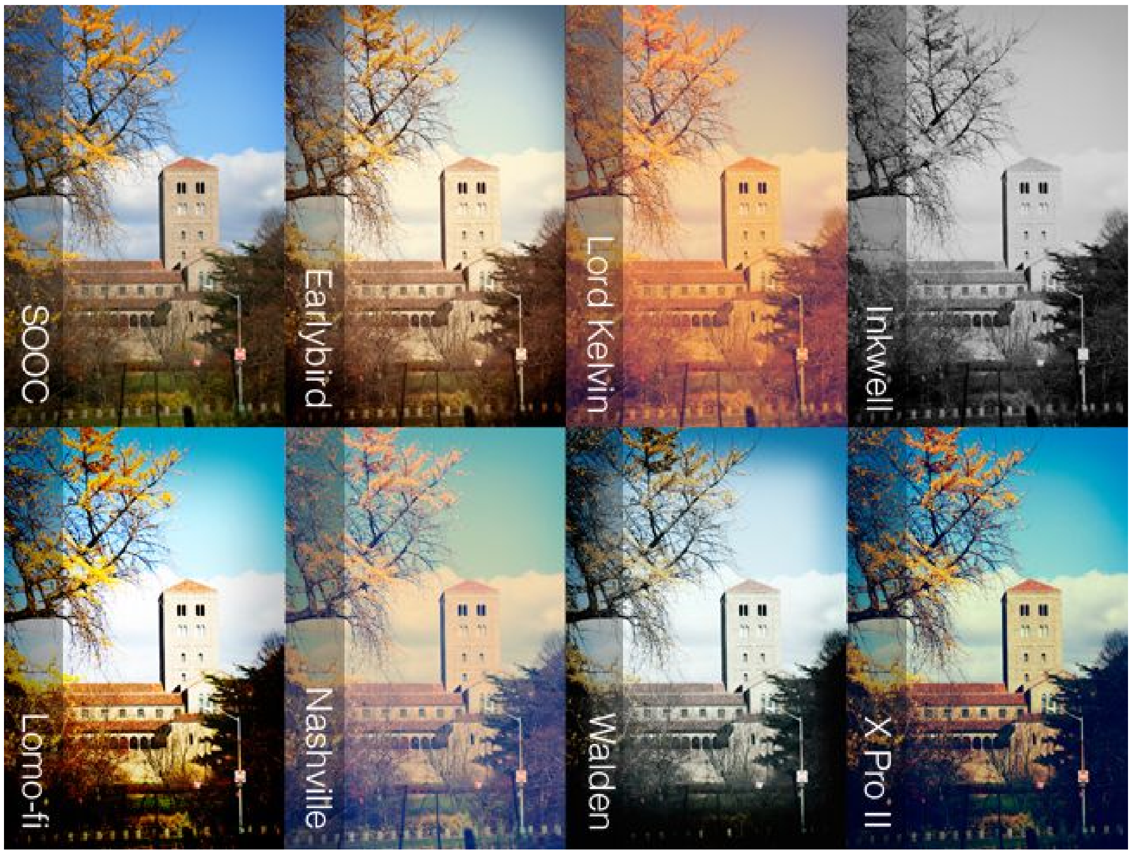
Now that we've covered the basics of computer vision, let's look at how this can be applied in a biological setting with some in-depth examples of how to analyze biological images. By the end of this lecture, you should be able:

- Explain convolutional filters and how they perform operations such as blurring, sharpening, and edge finding
- Implement erosion and dilation and define them in the context of image processing
- Provide an end-to-end example of how to segment an image (for example: count cells)

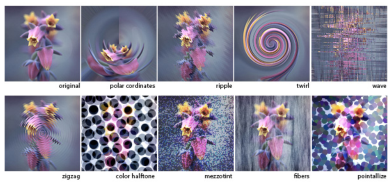
1.2 Part 1: Filters

What does "filter" mean to you?

Instagram?



Photoshop?



Both are technically “yes”, as they use the same underlying principle.

A filter applies a *convolution kernel* to an image.

Although this sounds fancy, this is just a generalization of a very simple idea: rather than applying a specific function to one pixel, this uses the pixel’s surrounding neighborhood to apply a function.

The kernel is represented by an $n \times n$ matrix where the target pixel is in the center (so n needs to be an odd number).

The output of the filter is the sum of the products of the matrix elements with the corresponding pixels they overlap.

Confused?

Here’s what it looks like. The image is the bigger 5×5 matrix; the filter is the smaller 3×3 matrix:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

https://github.com/eds-uga/cbio4835-fa18/blob/master/lectures/Bioimaging1/convolution.gif

1.2.1 Convolutional Kernels

A convolution is basically a multiplication: the smaller matrix is multiplied element-wise with the overlapped entries of the larger matrix. These products are then all summed together into a new value for the pixel at the very center of the filter.

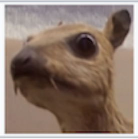
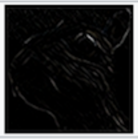
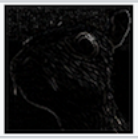
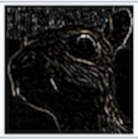
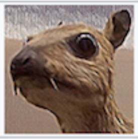
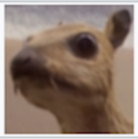
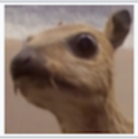
Then the filter is moved and the process repeats. This is true for any and all filters (or *kernels*).

What creates the specific effect, then–edge finding, blurring, sharpening, and so on–is the specific numbers in the filter. As such, there are some common filters:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Identity Blur Edge Detection

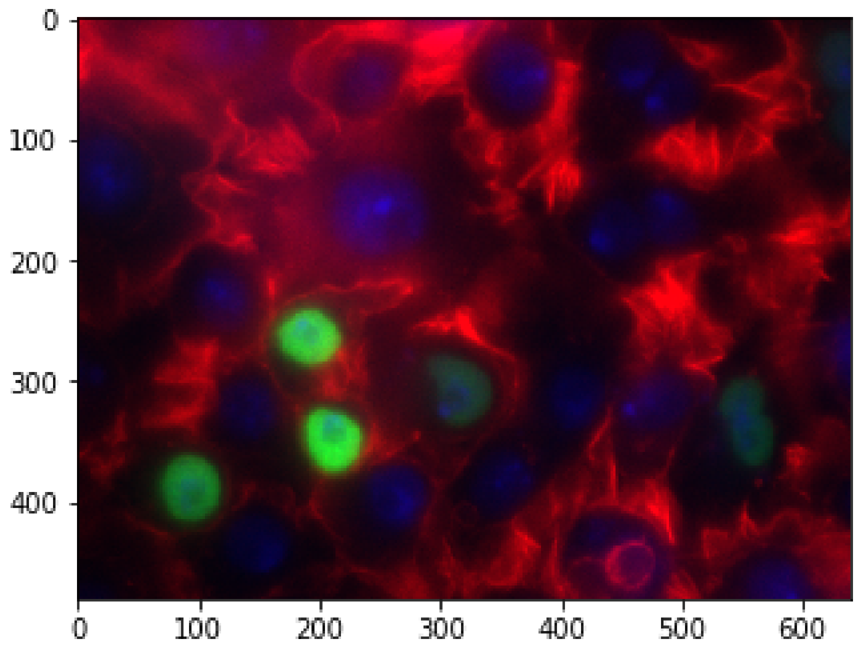
Wikipedia has a whole page on common convolutional filters.

