Quantitative Big Imaging - Basic segmentation

Anders Kaestner

Mar 12, 2021

CONTENTS

Quantitative Big Imaging ETHZ: 227-0966-00L

Part 1: Image formation and thresholding

CHAPTER

ONE

TODAY'S LECTURE

- Motivation
- Qualitative Approaches
- Image formation and interpretation problems
- Thresholding
- Other types of images
- Selecting a good threshold
- Implementation
- Morphology
- Partial volume effects

CHAPTER

TWO

APPLICATIONS

In this lecture we are going to focus on basic segmentation approaches that work well for simple two-phase materials. Segmenting complex samples like

- Beyond 1 channel of depth
- Multiple phase materials
- Filling holes in materials
- Segmenting Fossils
- Attempting to segment the cortex in brain imaging (see figure below)

can be a very challenging task. Such tasks will be covered in later lectures.

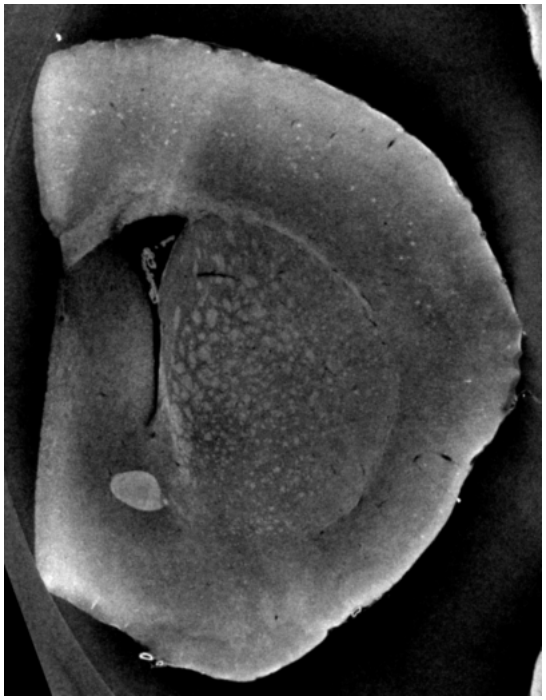


Fig. 2.1: An x-ray CT slice of the cortex. ``

- Simple two-phase materials (bone, cells, etc)
- Beyond 1 channel of depth
  - Multiple phase materials

- Filling holes in materials
- Segmenting Fossils
- Attempting to segment the cortex in brain imaging

CHAPTER

THREE

LITERATURE / USEFUL REFERENCES

- John C. Russ, “The Image Processing Handbook”,(Boca Raton, CRC Press)
- Available online within domain ethz.ch (or proxy.ethz.ch / public VPN)

3.1 Models / ROC Curves

- Julia Evans - Recalling with Precision
- Stripe’s Next Top Model

CHAPTER

FOUR

MOTIVATION: WHY DO WE DO IMAGING EXPERIMENTS?

There are different reasons for performing an image experiment. This often depends on in which state you are in your project.

4.1 Exploratory

In the initial phase, you want to learn what your sample looks like with the chosen modality. Maybe, you don't even know what is in there to see. The explorative type of experiment mostly only allows qualitative conclusions. These conclusions will however help you to formulate better hypotheses for more detailed experiments.

- To visually, qualitatively examine samples and differences between them
- No prior knowledge or expectations

4.2 To test a hypothesis

When you perform an experiment to test a hypothesis, you already know relatively much about your sample and want make an investigation where you can quantify characteristic features.

Quantitative assessment coupled with statistical analysis

- Does temperature affect bubble size?
- Is this gene important for cell shape and thus mechanosensation in bone?
- Does higher canal volume make bones weaker?
- Does the granule shape affect battery life expectancy?

4.3 What we are looking at?

4.4 To test a hypothesis

We perform an experiment bone to see how big the cells are inside the tissue:

↓

4.4.1 2560 x 2560 x 2160 x 32 bit = 56GB / sample

↓

4.4.2 20h of computer time later . . .

4.4.3 Way too much data, we need to reduce

4.5 What did we want in the first place?

4.5.1 *Single numbers*:

- volume fraction,
- cell count,
- average cell stretch,
- cell volume variability

4.6 Why do we perform segmentation?

In model-based analysis every step we peform, simple or complicated is related to an underlying model of the system we are dealing with

- Identify relevant regions in the images
- Many methods are available to solve the segmentation task.
- Choose wisely. . . *Occam's Razor* is very important here : **The simplest solution is usually the right one**

Advanced methods like a Bayesian, neural networks optimized using genetic algorithms with Fuzzy logic has a much larger parameter space to explore, establish sensitivity in, and must perform much better and be tested much more thoroughly than thresholding to be justified.

The next two lectures will cover powerful segmentation techinques, in particular with unknown data.

4.7 Review: Filtering and Image Enhancement

This was a noise process which was added to otherwise clean imaging data

$$I_{measured}(x,y) = I_{sample}(x,y) + \text{Noise}(x,y)$$

- What would the perfect filter be

$$Filter * I_{sample}(x,y) = I_{sample}(x,y)$$

What **most filters** end up doing $*Filter* * I_{measured}(x,y) = 90\% I_{real}(x,y) + 10\%\text{Noise}(x,y)$$

What **bad filters** do $*Filter* * I_{measured}(x,y) = 10\% I_{real}(x,y) + 90\%\text{Noise}(x,y)$$

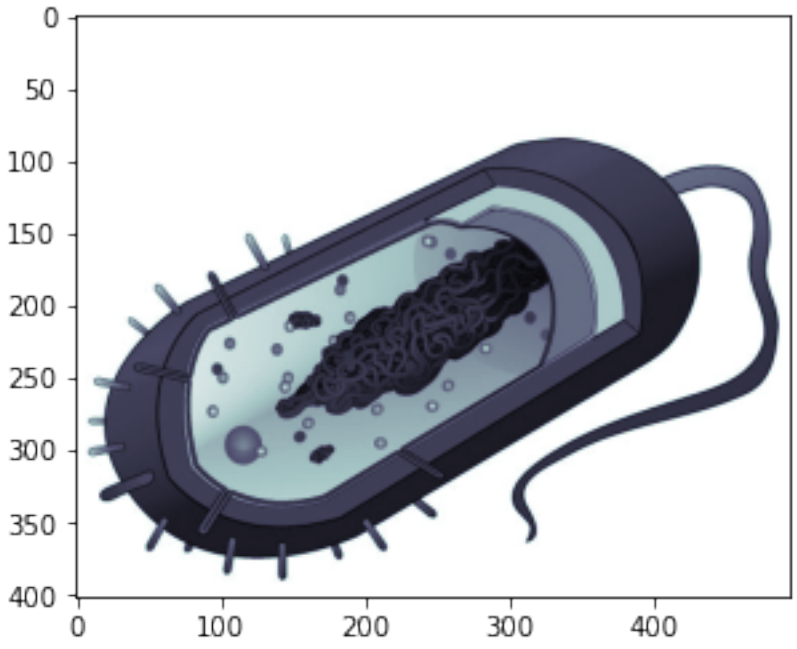
4.8 What we get from the imaging modality

To demonstrate what we get from a modality, we load the cell image as a toy example.

```
%matplotlib inline
from skimage.io import imread
from skimage.color import rgb2gray
import matplotlib.pyplot as plt
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-1-51744a8c7ee8> in <module>
      1 get_ipython().run_line_magic('matplotlib', 'inline')
----> 2 from skimage.io import imread
      3 from skimage.color import rgb2gray
      4 import matplotlib.pyplot as plt

ModuleNotFoundError: No module named 'skimage'
```

```
dkimg = imread("figures/Average_prokaryote_cell.jpg")
plt.imshow(rgb2gray(dkimg), cmap = 'bone');
```



CHAPTER

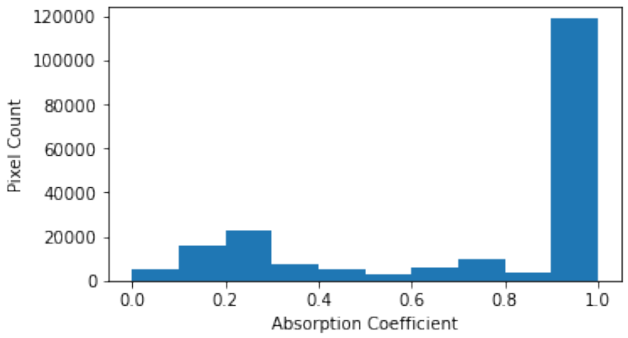
FIVE

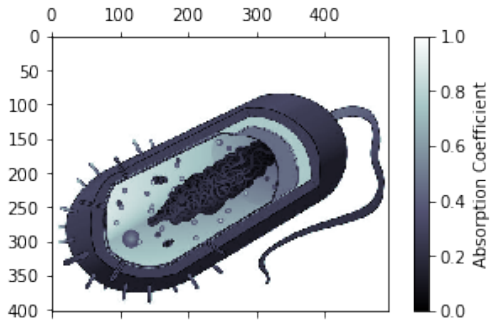
QUALITATIVE METRICS: WHAT DID PEOPLE USE TO DO?

- What comes out of our detector / enhancement process

```
%matplotlib inline
from skimage.io import imread
from skimage.color import rgb2gray
import matplotlib.pyplot as plt
```

```
dkimg = rgb2gray(imread("figures/Average_prokaryote_cell.jpg"))
fig, (ax_hist, ax_img) = plt.subplots(1, 2, figsize = (12,3))

ax_hist.hist(dkimg.ravel())
ax_hist.set_xlabel('Absorption Coefficient')
ax_hist.set_ylabel('Pixel Count')

m_show_obj = ax_img.matshow(dkimg, cmap = 'bone')
cb_obj = plt.colorbar(m_show_obj)
cb_obj.set_label('Absorption Coefficient')
```





5.1 Identify objects by eye

The first qualitative analysis is mostly done by eye. You look at the image to describe what you see. This first assessment will help you decide how to approach the quantitative analysis task. Here, it is important to think about using words that can be translated into an image processing workflow.

- Count,
- Describe qualitatively: “many little cilia on surface”, “long curly flaggelum”, “elongated nuclear structure”

5.2 Morphometrics

- Trace the outline of the object (or sub-structures)
- Employing the "cut-and-weigh" method

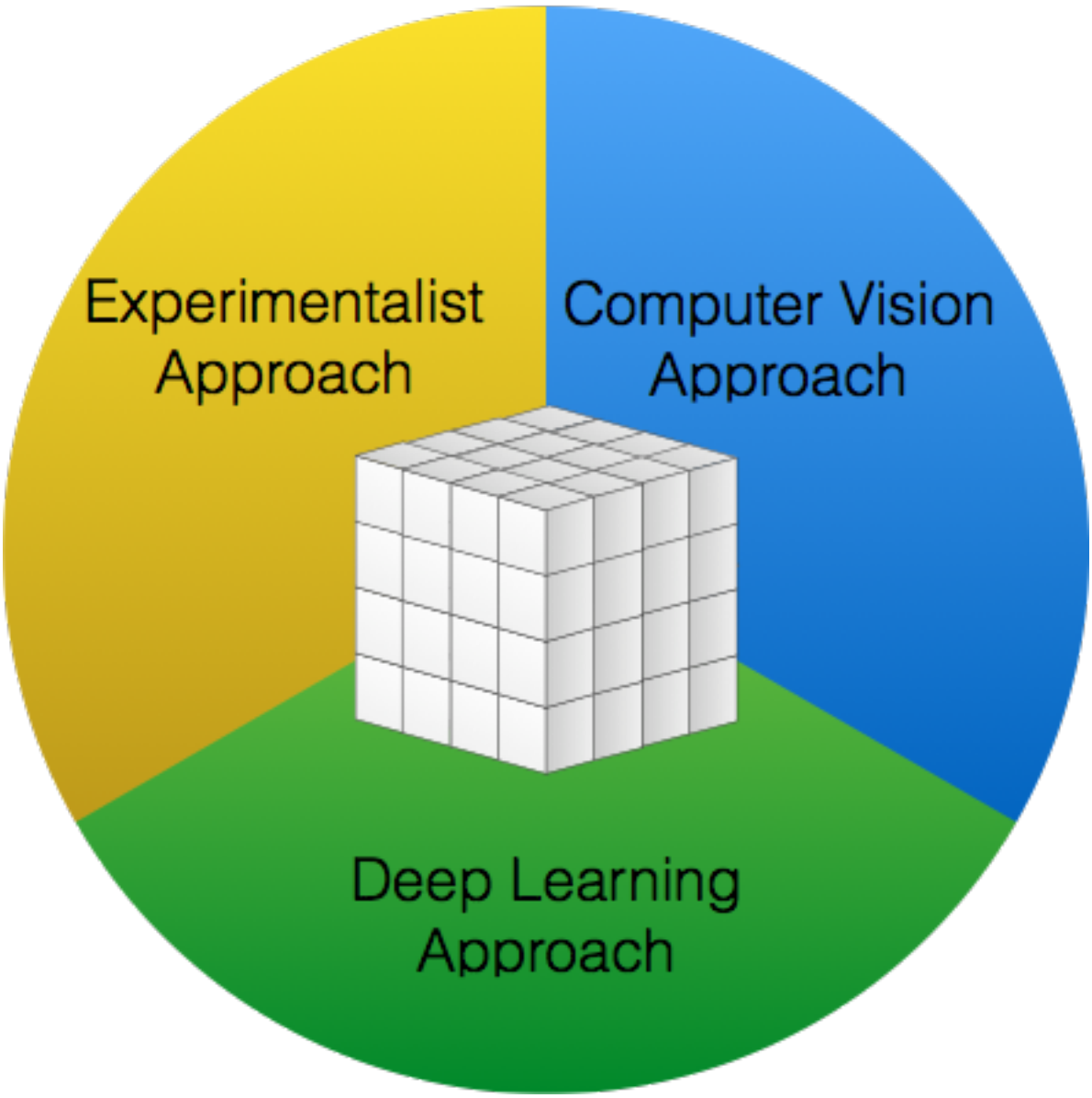
CHAPTER

SIX

SEGMENTATION APPROACHES

In the introduction lecture we talked about how people approach an image analysis problem depending on their background. This is something that becomes very clear when an image is about to be segmented.