Operation	Kernel	Image result
Identity	$\begin{bmatrix}0&0&0\\0&1&0\\0&0&0\end{bmatrix}$	
Edge detection	$\begin{bmatrix}1&0&-1\\0&0&0\\-1&0&1\end{bmatrix}$	
	$\begin{bmatrix}0&1&0\\1&-4&1\\0&1&0\end{bmatrix}$	
	$\begin{bmatrix}-1&-1&-1\\-1&8&-1\\-1&-1&-1\end{bmatrix}$	
Sharpen	$\begin{bmatrix}0&-1&0\\-1&5&-1\\0&-1&0\end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$	
Gaussian blur 3 × 3 (approximation)	$\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}$	

Of course, you don't have to design the filter and code up the convolution yourself (though you could!). Most image processing packages have default versions of these filters included.

```
[1]: %matplotlib inline
     import matplotlib.pyplot as plt
     import matplotlib.image as mpimg
     from PIL import Image
     from PIL import ImageFilter

     img = Image.open("ComputerVision/image1.png")
     plt.imshow(img)
```

```
[1]: <matplotlib.image.AxesImage at 0x11ded59e8>
```



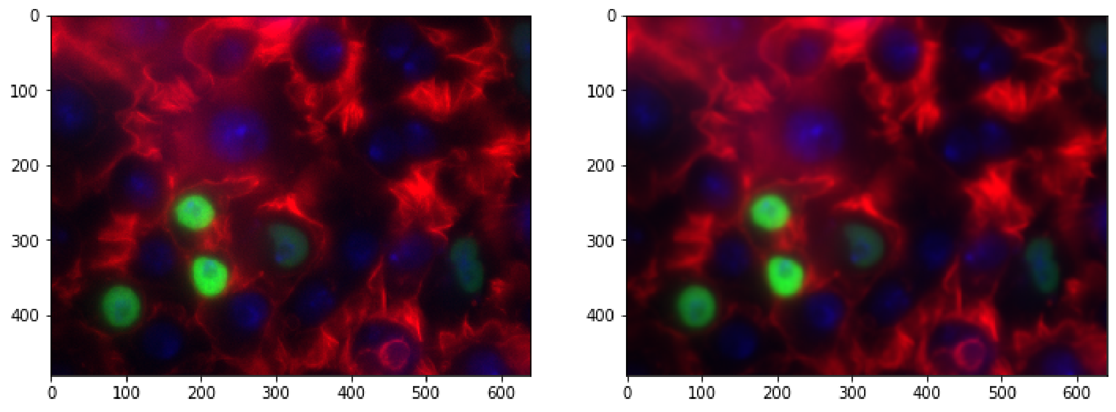
Some of the PIL built-in filters include:

- BLUR
- CONTOUR
- DETAIL
- EDGE_ENHANCE
- EDGE_ENHANCE_MORE
- EMBOSS
- FIND_EDGES
- SMOOTH
- SMOOTH_MORE
- SHARPEN

Here's BLUR:

[2]:
```
import numpy as np

blurred = img.filter(ImageFilter.BLUR)
f = plt.figure(figsize = (12, 6))
f.add_subplot(1, 2, 1)
plt.imshow(np.array(img))
f.add_subplot(1, 2, 2)
plt.imshow(np.array(blurred))
```

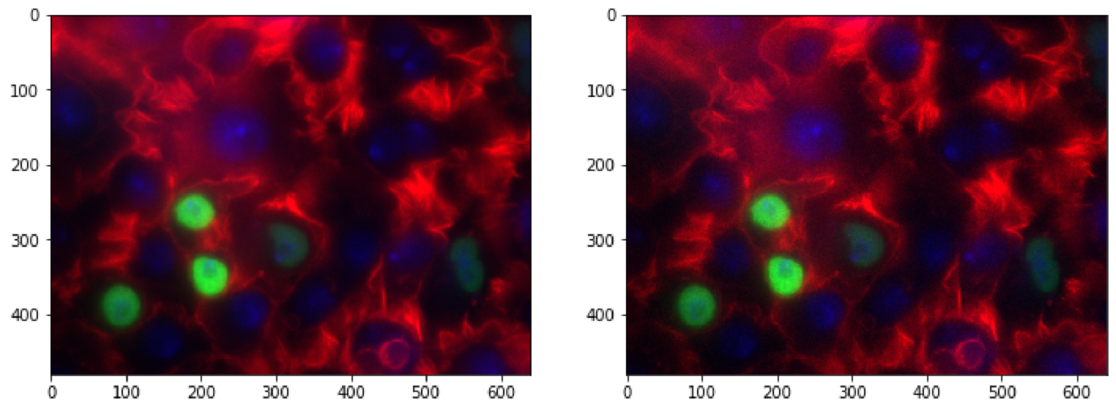
[2]:
```
<matplotlib.image.AxesImage at 0x123a96748>
```



Here's SHARPEN:

```
[3]: sharpened = img.filter(ImageFilter.SHARPEN)
     f = plt.figure(figsize = (12, 6))
     f.add_subplot(1, 2, 1)
     plt.imshow(np.array(img))
     f.add_subplot(1, 2, 2)
     plt.imshow(np.array(sharpened))
```

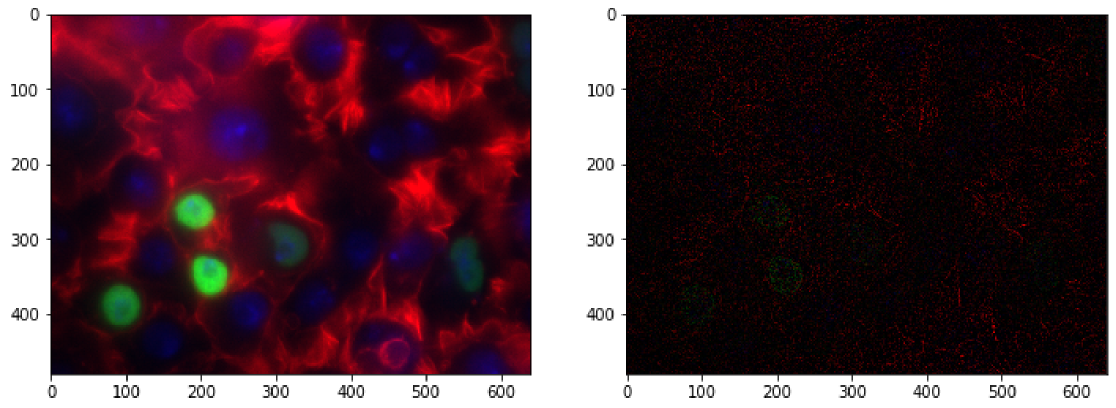
```
[3]: <matplotlib.image.AxesImage at 0x110164e80>
```



And FIND_EDGES (more on this later):

```
[4]: edges = img.filter(ImageFilter.FIND_EDGES)
     f = plt.figure(figsize = (12, 6))
     f.add_subplot(1, 2, 1)
     plt.imshow(np.array(img))
     f.add_subplot(1, 2, 2)
     plt.imshow(np.array(edges))
```

```
[4]: <matplotlib.image.AxesImage at 0x12436c1d0>
```



There are other filters that don't quite abide by the "multiply every corresponding element and then sum the products" rule of convolution.