They match up well to the world view / perspective



6.1 How to approach the segmenation task

6.1.1 Model based segmentation

The experimentalists approached the segmenation task based on their experience and knowledge about the samples. This results in a top-down approach and quite commonly based on models fitting the real world, what we actually can see in the images. The analysis aims at solving the problems needed to provide answers to the defined hypothesis.

6.1.2 Algorithmic segmentation approach

The opposite approach is to find and use generalized algorithms that provides the results. This approach is driven by the results as the computer vision and deep learning experts often don't have the knowledge to interpret the data.

Problem-driven

- Top-down
- *Reality* Model-based

6.2 Model-based Analysis

The image formation process is the process to use some kind of excitation or impulse probe a sample. This requires the interaction of the four parts in the figure below.

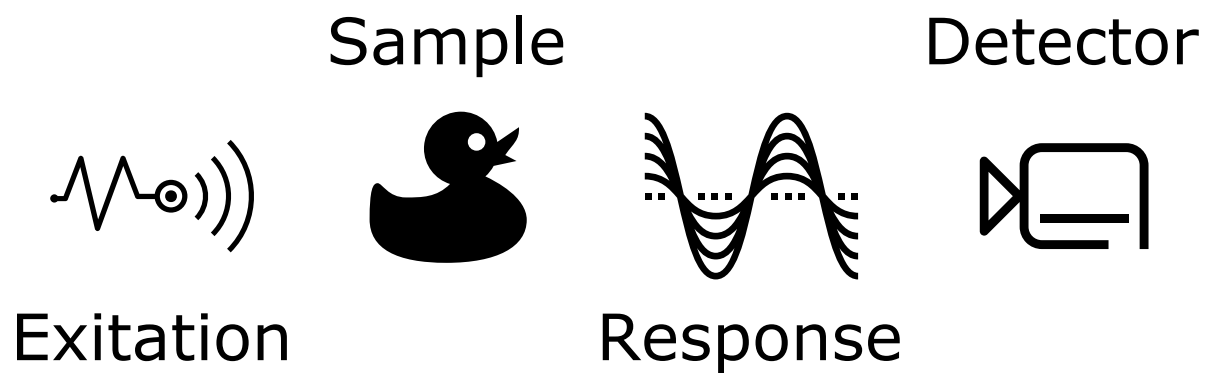


Fig. 6.1: The elements of the image formation process.

- **Impulses** Light, X-Rays, Electrons, A sharp point, Magnetic field, Sound wave
- **Characteristics** Electron Shell Levels, Electron Density, Phonons energy levels, Electronic, Spins, Molecular mobility
- **Response** Absorption, Reflection, Phase Shift, Scattering, Emission
- **Detection** Your eye, Light sensitive film, CCD / CMOS, Scintillator, Transducer
- Many different imaging modalities (μCT to MRI to Confocal to Light-field to AFM).
- Similarities in underlying equations, but different *coefficients*, *units*, and *mechanism*

$$I_{measured}(\vec{x}) = F_{system}(I_{stimulus}(\vec{x}), S_{sample}(\vec{x}))$$

6.2.1 Direct Imaging (simple)

In many setups there is un-even illumination caused by incorrectly adjusted equipment and fluctations in power and setups

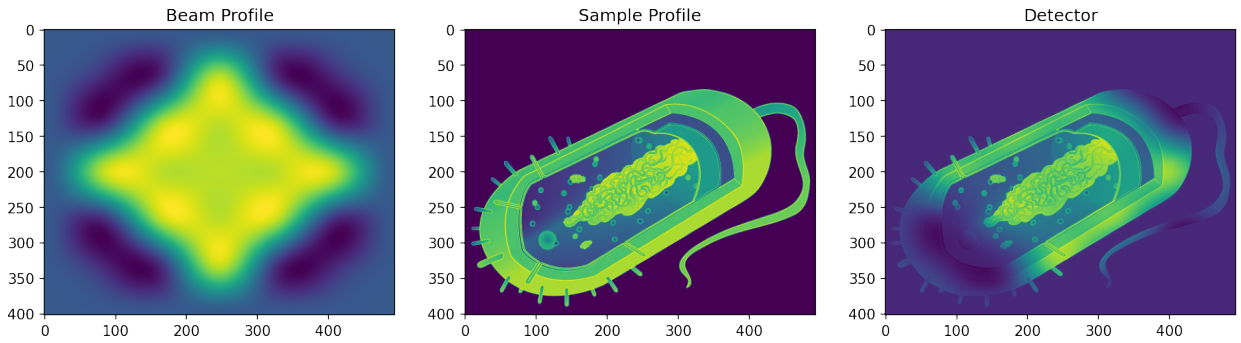
$F_{system}(a,b) = a * b$

$I_{stimulus} = \text{Beam}_{profile}\ S_{system} = \alpha(\vec{x}) \longrightarrow \alpha(\vec{x}) = \frac{I_{measured}(\vec{x})}{\text{Beam}_{profile}(\vec{x})}$

Let's create a simulated image acquisition with the cell image where you have beam profile that is penetrating the sample:

```
%matplotlib inline
from skimage.io import imread
from skimage.color import rgb2gray
import matplotlib.pyplot as plt
from skimage.morphology import disk
from scipy.ndimage import zoom
import numpy as np
```

```
cell_img = 1-rgb2gray(imread("figures/Average_prokaryote_cell.jpg"))
s_beam_img = np.pad(disk(2)/1.0, [[1,1], [1,1]], mode = 'constant', constant_values =␣
↪0.2)
beam_img = zoom(s_beam_img, [cell_img.shape[0]/7.0, cell_img.shape[1]/7.0])

fig, (ax_beam, ax_img, ax_det) = plt.subplots(1, 3, figsize = (15,6),dpi=150)

ax_beam.imshow(beam_img, cmap = 'viridis'); ax_beam.set_title('Beam Profile')

ax_img.imshow(cell_img, cmap = 'viridis'); ax_img.set_title('Sample Profile')

ax_det.imshow(cell_img*beam_img, cmap = 'viridis'); ax_det.set_title('Detector');
```



6.2.2 Profiles across the image

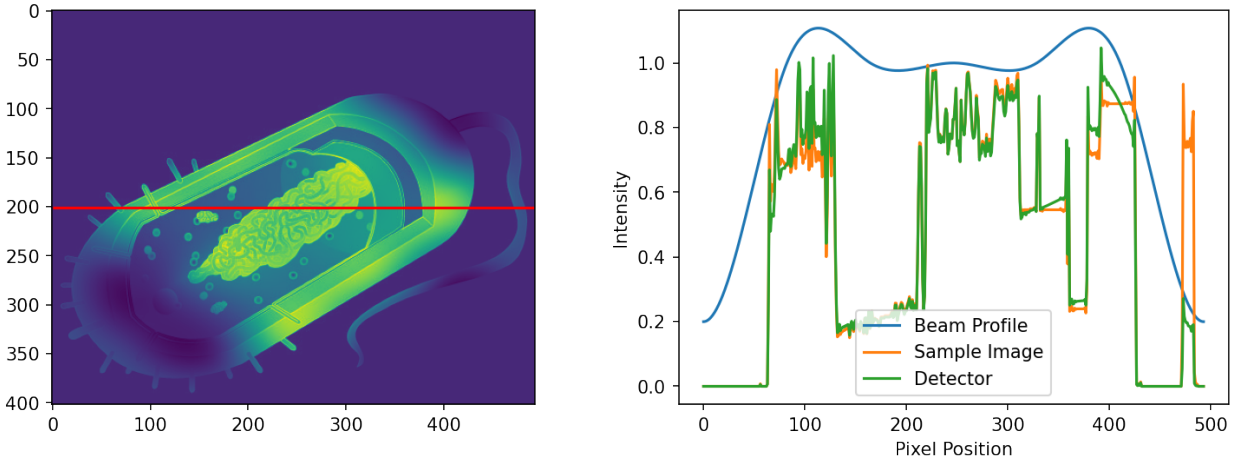
A first qualitative analysis on images of this type is to extract line profiles to see how the transmitted intensity changes across the sample. What we can see in this particular example is that the acquired profile tapers off with the beam intensity. With this in mind, it may come clear to you that you need to normalize the images by the beam profile.

```
fig, ax = plt.subplots(1, 2, figsize = (12,4),dpi=150)
ax[0].imshow(cell_img*beam_img); ax[0].hlines(beam_img.shape[0]//2,xmin=0,xmax=beam_
↪img.shape[1]-1,color='red')
ax[1].plot(beam_img[beam_img.shape[0]//2], label = 'Beam Profile')
```

(continues on next page)

(continued from previous page)

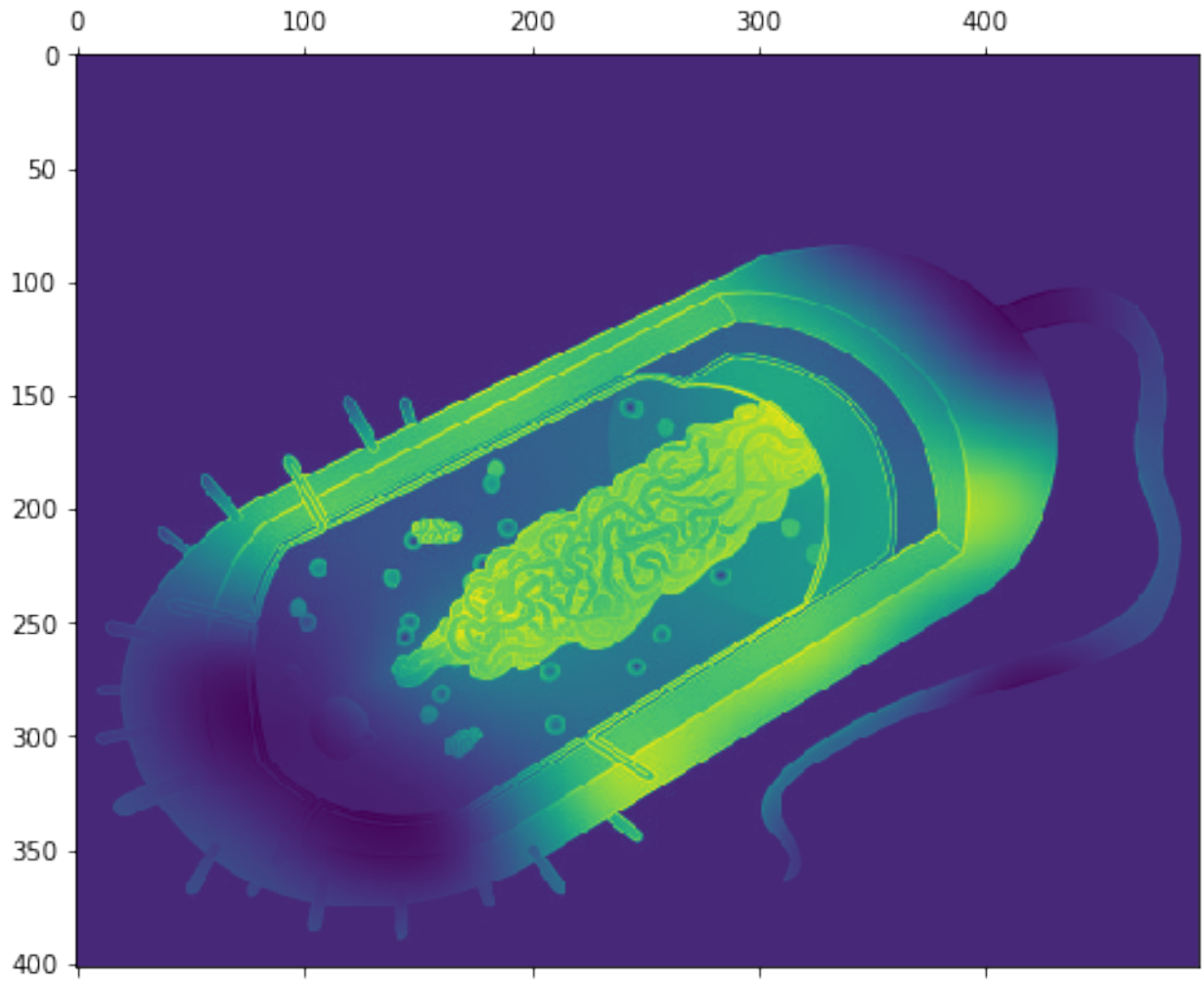
```
ax[1].plot(cell_img[beam_img.shape[0]//2], label = 'Sample Image')
ax[1].plot((cell_img*beam_img)[beam_img.shape[0]//2], label = 'Detector')
ax[1].set_ylabel('Intensity'); ax[1].set_xlabel('Pixel Position');ax[1].legend(loc=
↪"lower center");
```