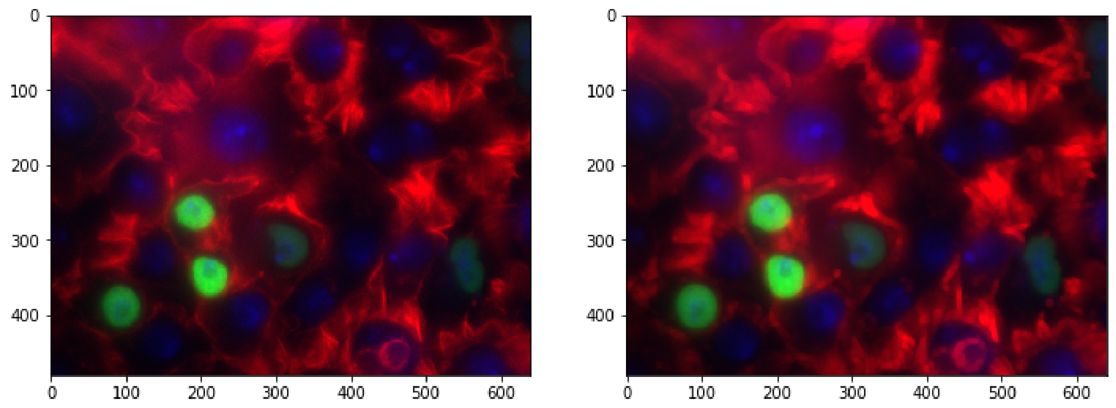
In deep learning parlance, these filters are known as "pooling" operators, because while you still have a filter that slides over the image, you *choose one* of the pixel values within that filter instead of computing a function over *all the pixel values*.

As common examples, we have - Median filters: picking the median pixel value in the filter range - Max and min filters: picking the maximum (max-pool) or minimum (min-pool) pixel value in the filter range

Here's max-pooling:

```
[5]: max_pool = img.filter(ImageFilter.MaxFilter(5))  # This means a 5x5 filter
     f = plt.figure(figsize = (12, 6))
     f.add_subplot(1, 2, 1)
     plt.imshow(np.array(img))
     f.add_subplot(1, 2, 2)
     plt.imshow(np.array(max_pool))
```

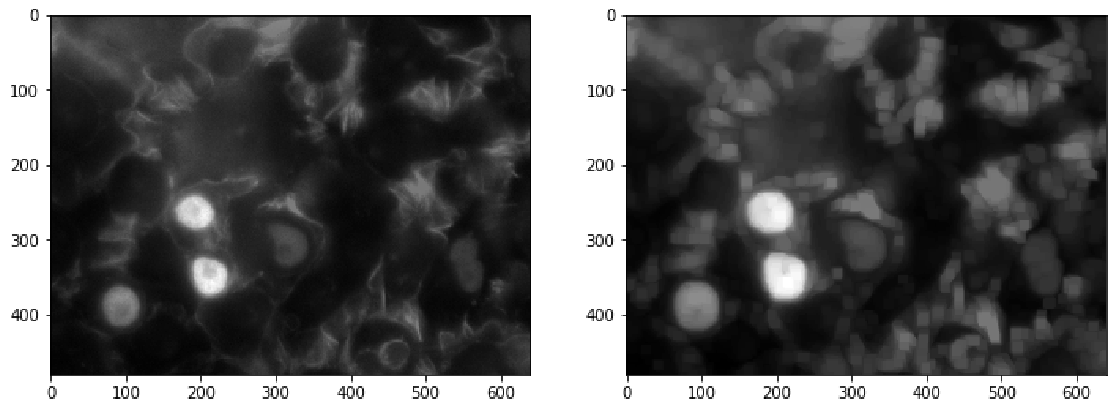
```
[5]: <matplotlib.image.AxesImage at 0x1243de908>
```



(just to help things, let's convert to grayscale and try that again)

```
[6]: img = img.convert("L")

     # Same code as before.
     max_pool = img.filter(ImageFilter.MaxFilter(11))  # This means an 11x11 filter
     f = plt.figure(figsize = (12, 6))
     f.add_subplot(1, 2, 1)
     plt.imshow(np.array(img), cmap = "gray")
     f.add_subplot(1, 2, 2)
     plt.imshow(np.array(max_pool), cmap = "gray")
```

```
[6]: <matplotlib.image.AxesImage at 0x1245e0da0>
```

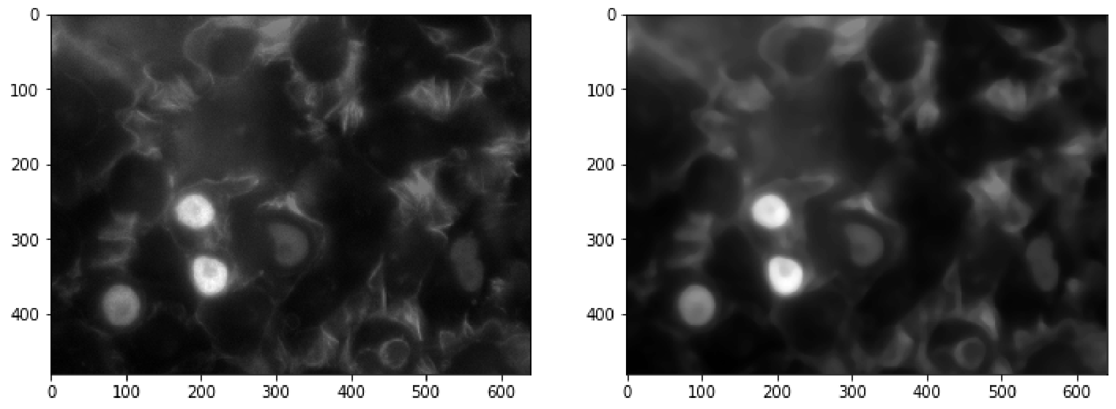


How about median pooling?

```
[7]: median_pool = img.filter(ImageFilter.MedianFilter(11))  # This means an 11x11␣
     ↪filter
```

```
f = plt.figure(figsize = (12, 6))
f.add_subplot(1, 2, 1)
plt.imshow(np.array(img), cmap = "gray")
f.add_subplot(1, 2, 2)
plt.imshow(np.array(median_pool), cmap = "gray")
```

```
[7]: <matplotlib.image.AxesImage at 0x1250e5e80>
```



1.2.2 So...

What are the benefits of filtering and pooling? Why would something like a blur filter or a median filter be nice to use?

These filters clear out a lot of noise.

- By considering neighborhoods of pixels, you're able to automatically dampen tiny flecks of fluorescence that don't correspond to anything, because they'll be surrounded by black.
- On the other hand, filters like blur and median won't [strongly] affect large sources of fluorescence, since the neighborhoods of any given pixel will also be emitting lots of signal.