6.2.3 Inhomogeneous illumination

- Frequently there is a fall-off of the beam away from the center (as is the case of a Gaussian beam which frequently shows up for laser systems).
- This can make extracting detail away from the center much harder.

```
fig, ax1 = plt.subplots(1,1, figsize = (8,8))
ax1.matshow(cell_img*beam_img,cmap = 'viridis');
```



6.2.4 Absorption Imaging (X-ray, Ultrasound, Optical)

For absorption/attenuation imaging → Beer-Lambert Law

$$I_{detector} = \underbrace{I_{source}}_{I_{stimulus}} \underbrace{e^{-\alpha d}}_{S_{sample}}$$

Different components have a different α based on the strength of the interaction between the light and the chemical / nuclear structure of the material

$$I_{sample}(x,y) = I_{source} \cdot e^{-\alpha(x,y)\cdot d}$$

For segmentation this model is:

- there are 2 (or more) distinct components that make up the image
- these components are distinguishable by their values (or vectors, colors, tensors, . . .)

```
%matplotlib inline
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
import numpy as np
import pandas as pd
```

6.2.5 A numerical transmission imaging example (1D)

In this example we create a sample with three different materials and the sample thickness 1.0. The attenuation coefficient is modelled by random models to give them some realistic spread.

The transmission uses Beer Lambert's law.

```
I_source = 1.0
d = 1.0
alpha_1 = np.random.normal(1, 0.25, size = 100) # Material 1
alpha_2 = np.random.normal(2, 0.25, size = 100) # Material 2
alpha_3 = np.random.normal(3, 0.5, size = 100)  # Material 3

abs_df = pd.DataFrame([dict(alpha = c_x, material = c_mat) for c_vec, c_mat in␣
↪zip([alpha_1, alpha_2, alpha_3],
                      ['material 1', 'material 2', 'material 3']) for c_x in c_vec])

abs_df['I_detector'] = I_source*np.exp(-abs_df['alpha']*d)
abs_df.sample(5)
```

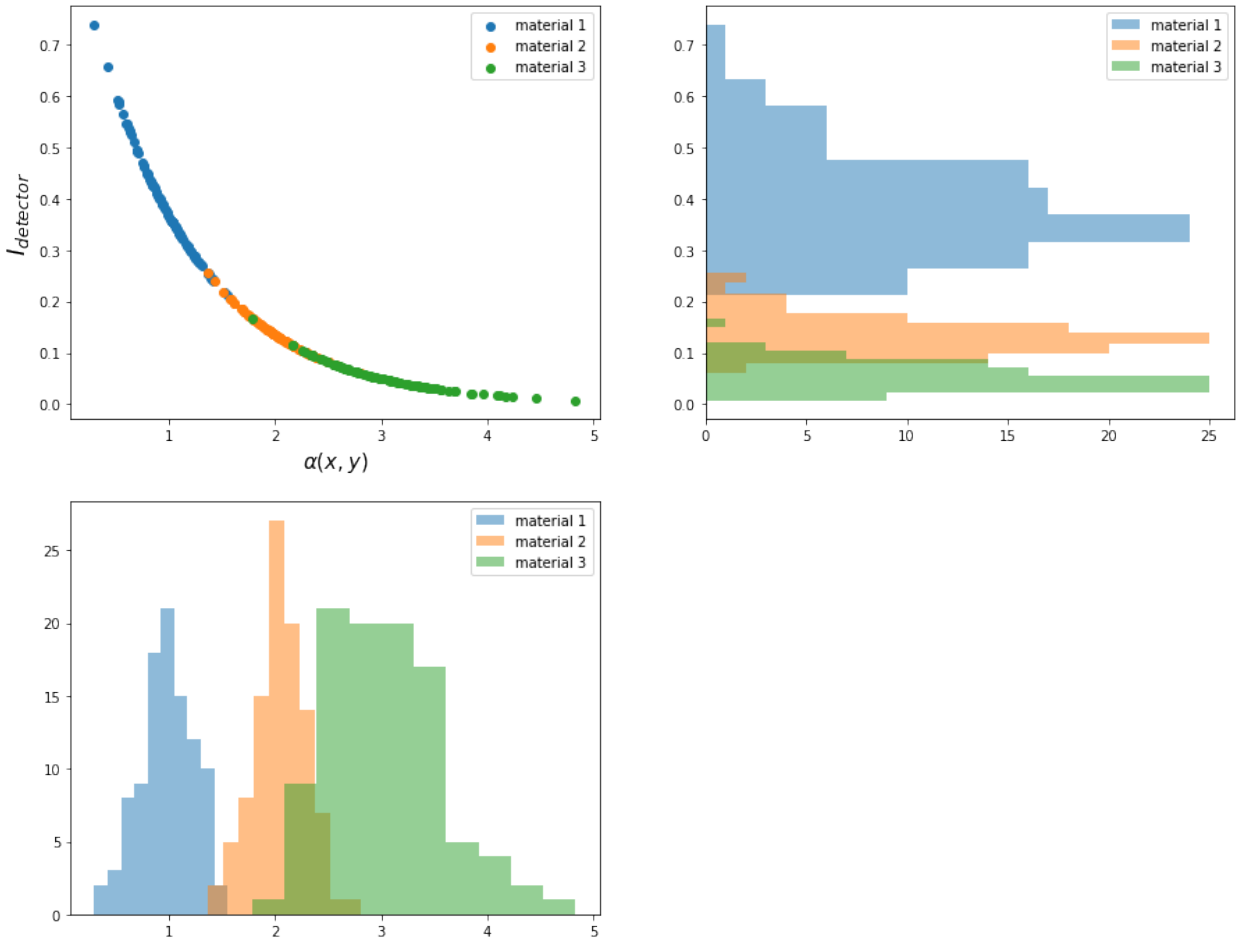
	alpha	material	I_detector
270	3.853025	material 3	0.021215
73	0.886718	material 1	0.412006
178	1.519744	material 2	0.218768
180	1.745622	material 2	0.174536
256	3.399326	material 3	0.033396

In the table, you can see that we measure different intensities on the detector depending on the material the beam is penetrating.

Plotting measured intensities

Let's now plot the intensities and attenuation coefficients and compare the outcome of our transmission experiment.

```
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2,2, figsize = (15, 12))
for c_mat, c_df in abs_df.groupby('material'):
    ax1.scatter(x = c_df['alpha'],
                y = c_df['I_detector'],
                label = c_mat)
    ax3.hist(c_df['alpha'], alpha = 0.5, label = c_mat)
    ax2.hist(c_df['I_detector'],  alpha = 0.5, label = c_mat, orientation="horizontal
↪")
ax1.set_xlabel('$\\alpha(x,y)$', fontsize = 15); ax1.set_ylabel('$I_{detector}$',␣
↪fontsize = 18)

ax1.legend(); ax2.legend(); ax3.legend(loc = 0); ax4.axis('off');
```



The α-$I_{detector}$ plot shows the curved exponential behaviour we can expect from Beer Lambert's law. Now, if we look at the histogram, we can see that distribution of attenuation coefficients doesn't really match the measured intensity. In this example, it is even so that the widths of the diffent materials have changed places. Great attenuation coefficient results in little transmission and small attenuation coefficient allow more of the beam to penetrate the sample.

CHAPTER

SEVEN

EXAMPLE MAMMOGRAPHY

Mammographic imaging is an area where model-based absorption imaging is problematic.

Even if we assume a constant illumination (*rarely* the case),

$$I_{detector} = \underbrace{I_{source}}_{I_{stimulus}} \underbrace{\exp(-\alpha d)}_{S_{sample}}$$

$$\downarrow$$

$$I_{detector} = \exp(-\alpha(x,y)d(x,y))$$

$$\downarrow$$

$$I_{detector} = \exp\left(-\int_0^l \alpha(x,y,z)dz\right)$$

The assumption that the attenuation coefficient, α, is constant is rarely valid. Then you see that the exponent turns into an integral along the probing ray and that α is a function of the position in the sample. This of course leads ambiguity in the interpretation of what the pixel intensity really means.

7.1 Problems to interpret radiography images

Specifically the problem is related to the inability to separate the

- α - attenuation
- d - thickness terms.

To demonstrate this, we model a basic breast volume as a half sphere with a constant absorption factor:

	Air	Breast tissue
$\alpha(x,y,z)$	0	0.01

$\rightarrow$ The $\int$ then turns into a Σ in discrete space

7.2 Building a breast phantom

The breast is here modelled as a half sphere of constant attenuation coefficient:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from skimage.morphology import ball

# For the 3D rendering
import plotly.offline as py
from plotly.figure_factory import create_trisurf
from skimage.measure import marching_cubes
```

```
breast_mask = ball(50)[:,50:]  # This is our model

# just for 3D rendering, don't worry about it
py.init_notebook_mode()
vertices, simplices, _, _ = marching_cubes(breast_mask>0)
x,y,z = zip(*vertices)
fig = create_trisurf(   x=x, y=y, z=z,
                        plot_edges=False,
                        simplices=simplices,
                        title="Breast Phantom")
py.iplot(fig)
```

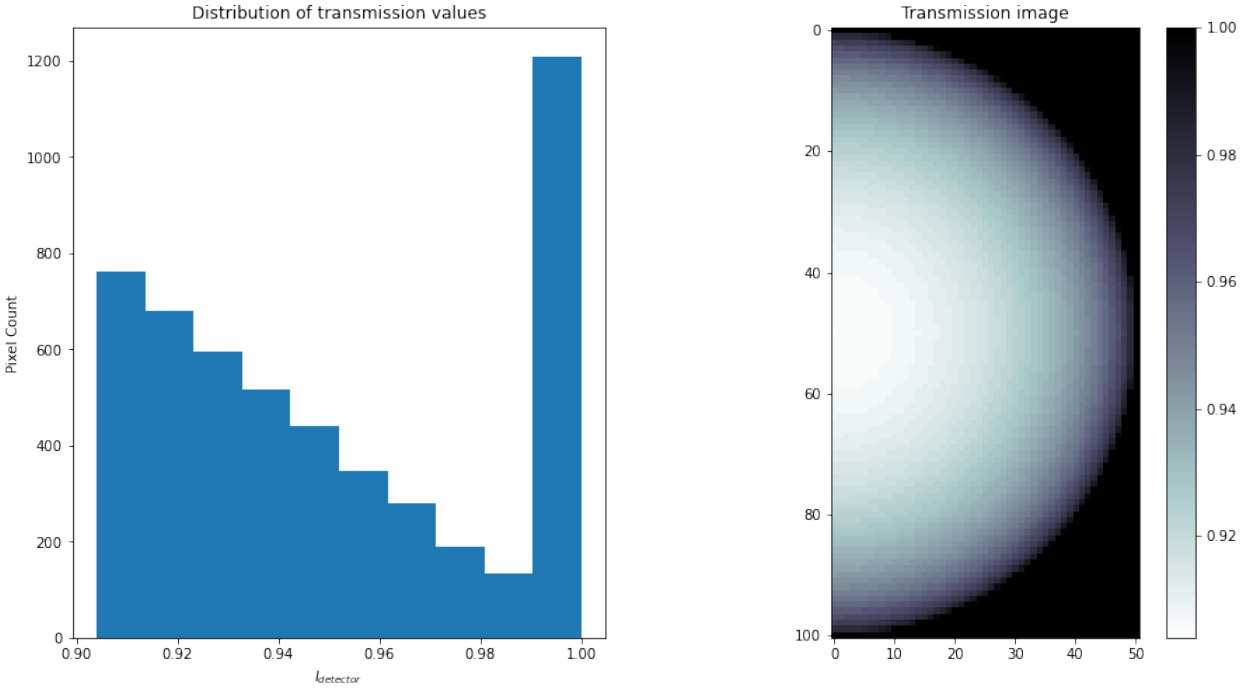
7.2.1 Transmission image of the breast phantom

Our first step is to simulate a transmission image of the breast. This is done by

1. Summing the attenuation coefficents times the pixel size.
2. Applying Beer-Lambert's law

This produces a 2D image of the side view of the breast.

```
breast_alpha = 1e-2                             # The attenuation coefficient
pixel_size   = 0.1                              # The simulated detector has 1mm pixels
breast_vol   = breast_alpha*breast_mask         # Scale the image intensity by␣
↪attenuation coefficient
i_detector   = np.exp(-np.sum(breast_vol,2)*pixel_size)  # Compute the transmission␣
↪through the phantom

fig, (ax_hist, ax_breast) = plt.subplots(1, 2, figsize = (15,8))

b_img_obj = ax_breast.imshow(i_detector, cmap = 'bone_r'); plt.colorbar(b_img_obj) ;
↪ax_breast.set_title('Transmission image')

ax_hist.hist(i_detector.flatten()); ax_hist.set_xlabel('$I_{detector}$'); ax_hist.set_
↪ylabel('Pixel Count');ax_hist.set_title('Distribution of transmission values');
```



The histogram shows the distribution of the transmitted intensity.

7.2.2 Compute the thickness

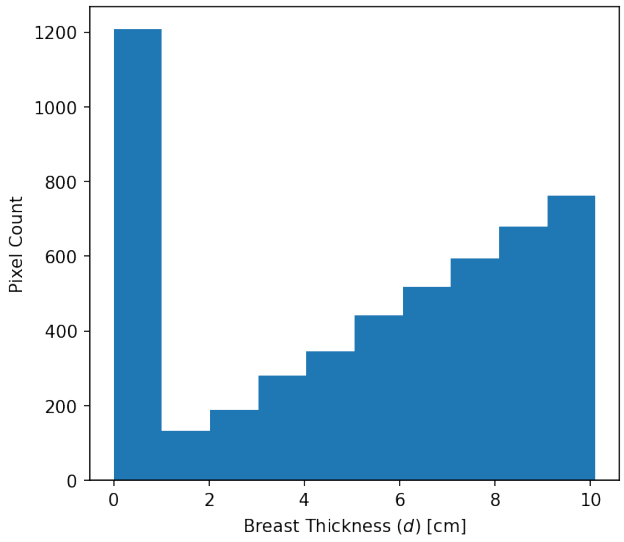
If we know that α is constant we can reconstruct the thickness d from the image:

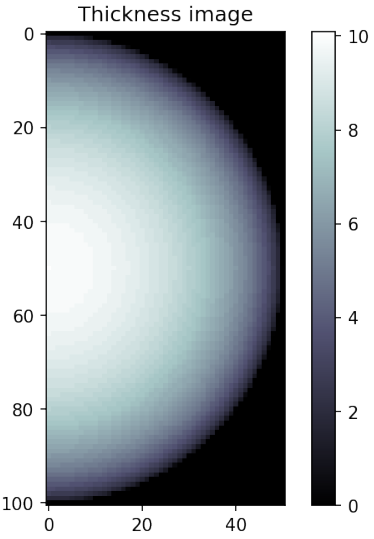
$$d = -\log(I_{detector})$$

This is only valid because we have air ($\alpha = 0$) as the second component in the phantom. Otherwise, if it was a denser material we would have a material mixture.

Now, let's compute the breast thickness from the transmission image:

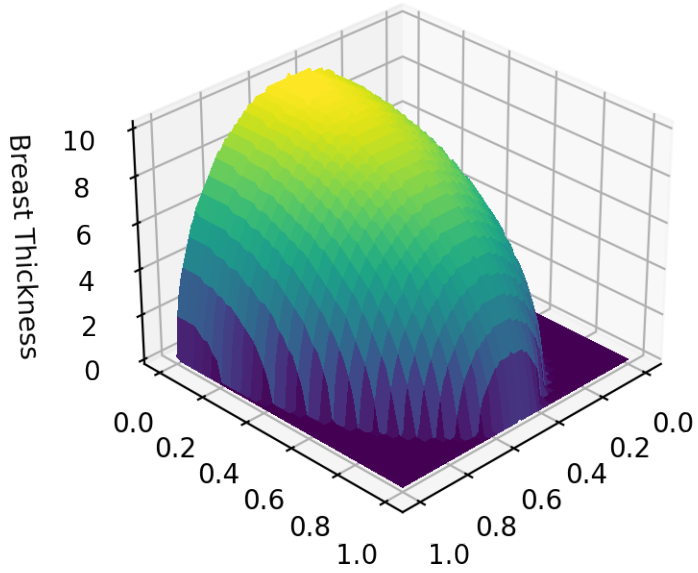
```
breast_thickness = -np.log(i_detector)/breast_alpha
fig, (ax_hist, ax_breast) = plt.subplots(1, 2, figsize = (12,5), dpi=150)

b_img_obj = ax_breast.imshow(breast_thickness, cmap = 'bone'); ax_breast.set_title(
↪'Thickness image')
plt.colorbar(b_img_obj)

ax_hist.hist(breast_thickness.flatten()) ; ax_hist.set_xlabel('Breast Thickness ($d$)␣
↪[cm]'); ax_hist.set_ylabel('Pixel Count');
```





7.2.3 Visualizing the thickness

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize = (8, 4), dpi = 200)
ax  = fig.gca(projection='3d')
# Plot the surface.
yy, xx = np.meshgrid(np.linspace(0, 1, breast_thickness.shape[1]),
                     np.linspace(0, 1, breast_thickness.shape[0]))
surf = ax.plot_surface(xx, yy, breast_thickness, cmap=plt.cm.viridis,
                       linewidth=0, antialiased=False)
ax.view_init(elev = 30, azim = 45)
ax.set_zlabel('Breast Thickness');
```



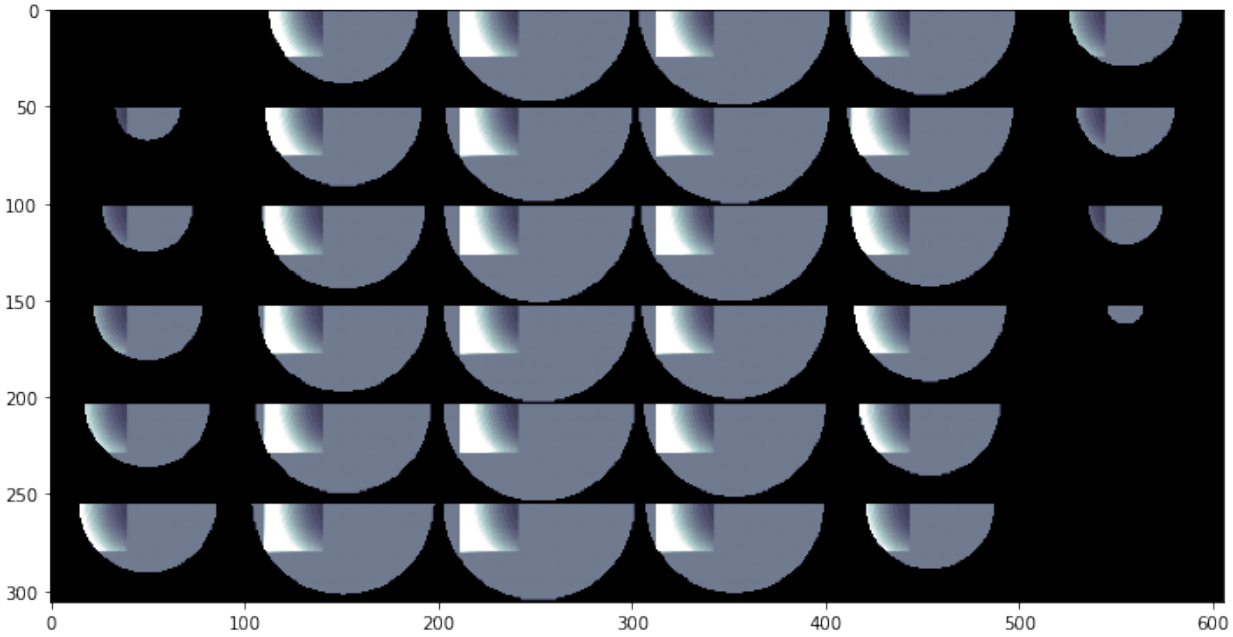
7.3 What if α is not constant?

We run into problems when the α is no longer constant.

- For example if we place a dark lump in the center of the breast.
- It is **impossible** to tell if the breast is *thicker* or if the lump inside is *denser*.

For the lump below we can see on the individual slices of the sample that the lesion appears quite clearly and is very strangely shaped.

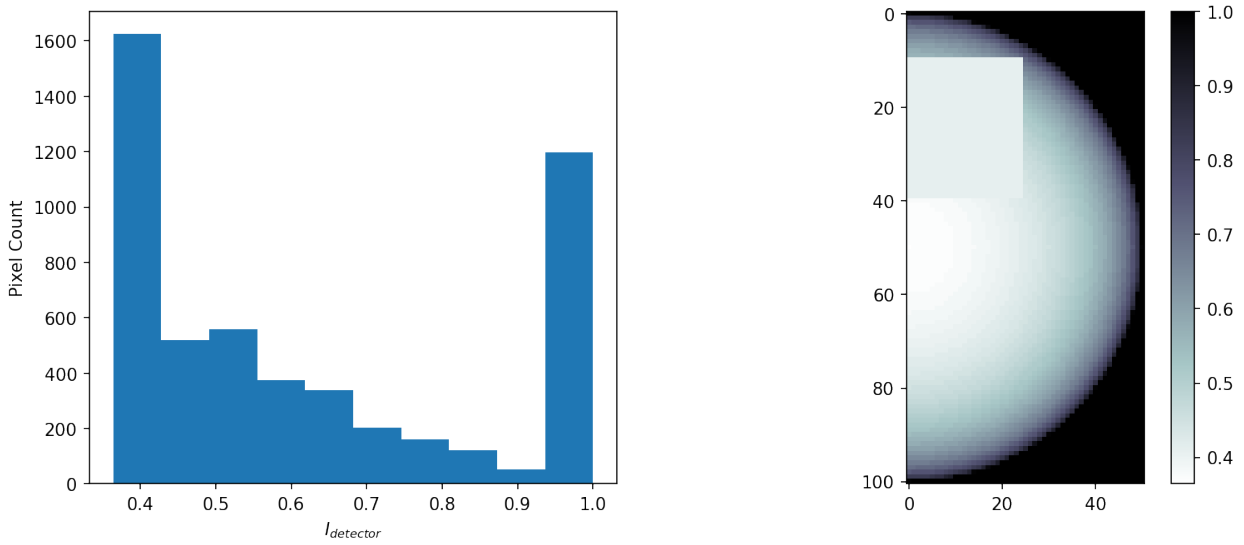
```
breast_vol = breast_alpha*breast_mask
renorm_slice = np.sum(breast_mask[10:40, 0:25], 2)/np.sum(breast_mask[30, 10])
breast_vol[10:40, 0:25] /= np.stack([renorm_slice]*breast_vol.shape[2],-1)

from skimage.util import montage as montage2d
fig, ax1 = plt.subplots(1,1, figsize = (12, 12))
ax1.imshow(montage2d(breast_vol.swapaxes(0,2).swapaxes(1,2)[::3]).transpose(),
           cmap = 'bone', vmin = breast_alpha*.8, vmax = breast_alpha*1.2);
```



7.3.1 Looking at the thickness again

When we make the projection and apply Beer's Law we see that it appears as a relatively constant region in the image

```
i_detector = np.exp(-np.sum(breast_vol,2))

fig, (ax_hist, ax_breast) = plt.subplots(1, 2, figsize = (12,5),dpi=150)

b_img_obj = ax_breast.imshow(i_detector, cmap = 'bone_r')
plt.colorbar(b_img_obj)

ax_hist.hist(i_detector.flatten())
ax_hist.set_xlabel('$I_{detector}$')
ax_hist.set_ylabel('Pixel Count');
```



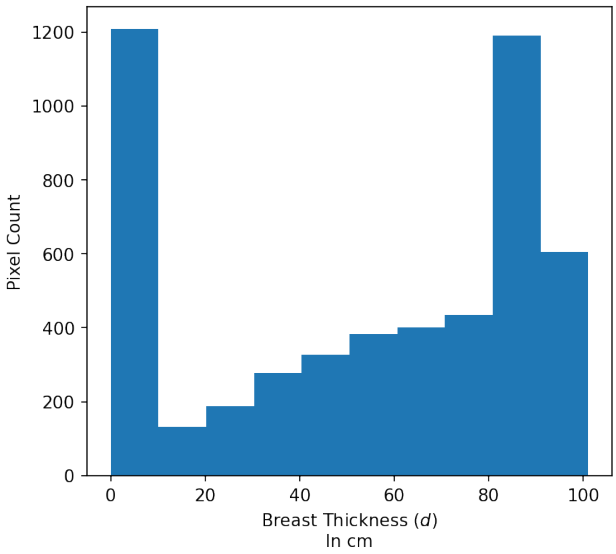
7.3.2 An anomaly in the thickness reconstruction

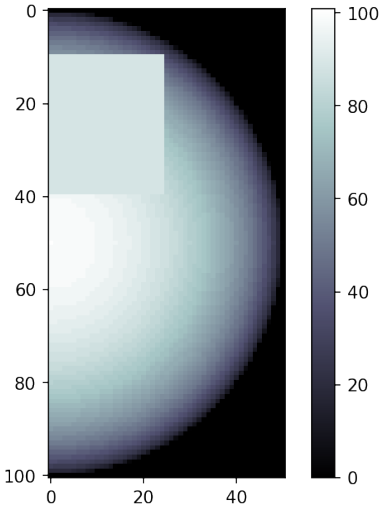
It appears as a flat constant region in the thickness reconstruction.