Protip: median filters are awesome for getting rid of tiny little specks of light you don't want, while maintaining sharp edges between objects (unlike Gaussian blur filters).

1.3 Part 2: Edges

Edge-finding is an important aspect of bioimaging and image processing in general.

Edges essentially denote "image derivatives" (and are essentially calculated as such!). Edges are formed by the image pixels changing suddenly.

Edges delineate objects. The question isn't necessarily where the edges of the object are, but *which* edges belong to the object you're interested in.

1.3.1 Canny Edge Detector

One of the most popular algorithms for finding edges in an image is the *Canny Edge Detector*.

The Canny edge detector works in three distinct phases (don't worry, you won't have to implement any of these):

1: It runs a filter (!) over the image, using the Gaussian formulation, to generate a filtered image. However, this Gaussian filter is a special variant–it actually represents the *first derivative* of a Gaussian. Running this filter over the image *essentially* computes the image derivatives at each pixel.

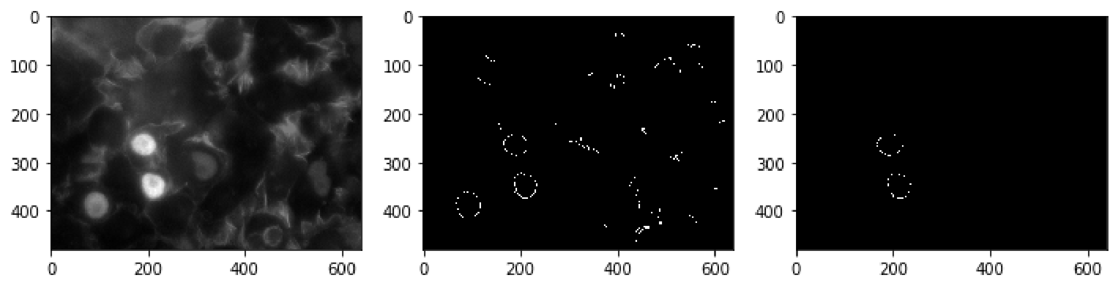
2: The filter generates a lot of "candidate" edges that have to be pruned down; the edges are reduced until they're only 1 pixel thick.

3: Finally, a threshold is applied: a pixel is considered part of an "edge" if its derivative (as computed in step 1) exceeds a certain value. The higher the threshold, the larger the pixel derivative has to be to be considered an edge.

In action, Canny edge detectors looks something like this:

[8]:
```python
import skimage.feature as feature

img = np.array(img)
canny_small = feature.canny(img, sigma = 1)
canny_large = feature.canny(img, sigma = 3)

f = plt.figure(figsize = (12, 6))
f.add_subplot(1, 3, 1)
plt.imshow(img, cmap = "gray")
f.add_subplot(1, 3, 2)
plt.imshow(canny_small, cmap = "gray")
f.add_subplot(1, 3, 3)
plt.imshow(canny_large, cmap = "gray")
```

[8]:
```
<matplotlib.image.AxesImage at 0x111a197f0>
```



- Note how many fewer edges there are with `sigma = 3` than `sigma = 1`.

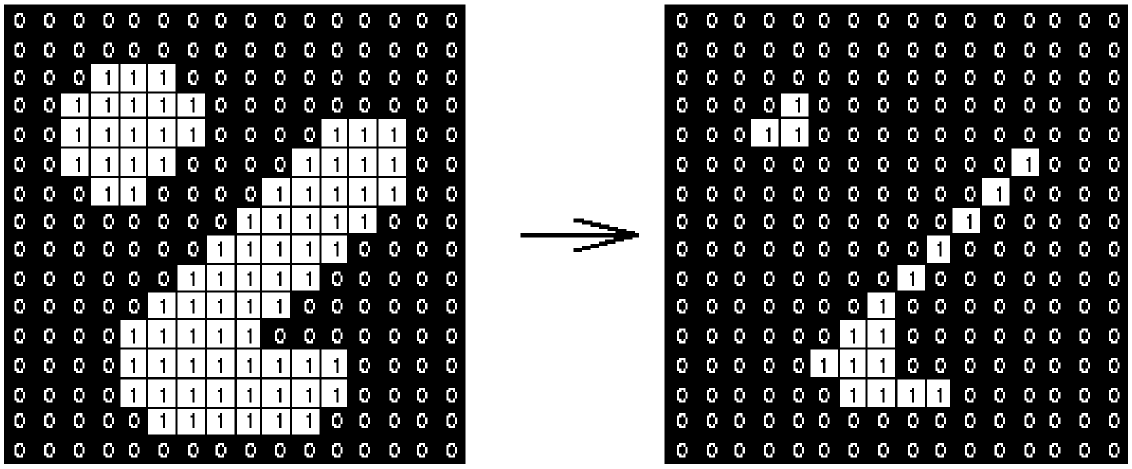
- By making this sigma larger, you're upweighting neighboring pixels in the Gaussian derivative filter.
- This makes it harder for the middle pixel (the one under consideration as an edge, or not) to exceed the "threshold" from step 3 to be considered an edge.
- The `sigma` argument essentially equates to the "width" of the filter–smaller width, smaller neighborhood; therefore, more representation by the middle pixel.

1.3.2 Erosion and Dilation

Another common noise-reducing step is some combination of *erosion* and *dilation*.

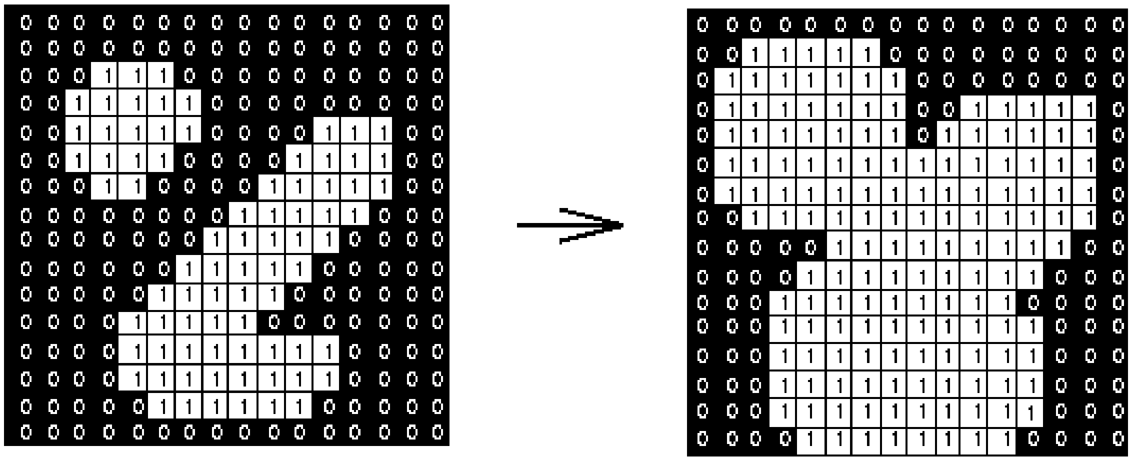
These don't find edges per se, but rather they modify the edges.