So we fundamentally from this single image cannot answer:

- is the breast oddly shaped?
- or does it have an possible tumor inside of it?

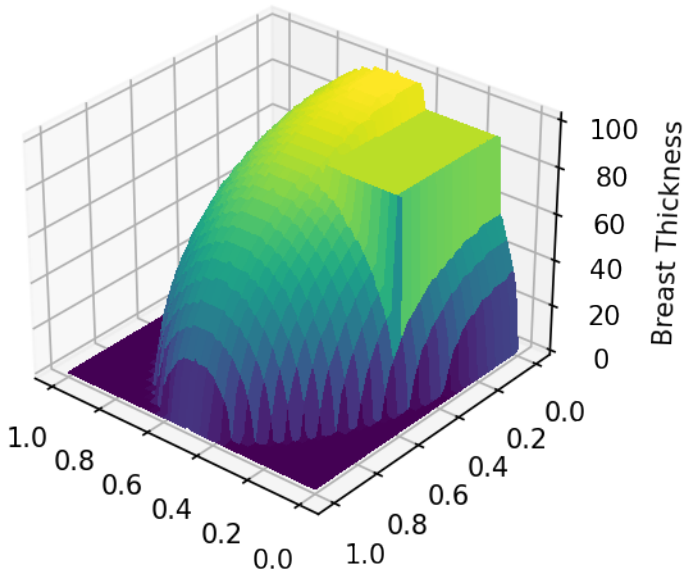
```
breast_thickness = -np.log(i_detector)/1e-2
fig, (ax_hist, ax_breast) = plt.subplots(1, 2, figsize = (12,5),dpi=150)

b_img_obj = ax_breast.imshow(breast_thickness, cmap = 'bone')
plt.colorbar(b_img_obj)

ax_hist.hist(breast_thickness.flatten())
ax_hist.set_xlabel('Breast Thickness ($d$)\nIn cm')
ax_hist.set_ylabel('Pixel Count');
```





7.3.3 Looking at the thickness profile with lump

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize = (8, 4), dpi = 150)
ax  = fig.gca(projection='3d')

# Plot the surface.
yy, xx = np.meshgrid(np.linspace(0, 1, breast_thickness.shape[1]),
                     np.linspace(0, 1, breast_thickness.shape[0]))
surf = ax.plot_surface(xx, yy, breast_thickness, cmap=plt.cm.viridis,
                       linewidth=0, antialiased=False)
ax.view_init(elev = 30, azim = 130)
ax.set_zlabel('Breast Thickness');
```



CHAPTER

EIGHT

SEGMENTATION

8.1 Where does segmentation get us?

We can convert a decimal value or something even more complicated like

- 3 values for RGB images,
- a spectrum for hyperspectral imaging,
- or a vector / tensor in a mechanical stress field

To a single or a few discrete values:

- usually true or false,
- but for images with phases it would be each phase, e.g. bone, air, cellular tissue.

2560 x 2560 x 2160 x 32 bit = 56GB / sample $\rightarrow$ 2560 x 2560 x 2160 x **1 bit** = 1.75GB / sample

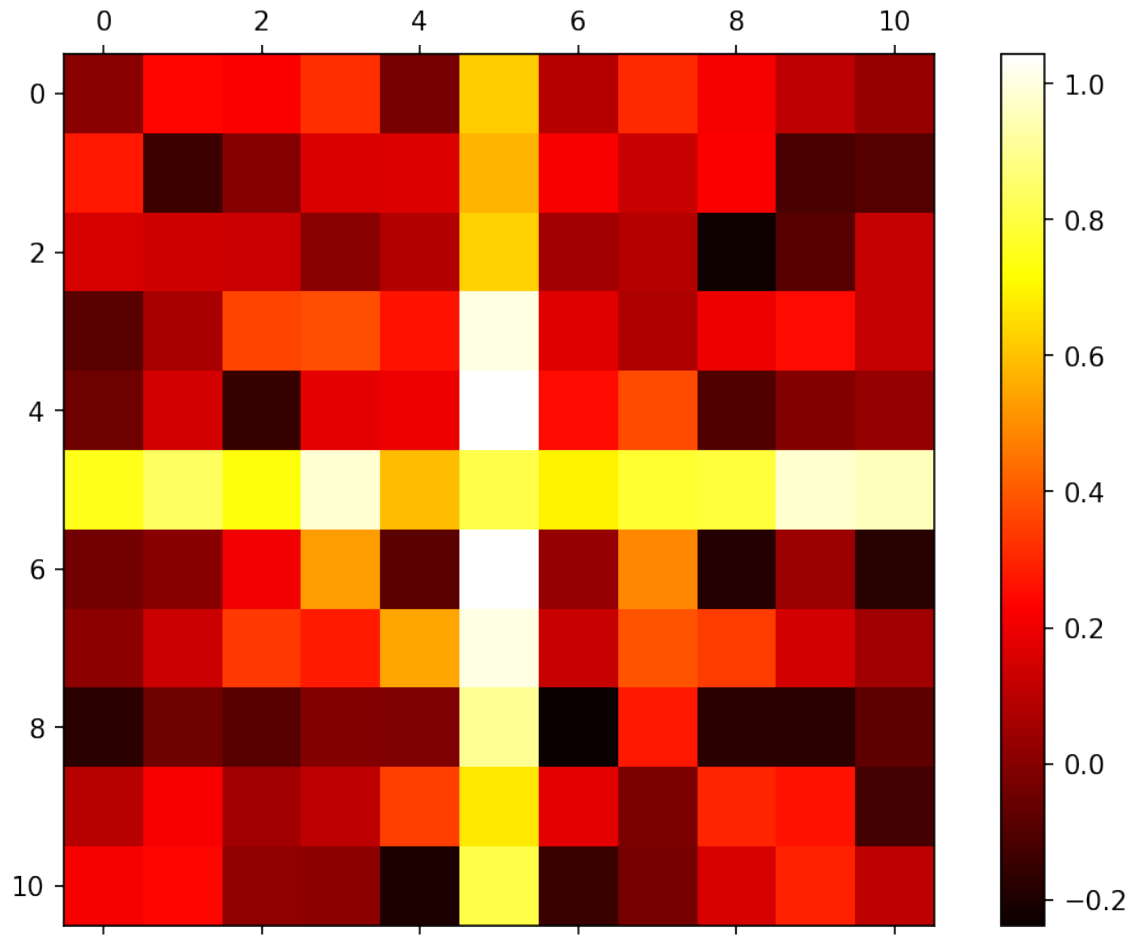
8.2 Basic segmentation: Applying a threshold to an image

Start out with a simple image of a cross with added noise $$I(x,y) = f(x,y)$$

Here, we create a test image with two features embedded in uniform noise; a cross with values in the order of '1' and background with values in the order '0'. The figure below shows the image and its histogram. The histogram helps us to see how the graylevels are distributed which guides the decision where to put a threshold that segments the cross from the background.

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

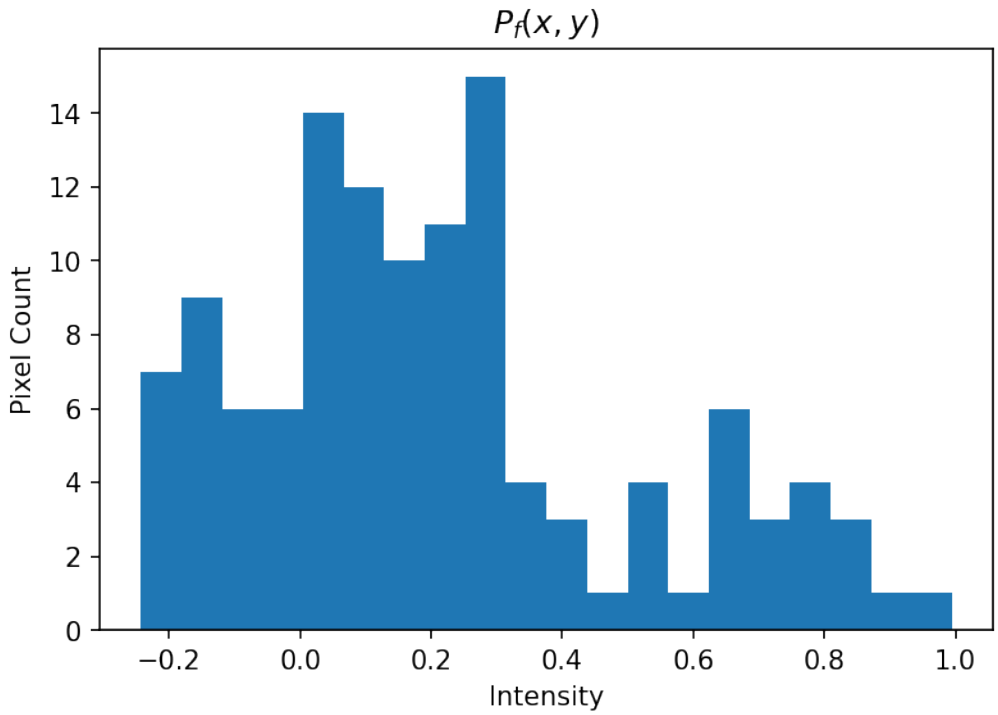
```
nx = 5; ny = 5
xx, yy = np.meshgrid(np.arange(-nx, nx+1)/nx*2*np.pi,
                     np.arange(-ny, ny+1)/ny*2*np.pi)
cross_im = 1.5*np.abs(np.cos(xx*yy))/(np.abs(xx*yy)+(3*np.pi/nx))+np.random.uniform(-
↪0.25, 0.25, size = xx.shape)

fig,ax = plt.subplots(1,1,figsize=(9,6), dpi=150)
im=ax.matshow(cross_im, cmap = 'hot')
fig.colorbar(im);
```



8.3 The histogram

The intensity can be described with a probability density function $$P_f(x,y)$$