Erosion will move along objects and erode them by 1 pixel (or more).



This has the benefit of utterly wiping out objects that are very small–these are usually noise anyway.

Dilation is the inverse operation: it will move along objects, padding their edges by 1 pixel (or more).

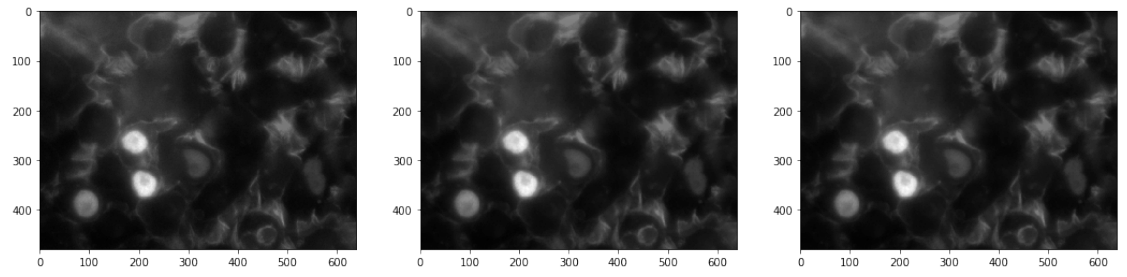


This has the effect of smoothing object edges–jagged edges are filled in.

These two effects are often used in tandem to smooth out potentially rough images.

[9]:
```
import skimage.morphology as morphology

de1 = morphology.dilation(morphology.erosion(img))
de2 = morphology.dilation(morphology.erosion(de1))

f = plt.figure(figsize = (18, 9))
f.add_subplot(1, 3, 1)
plt.imshow(img, cmap = "gray")
f.add_subplot(1, 3, 2)
plt.imshow(de1, cmap = "gray")
f.add_subplot(1, 3, 3)
plt.imshow(de2, cmap = "gray")
```

[9]:
```
<matplotlib.image.AxesImage at 0x1c284c5470>
```



Hmm, not an obvious effect in the full grayscale.

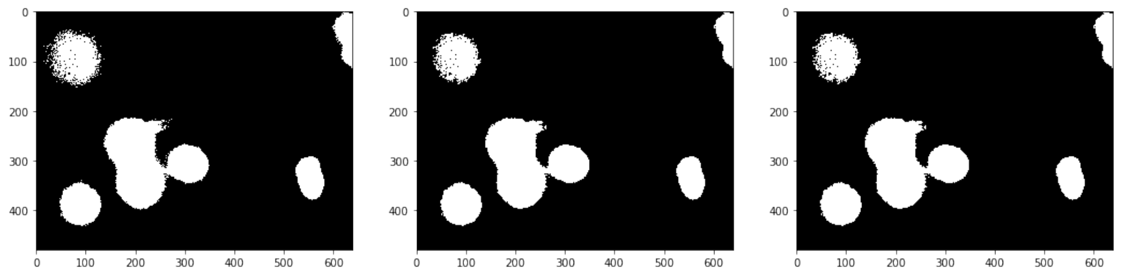
Let's try again, taking a page from yesterday's lecture–specifically, let's use a single channel and threshold the image.

[10]:
```
import scipy.ndimage as ndimg

img = ndimg.imread("ComputerVision/image1.png")
hsp = img[:, :, 1]   # Green channel - heat shock protein
hsp_bin = hsp > np.mean(hsp)   # Median threshold.
```

```
/opt/python/lib/python3.6/site-packages/ipykernel_launcher.py:3:
DeprecationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0.
Use ``matplotlib.pyplot.imread`` instead.
  This is separate from the ipykernel package so we can avoid doing imports
until
```

[11]:
```
# Now we'll do the same code as before.
de1 = morphology.binary_dilation(morphology.binary_erosion(hsp_bin))
de2 = morphology.binary_dilation(morphology.binary_erosion(de1))

f = plt.figure(figsize = (18, 9))
f.add_subplot(1, 3, 1)
plt.imshow(hsp_bin, cmap = "gray")
f.add_subplot(1, 3, 2)
plt.imshow(de1, cmap = "gray")
f.add_subplot(1, 3, 3)
plt.imshow(de2, cmap = "gray")
```

[11]:
```
<matplotlib.image.AxesImage at 0x1c28937be0>
```



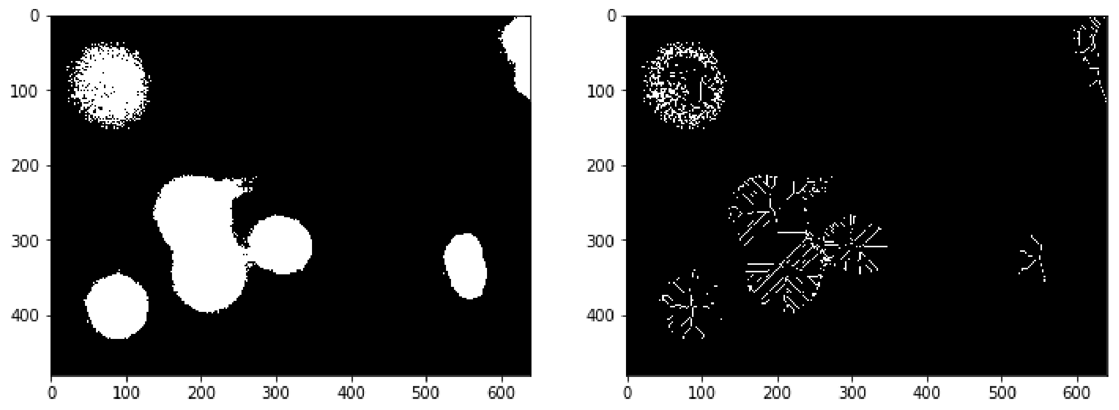
1.3.3 Skeletonization

You can go a step further. If you're not interested in the overall *shape* of the objects, but just want some measure of how many and where, you can use skeletonization.

This essentially performs the erosion operation over and over until each object is only 1 pixel wide.

[12]:
```
skeleton = morphology.skeletonize(hsp_bin)

f = plt.figure(figsize = (12, 6))
f.add_subplot(1, 2, 1)
plt.imshow(hsp_bin, cmap = "gray")
f.add_subplot(1, 2, 2)
plt.imshow(skeleton, cmap = "gray")
```

[12]:
```
<matplotlib.image.AxesImage at 0x1c293e3cf8>
```

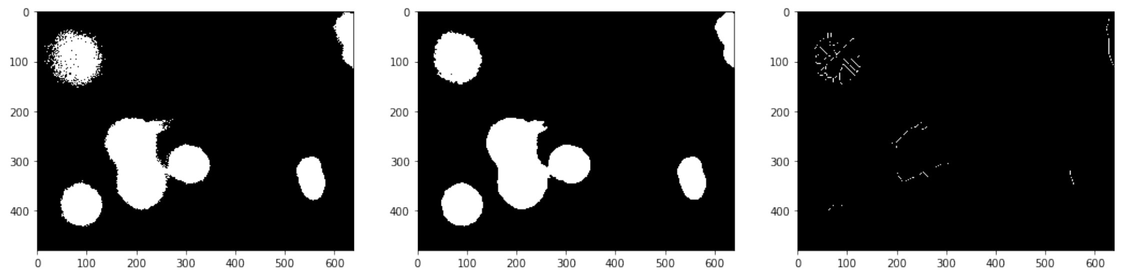


What went wrong, do you think? Why didn't we just get 1-2 lines per object?

We should try to eliminate all those specks!

[13]:
```
import skimage.filters as filters

bin_median = filters.median(hsp_bin, morphology.square(5))
bin_median[bin_median == 255] = 1
skeleton = morphology.skeletonize(bin_median)

f = plt.figure(figsize = (18, 9))
f.add_subplot(1, 3, 1)
plt.imshow(hsp_bin, cmap = "gray")
f.add_subplot(1, 3, 2)
plt.imshow(bin_median, cmap = "gray")
f.add_subplot(1, 3, 3)
plt.imshow(skeleton, cmap = "gray")
```

[13]:
```
<matplotlib.image.AxesImage at 0x1c286aaf28>
```



1.3.4 Convex Hull

The inverse operation of skeletonization is the to find the *convex hull* of objects in the image. This is basically a fancy way of saying:

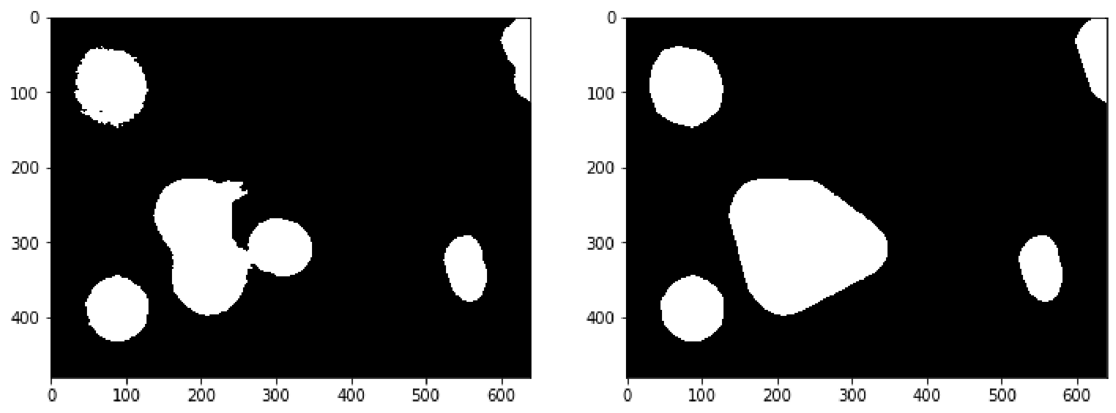
- Draw the simplest *n*-gon you can to encompass the entire object.

This is a very good way of deriving *masks* for complex!

Let's give ourselves a bit of a head start by using that binarized median filtered image:

[14]:
```
convex_hulls = morphology.convex_hull_object(bin_median)

f = plt.figure(figsize = (12, 6))
f.add_subplot(1, 2, 1)
plt.imshow(bin_median, cmap = "gray")
f.add_subplot(1, 2, 2)
plt.imshow(convex_hulls, cmap = "gray")
```

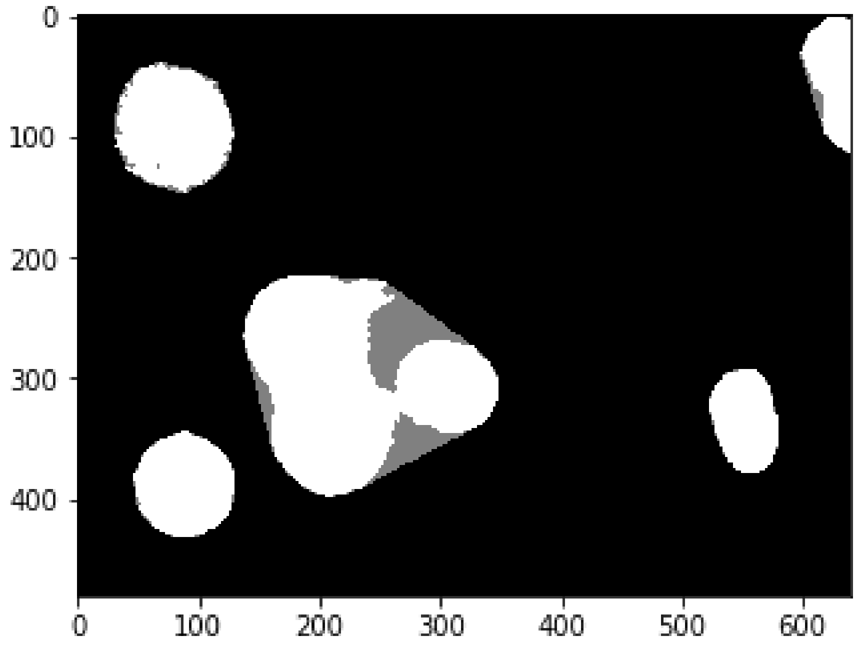
[14]:
```
<matplotlib.image.AxesImage at 0x1c28c0e860>
```



We can overlay the two plots to show how the convex hulls were drawn:

[15]:
```
import skimage
img_float = skimage.img_as_float(convex_hulls.copy())
img_float[bin_median == 1] = 2

plt.imshow(img_float, cmap = "gray")
```

[15]:
```
<matplotlib.image.AxesImage at 0x1c29c05320>
```



1.4 Part 3: Segmentation

Ok–we've seen so far how to identify objects (kinda) in images. It'd be great to actually get their coordinates, extract them, and learn something about their shape and size and general morphology.

This is the process of *segmentation*: explicitly developing a mask that pulls out the objects you're interested in.

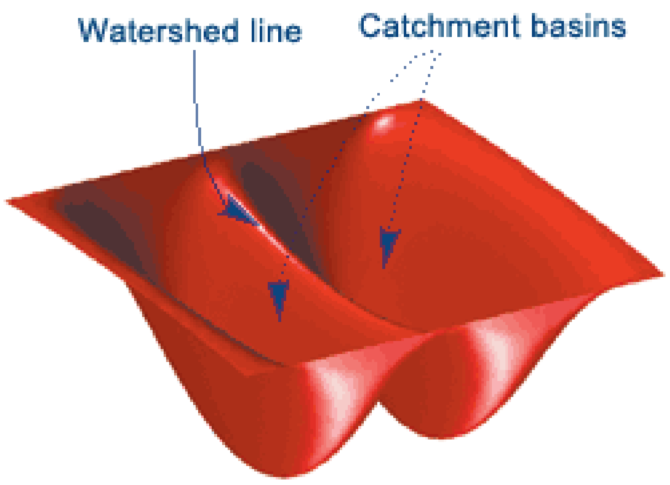
The convex hulls are a good start, but that one blob near the middle combined a bunch of the cells.