```
fig, ax1 = plt.subplots(1,1,dpi=150)
ax1.hist(cross_im.ravel(), 20)
ax1.set_title('$P_f(x,y)$'); ax1.set_xlabel('Intensity'); ax1.set_ylabel('Pixel Count
↪');
```



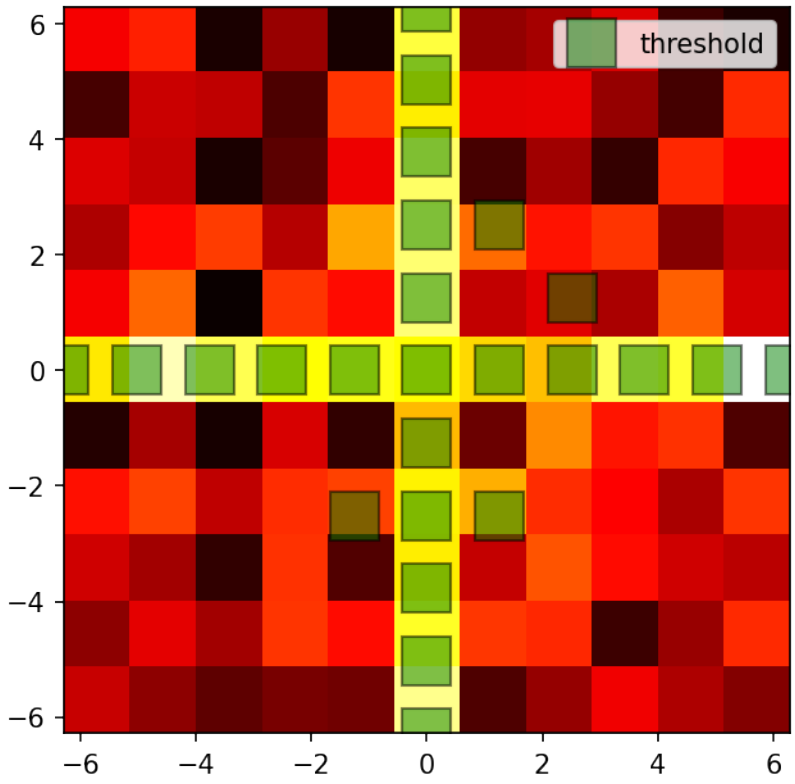
8.4 Applying a threshold to an image

By examining the image and probability distribution function, we can *deduce* that the underyling model is a whitish phase that makes up the cross and the darkish background

Applying the threshold is a deceptively simple operation

$$I(x,y) = \begin{cases} 1, & f(x,y) \geq 0.40 \\ 0, & f(x,y) < 0.40 \end{cases}$$

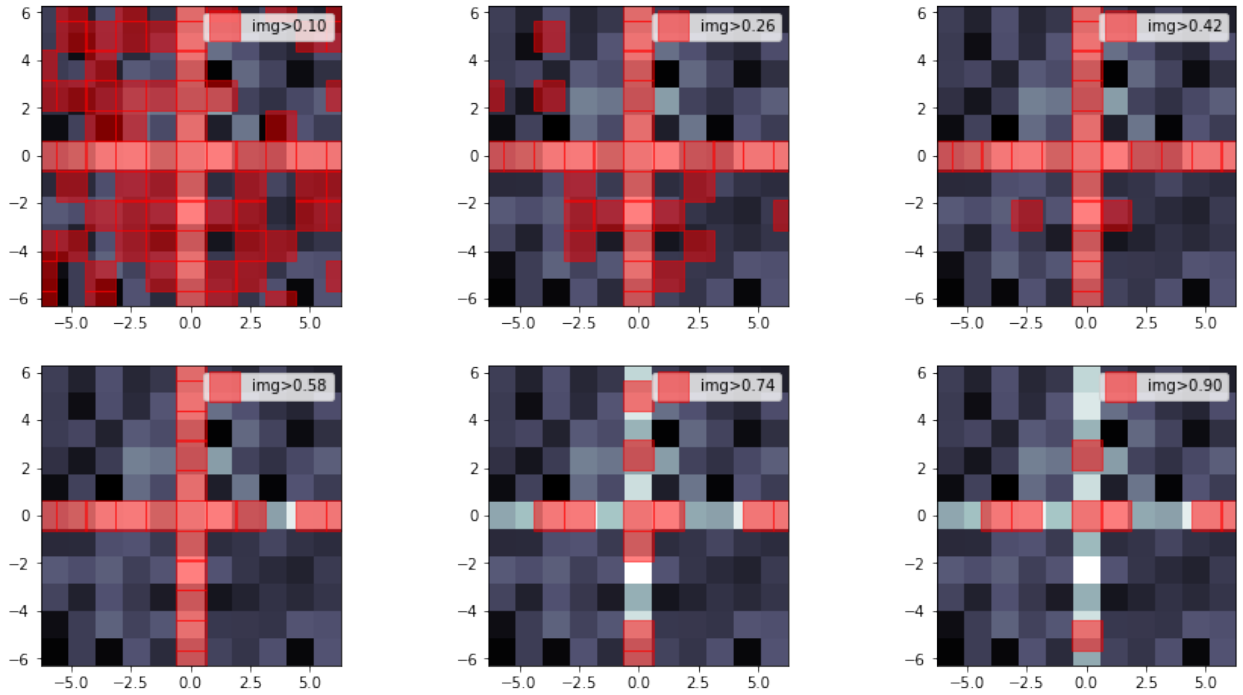
```
threshold = 0.4
fig, ax1 = plt.subplots(1,1,figsize=(8,5),dpi=150)
ax1.imshow(cross_im, cmap = 'hot', extent = [xx.min(), xx.max(), yy.min(), yy.max()])
thresh_img = cross_im > threshold

ax1.plot(xx[np.where(thresh_img)], yy[np.where(thresh_img)],
         'ks', markerfacecolor = 'green', alpha = 0.5, label = 'threshold',␣
↪markersize = 18)
ax1.legend();
```



8.4.1 Various Thresholds

We can see the effect of choosing various thresholds

```
fig, m_axs = plt.subplots(2,3,
                          figsize = (15, 8))
for c_thresh, ax1 in zip(np.linspace(0.1, 0.9, 6), m_axs.flatten()):

    ax1.imshow(cross_im,
               cmap = 'bone',
               extent = [xx.min(), xx.max(), yy.min(), yy.max()])
    thresh_img = cross_im > c_thresh

    ax1.plot(xx[np.where(thresh_img)], yy[np.where(thresh_img)], 'rs', alpha = 0.5,␣
↪label = 'img>%2.2f' % c_thresh, markersize = 20)
    ax1.legend(loc = 1);
```



In this fabricated example we saw that thresholding can be a very simple and quick solution to the segmentation problem. Unfortunately, real data is often less obvious. The features we want to identify for our qantitative analysis are often obscured be different other features in the image. They may be part of the setup of caused by the acquisition conditions.

CHAPTER

NINE

SEGMENTING CELLS

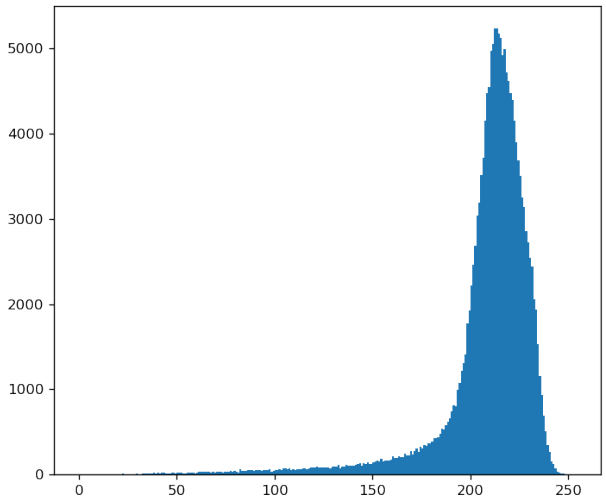
We can peform the same sort of analysis with this image of cells

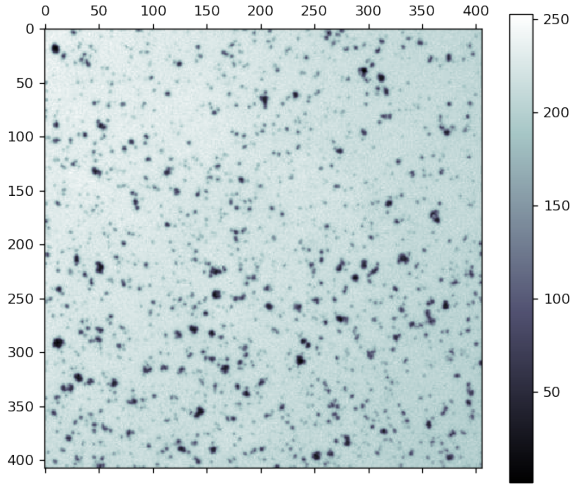
This time we can derive the model from the basic physics of the system

- The field is illuminated by white light of nearly uniform brightness
- Cells absorb light causing darker regions to appear in the image
- *Lighter* regions have no cells
- **Darker** regions have cells

```
%matplotlib inline
from skimage.io import imread
import matplotlib.pyplot as plt
import numpy as np
```

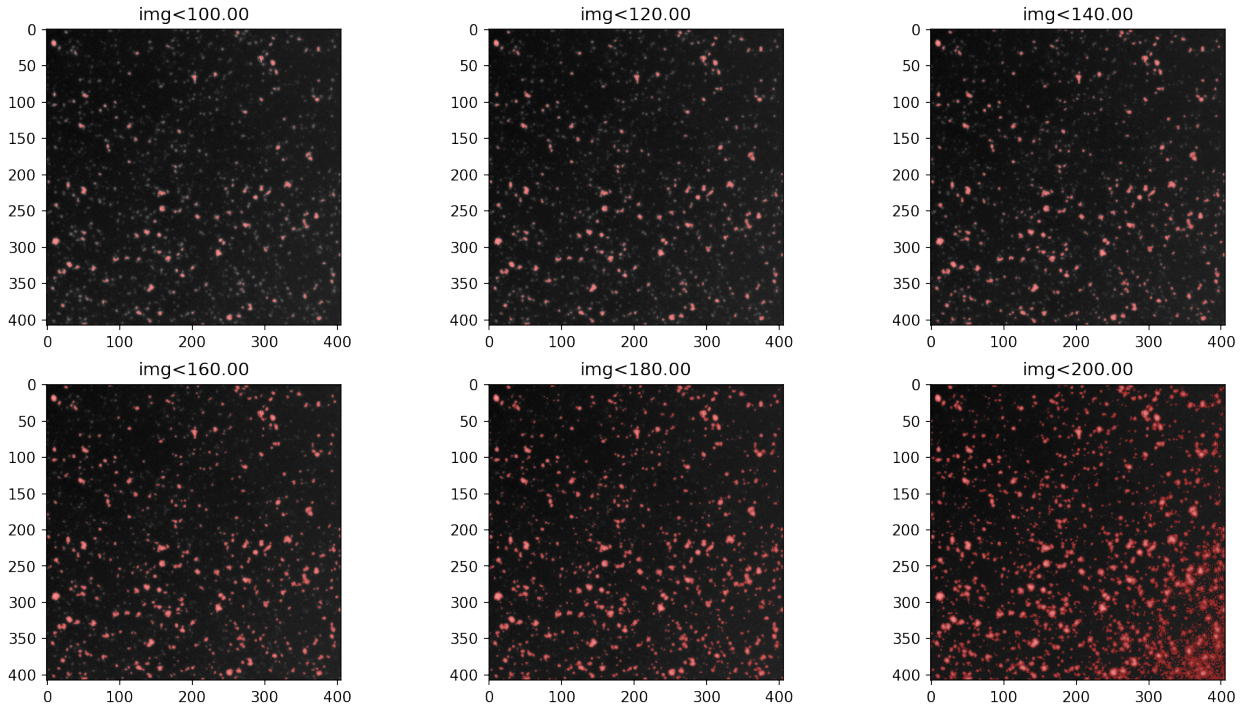
```
cell_img = imread("figures/Cell_Colony.jpg")

fig, (ax_hist, ax_img) = plt.subplots(1, 2, figsize = (15,6), dpi=120)
ax_hist.hist(cell_img.ravel(), np.arange(255))
ax_obj = ax_img.matshow(cell_img, cmap = 'bone')
plt.colorbar(ax_obj);
```





9.1 Trying different thresholds on the cell image

```
from skimage.color import label2rgb
fig, m_axs = plt.subplots(2,3,
                          figsize = (15, 8), dpi = 150)
for c_thresh, ax1 in zip(np.linspace(100, 200, 6), m_axs.flatten()):
    thresh_img = cell_img < c_thresh

    ax1.imshow(label2rgb(thresh_img, image = 1-cell_img, bg_label = 0, alpha = 0.4))
↪# Rgb coding of image and mask

    ax1.set_title('img<%2.2f' % c_thresh)
```



CHAPTER

TEN

OTHER IMAGE TYPES

While scalar images are easiest, it is possible for any type of image $\$I(x, y) = \vec{f}(x, y)\$$

```
%matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

Here, we create an image with vectors to show local orientation and intensities to meassure the streng of a signal.

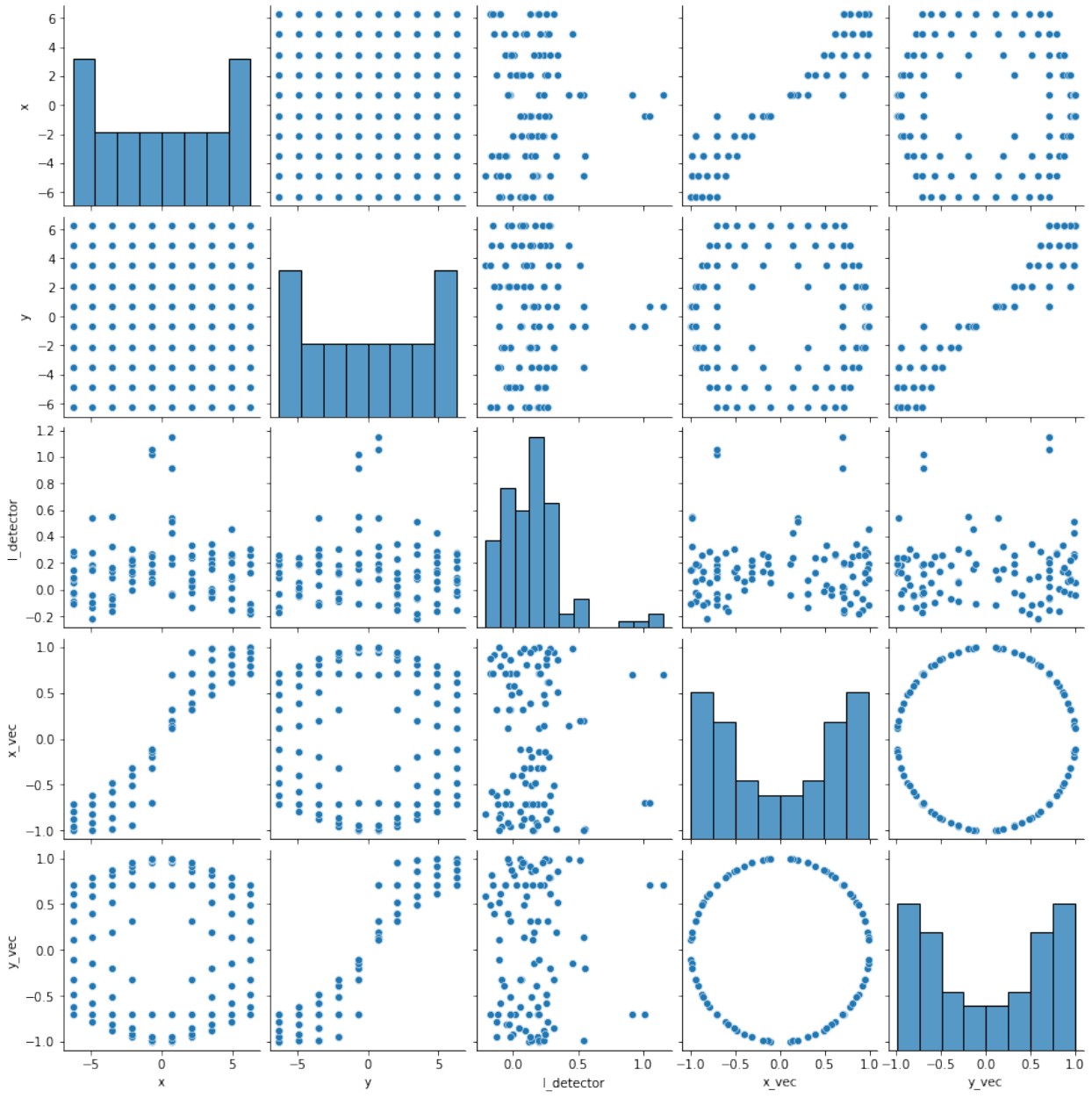
```
nx = 10
ny = 10
xx, yy = np.meshgrid(np.linspace(-2*np.pi, 2*np.pi, nx),
                     np.linspace(-2*np.pi, 2*np.pi, ny))

intensity_img = 1.5*np.abs(np.cos(xx*yy))/(np.abs(xx*yy)+(3*np.pi/nx))+np.random.
↪uniform(-0.25, 0.25, size = xx.shape)

base_df = pd.DataFrame(dict(x = xx.ravel(),
                            y = yy.ravel(),
                            I_detector = intensity_img.ravel()))

base_df['x_vec'] = base_df.apply(lambda c_row: c_row['x']/np.sqrt(1e-2+np.square(c_
↪row['x'])+np.square(c_row['y'])), 1)
base_df['y_vec'] = base_df.apply(lambda c_row: c_row['y']/np.sqrt(1e-2+np.square(c_
↪row['x'])+np.square(c_row['y'])), 1)

base_df.sample(5)
```