(yes, we're purposely hobbling ourselves by not using the DAPI channel; for illustration purposes)

We can employ a specific segmentation to try and break that blob apart into multiple objects, ideally correpsonding with the blobs we see in the grayscale image.

1.4.1 Watershed Segmentation

This starts with the idea of treating your image as a contour map, where pixel intensity indicates elevation.

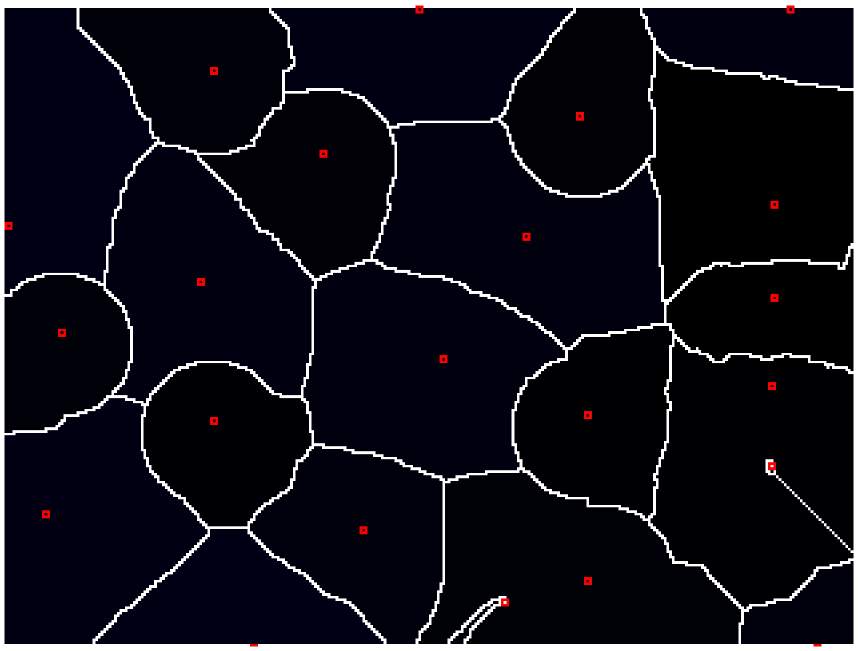


From the "lowest" parts of your image, you start filling your image with water.

Where water from each "basin" meets, that's a segmentation boundary.

Continue raising the water level of your image until you have fully segmented it.

This will give you an image with boundaries between all the objects.



"Wait a minute," you start, "that looks great and all, but how do you know where to start adding the water?"

Excellent question. Ideally, we'd like to place a "basin" (or "seed") inside each object. But that requires knowing where the objects are ahead of time, which kind of obviates the need for watershed in the first place.

Well, I kinda gave it away earlier–we're looking for pixels that have values at the extremes. These are known as *local maxima*.

We'd like to think that each object has a "center" that is bright; ideally, it's the brightest part of the object.

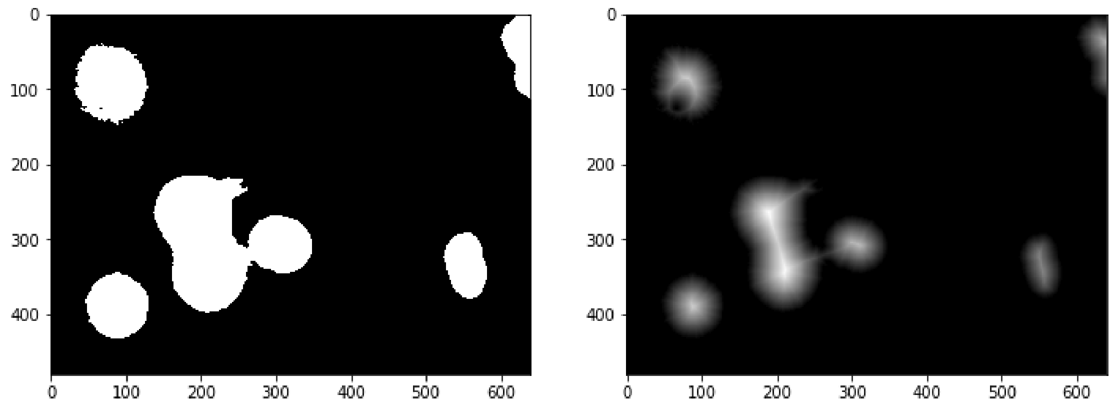
This isn't always the case, but we can kind of force it using a *distance map*. If we compute the distance of any point inside an object to its edge, the center should be the farthest, right?

1.4.2 Distance Maps

Let's first see about computing the distance map for our binarized median filtered image.

[16]:
```
distmap = ndimg.distance_transform_edt(bin_median)

f = plt.figure(figsize = (12, 6))
f.add_subplot(1, 2, 1)
plt.imshow(bin_median, cmap = "gray")
f.add_subplot(1, 2, 2)
plt.imshow(distmap, cmap = "gray")
```

[16]:
```
<matplotlib.image.AxesImage at 0x1c29e2b7b8>
```

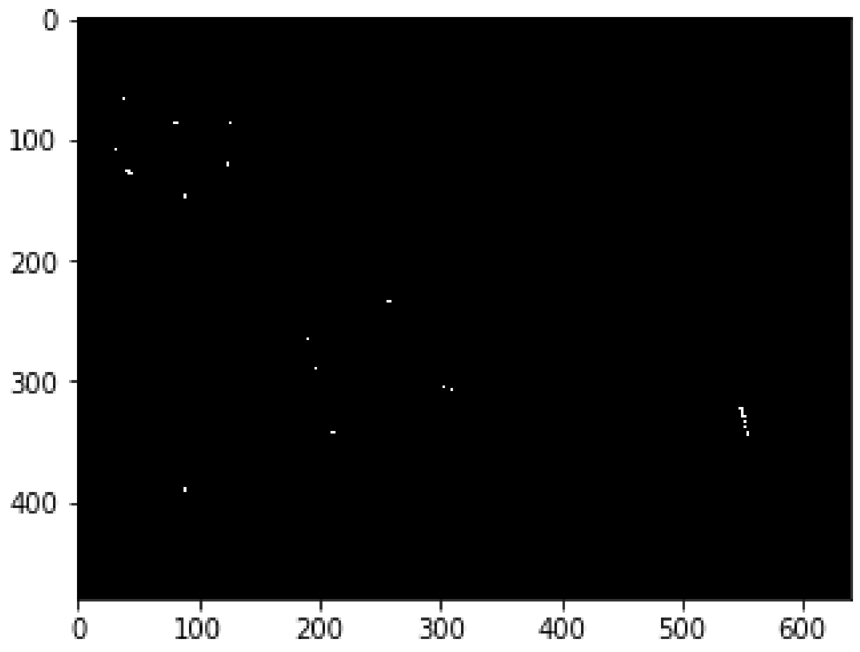


(yep, you could probably use skeletonization to achieve the same effect!)

Now, we'll find the coordinates of the local maxima and use those as seeds for watershed.

[17]:
```
maxima = feature.peak_local_max(distmap, indices = False, min_distance = 2,␣
    ↪labels = bin_median)
display = morphology.dilation(maxima)

plt.imshow(display, cmap = "gray")
```

[17]:
```
<matplotlib.image.AxesImage at 0x1c29ee5278>
```

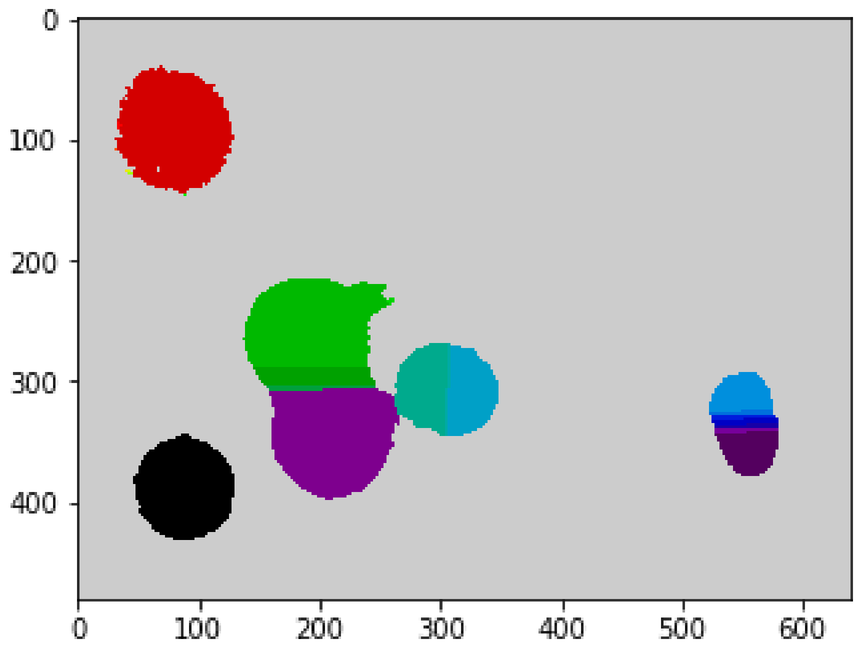


Now we'll use these coordinates as our starting points for watershed!

```
[18]: markers = ndimg.label(maxima)[0]
      labels = morphology.watershed(-distmap, markers, mask = bin_median)

      plt.imshow(labels, cmap = "nipy_spectral_r")
```

```
/opt/python/lib/python3.6/site-packages/skimage/util/arraycrop.py:175:
FutureWarning: Using a non-tuple sequence for multidimensional indexing is
deprecated; use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will
be interpreted as an array index, `arr[np.array(seq)]`, which will result either
in an error or a different result.
  cropped = np.array(ar[slices], order=order, copy=True)
```

```
[18]: <matplotlib.image.AxesImage at 0x1c2bc472e8>
```



It's certainly not perfect.

There are a lot of improvements that could be made, but this was just to illustrate how to get segmented regions.

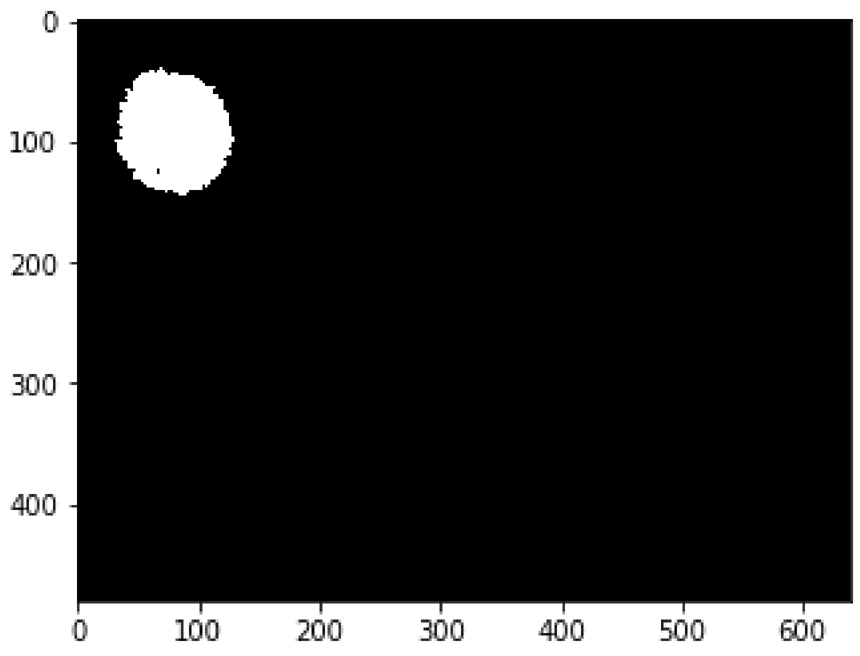
Once you have these regions, you can ask a lot of interesting questions about them:

[19]:
```
import skimage.measure as measure

regions = measure.regionprops(labels)
print(len(regions))   # 28 regions, each with some of the following properties:
```

```
28
```

[20]:
```
region1 = labels.copy()
region1[region1 != regions[1].label] = 0
plt.imshow(region1, cmap = "gray")

print(regions[1].area)
```

```
7601
```



[21]:
```
print(regions[1].centroid)
```

```
(92.7910801210367, 79.78897513485067)
```

[22]:
```
print(regions[1].major_axis_length)
print(regions[1].minor_axis_length)
```

```
104.25411574839266
93.36852033227642

/opt/python/lib/python3.6/site-packages/skimage/measure/_regionprops.py:250:
UserWarning: regionprops and image moments (including moments, normalized
moments, central moments, and inertia tensor) of 2D images will change from xy
coordinates to rc coordinates in version 0.16.
See http://scikit-
image.org/docs/0.14.x/release_notes_and_installation.html#deprecations for
details on how to avoid this message.
  warn(XY_TO_RC_DEPRECATION_MESSAGE)
/opt/python/lib/python3.6/site-packages/skimage/measure/_regionprops.py:260:
UserWarning: regionprops and image moments (including moments, normalized
moments, central moments, and inertia tensor) of 2D images will change from xy
coordinates to rc coordinates in version 0.16.
See http://scikit-
image.org/docs/0.14.x/release_notes_and_installation.html#deprecations for
details on how to avoid this message.
  warn(XY_TO_RC_DEPRECATION_MESSAGE)
```

```
[23]: print(regions[1].solidity)
```

```
0.9332105586249233
```

1.5 Review

In this lecture, we looked at some of the most common operations in bioimaging: edge detection, blurring/sharpening, and object finding and segmentation.

Most of this is done through the use of *convolutional filters*, though there are other ways, too.

Once you've found objects and generated masks, you can compute a lot of interesting properties about them (centroid, area, major and minor axis lengths).

All of this is in the `scikit-image` package. This is a well-designed, well-documented, easy-to-use imaging package in Python that comes automatically installed in any Anaconda Python version.

Next week, we'll wrap up imaging by going over OpenCV, the beast of computer vision!

1.6 Administrivia

- Has anyone started Assignment 5 yet? The earlier, the better!
- Any questions about the project?

1.7 Additional Resources

- scikit-image edge detection
- scikit-image segmentation
- scikit-image labeling
- scikit-image regionprops