	x	y	I_detector	x_vec	y_vec
69	6.283185	2.094395	0.128002	0.948575	0.316192
78	4.886922	3.490659	0.101807	0.813621	0.581158
83	-2.094395	4.886922	0.061300	-0.393850	0.918982
22	-3.490659	-3.490659	-0.117917	-0.706962	-0.706962
52	-3.490659	0.698132	0.329619	-0.980194	0.196039

10.1 Looking at colocation histograms

The colocation histogram is a powerful tool to visualize how different components are related to each other. It also called bi-variate histogram. In seaborn, there is the `pairplot` which shows colocation histograms for all combinations on the data. The diagonal is the histogram of the individual components.

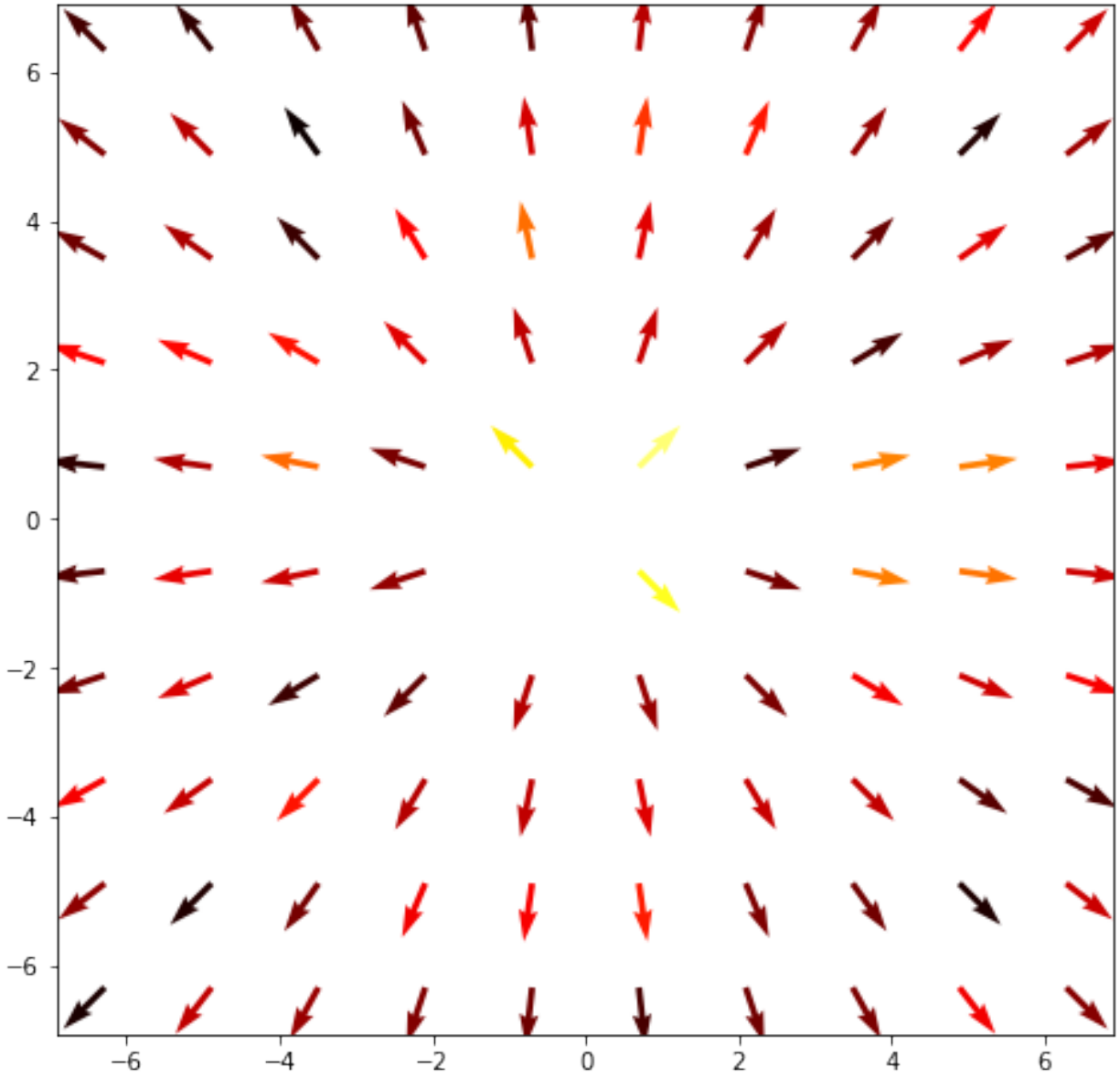
```
import seaborn as sns
sns.pairplot(base_df);
```



10.2 Vector field plot

The vector field is a common way to visualiz vector data. It does however only work for small data sets like in this example, otherwise it will be too cluttered and no relevant information will be visible.

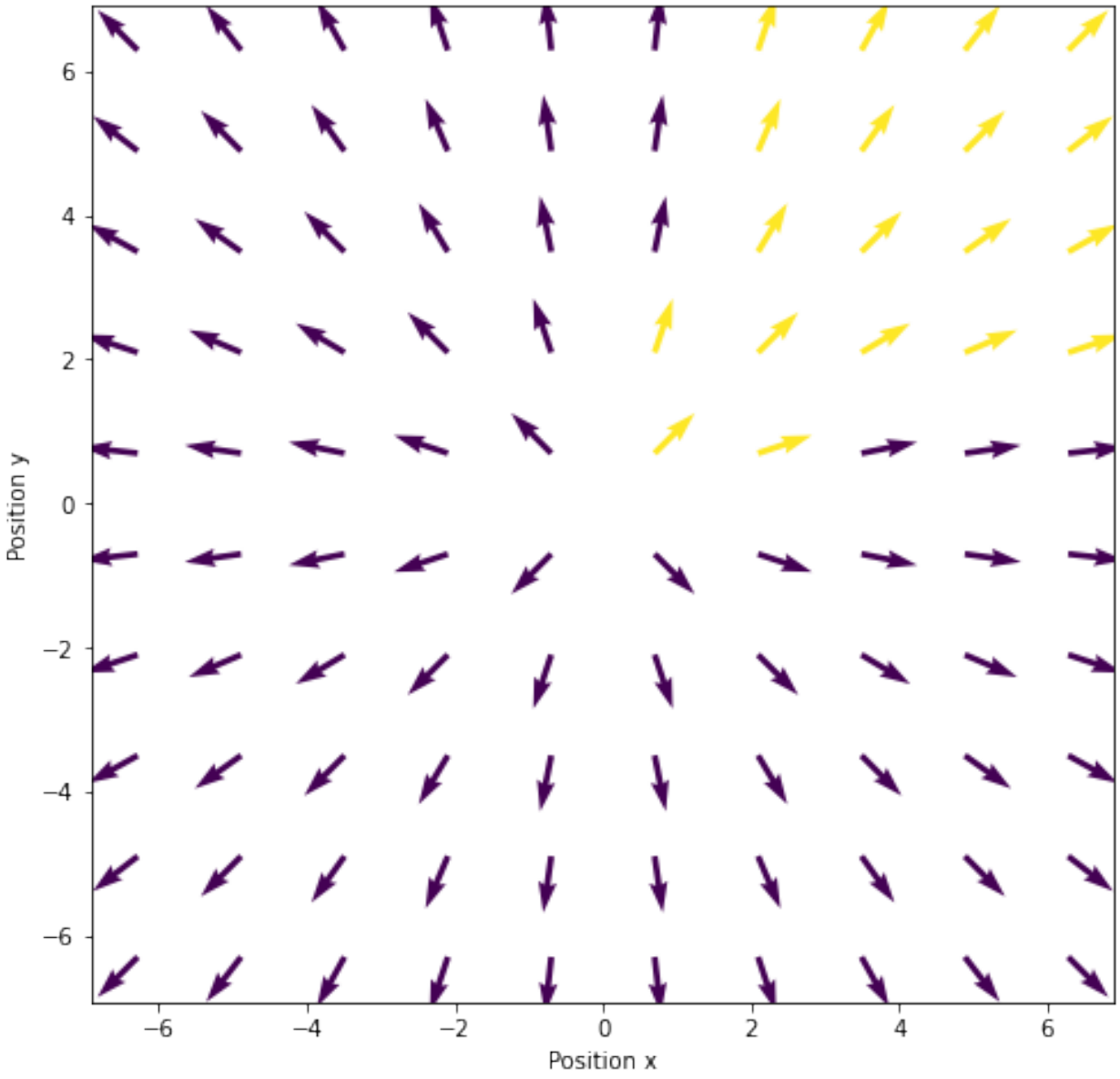
```
fig, ax1 = plt.subplots(1,1, figsize = (8, 8))
ax1.quiver(base_df['x'], base_df['y'], base_df['x_vec'], base_df['y_vec'], base_df['I_
↪detector'], cmap = 'hot');
```



10.3 Applying a threshold to vector valued image

A threshold is now more difficult to apply since there are now two distinct variables to deal with. The standard approach can be applied to both $I(x,y) = \begin{cases} 1, & \vec{f_x}(x,y) \geq 0.25 \text{ and } \\ & \vec{f_y}(x,y) \geq 0.25 \\ 0, & \text{otherwise} \end{cases}$

```
thresh_df = base_df.copy()
thresh_df['thresh'] = thresh_df.apply(lambda c_row: c_row['x_vec']>0.25 and c_row['y_
↪vec']>0.25, 1)

fig, ax1 = plt.subplots(1,1, figsize = (8, 8))
ax1.quiver(thresh_df['x'], thresh_df['y'], thresh_df['x_vec'], thresh_df['y_vec'],␣
↪thresh_df['thresh']);
ax1.set_xlabel('Position x'); ax1.set_ylabel('Position y');
```

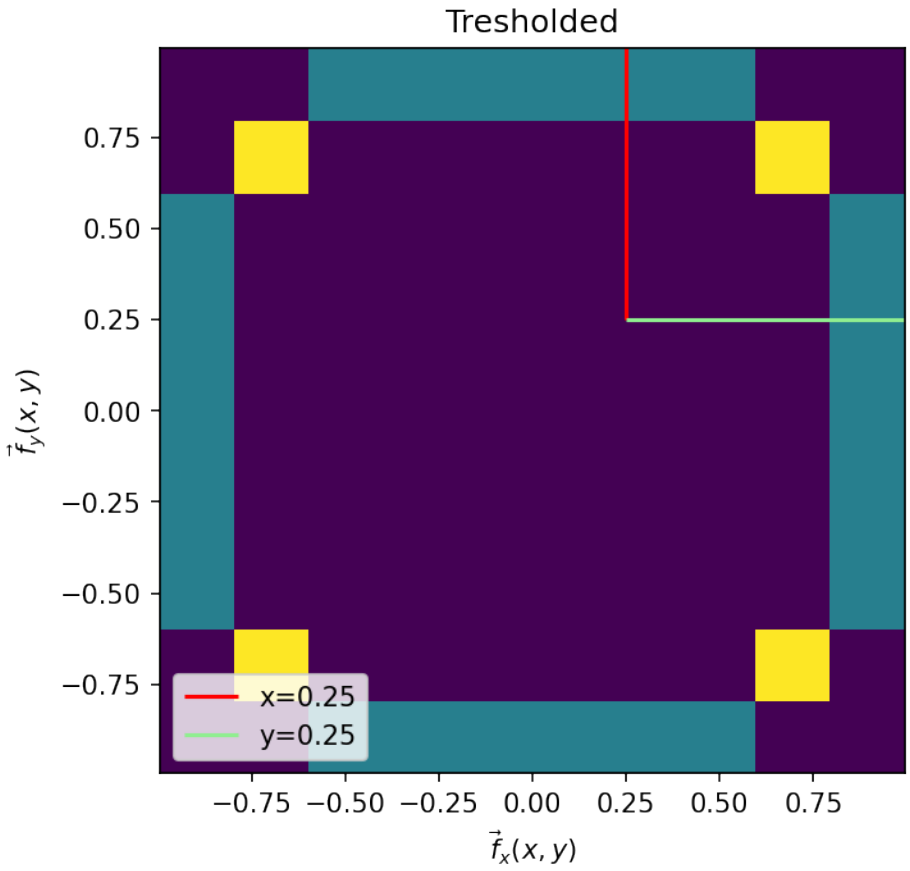


10.3.1 Histogram of the vectors

This can also be shown on the joint probability distribution as a bivariate histogram.

The lines here indicate the thresholded vector components.

```
fig, ax = plt.subplots(1,1, figsize = (5, 5), dpi = 150)
ax.hist2d(thresh_df['x_vec'], thresh_df['y_vec'], cmap = 'viridis'); ax.set_title(
↪'Tresholded');
ax.set_xlabel('$\\vec{f}_x(x,y)$'); ax.set_ylabel('$\\vec{f}_y(x,y)$');
ax.vlines(0.25,ymin=0.25,ymax=1,color='red',label='x=0.25');ax.hlines(0.25,xmin=0.25,
↪xmax=1,color='lightgreen', label='y=0.25');ax.legend(loc='lower left');
```



10.3.2 Applying a threshold

Given the presence of two variables; however, more advanced approaches can also be investigated. For example we can keep only components parallel to the x axis by using the dot product. $$I(x,y) = \begin{cases} 1, & |\vec{f}(x,y)\cdot\vec{i}| = 1 \\ 0, & \text{otherwise} \end{cases}$$

10.3.3 Thresholding orientations

We can tune the angular acceptance by using the fact that the scalar product can be expressed using the angle between the the vectors as

Scalar product definition $$\vec{x}\cdot\vec{y} = |\vec{x}||\vec{y}|\cos(\theta_{x\to y}) < br/ >< br/ > I(x,y) = \begin{cases} 1, & \cos^{-1}(\vec{f}(x,y)\cdot\vec{i}) \leq \theta^\circ \\ 0, & \text{otherwise} \end{cases}$$

Basic Segmentation and Discrete Binary Structures

Quantitative Big Imaging ETHZ: 227-0966-00L

Part 2

A Machine Learning Approach to Image Processing

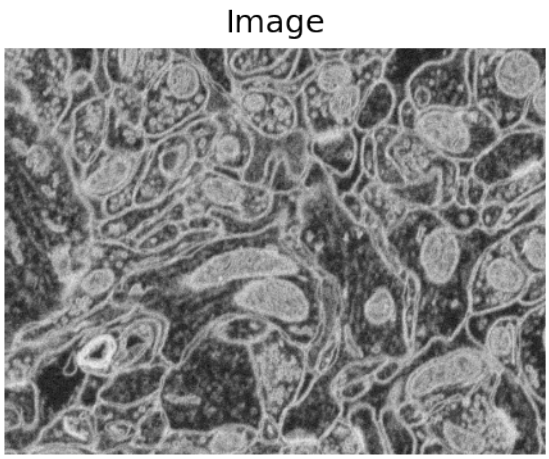
Segmentation and all the steps leading up to it are really a specialized type of learning problem.

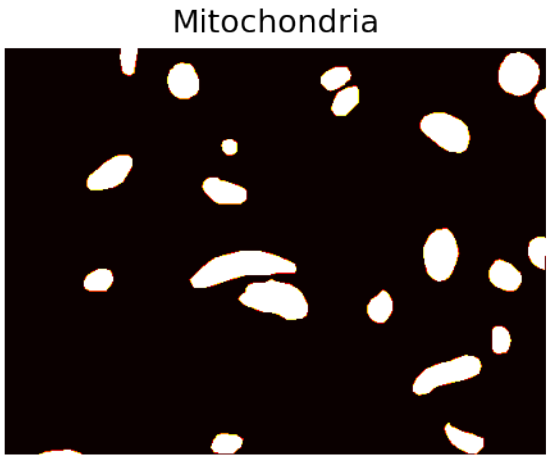
Let's look at an important problem for electron microscopy imaging…

```
import numpy as np
import matplotlib.pyplot as plt
from skimage.color import rgb2gray
from skimage.io import imread
%matplotlib inline
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-1-33a394319bb8> in <module>
      1 import numpy as np
      2 import matplotlib.pyplot as plt
----> 3 from skimage.color import rgb2gray
      4 from skimage.io import imread
      5 get_ipython().run_line_magic('matplotlib', 'inline')

ModuleNotFoundError: No module named 'skimage'
```

```
cell_img = (255-imread("data/em_image.png")[::2, ::2])/255.0
cell_seg = imread("data/em_image_seg.png")[::2, ::2]>0

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4), dpi=150)
ax1.imshow(cell_img, cmap='gray'); ax1.set_title('Image');         ax1.axis('off');
ax2.imshow(cell_seg, cmap='hot');  ax2.set_title('Mitochondria'); ax2.axis('off');
```





We want to identify which class each pixel belongs to.

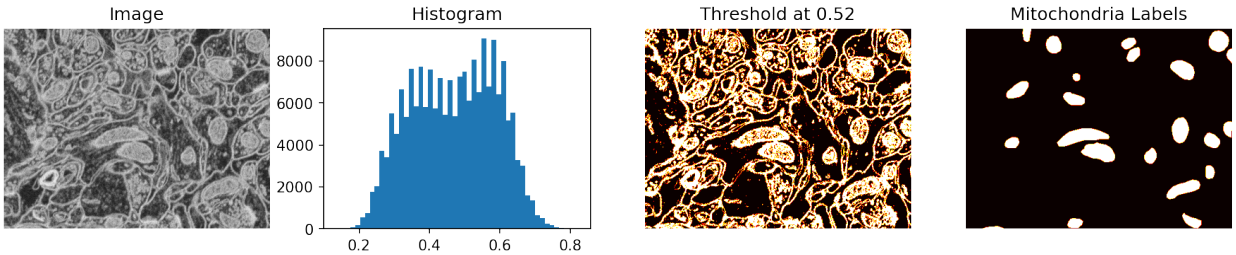
What does identify mean?

- Classify the pixels in a mitochondria as *Foreground*
- Classify the pixels outside of a mitochondria as *Background*

How do we quantify this?

- **True Positive** values in the mitochondria that are classified as *Foreground*
- **True Negative** values outside the mitochondria that are classified as *Background*
- **False Positive** values outside the mitochondria that are classified as *Foreground*
- **False Negative** values in the mitochondria that are classified as *Background*

```
fig, ax = plt.subplots(1, 4, figsize=(15, 2.5), dpi=150)

ax[0].imshow(cell_img, cmap='gray');  ax[0].set_title('Image');     ax[0].axis('off')
ax[1].hist(cell_img.ravel(),bins=50); ax[1].set_title('Histogram')

thresh     = 0.52
thresh_img = cell_img > thresh # Apply a single threshold

ax[2].imshow(thresh_img, cmap='hot');   ax[2].set_title('Threshold at {0}'.
↪format(thresh)); ax[2].axis('off')
ax[3].imshow(cell_seg,   cmap='hot');   ax[3].set_title('Mitochondria Labels');
↪       ax[3].axis('off');
```

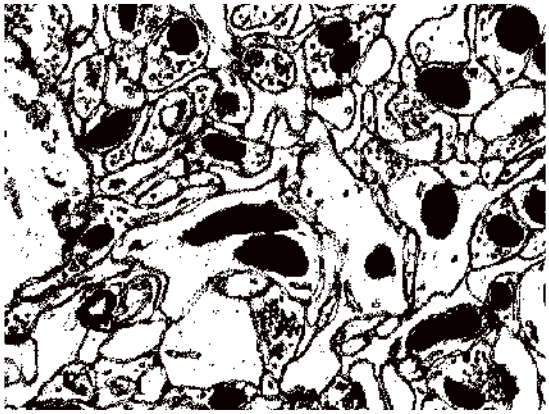


Check the performance of the thresholding

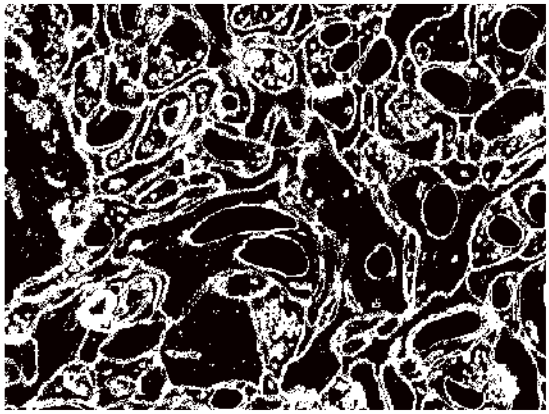
```
# Suport function for the plot labels
def tp_func(real_img_idx, pred_img_idx):
    if real_img_idx == 1 and pred_img_idx == 1:
        return 'True Positive','green'
    if real_img_idx == 0 and pred_img_idx == 0:
        return 'True Negative','green'
    if real_img_idx == 0 and pred_img_idx == 1:
        return 'False Positive','red'
    if real_img_idx == 1 and pred_img_idx == 0:
        return 'False Negative','red'
```

```
out_results = {}
fig, m_ax = plt.subplots(2, 2, figsize=(8, 7), dpi=150)
for real_img_idx, n_ax in zip([0, 1], m_ax):
    for pred_img_idx, c_ax in zip([0, 1], n_ax):
        match_img = (thresh_img == pred_img_idx) & (cell_seg == real_img_idx)
        (tp_title,color) = tp_func(real_img_idx, pred_img_idx)
        c_ax.matshow(match_img, cmap='hot')
        out_results[tp_title] = np.sum(match_img)
        c_ax.set_title("{0} ({1})".format(tp_title,out_results[tp_title]),color=color)
        c_ax.axis('off')
```

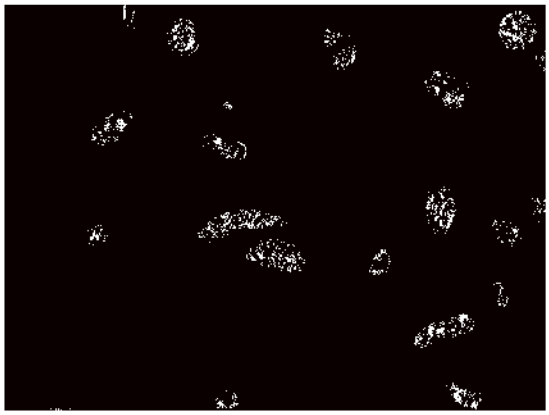
True Negative (118050)



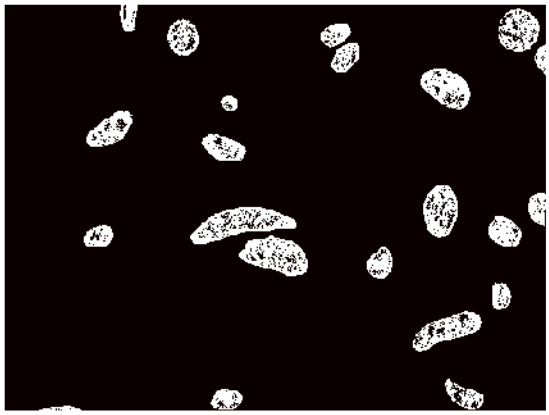
False Positive (61945)



False Negative (2932)



True Positive (13681)



Apply Precision and Recall

- **Recall** (sensitivity) $\frac{TP}{TP+FN}$
- **Precision** $\frac{TP}{TP+FP}$

```
print('Recall: {0:0.2f}'.format(out_results['True Positive'] /
                        (out_results['True Positive']+out_results['False Negative
↪'])))
print('Precision: {0:0.2f}'.format(out_results['True Positive'] /
                          (out_results['True Positive']+out_results['False Positive
↪'])))
```

```
Recall: 0.82
Precision: 0.18
```

The confusion matrix (revisited)

Confusion matrix

ROC Curve

Reciever Operating Characteristic (first developed for WW2 soldiers detecting objects in battlefields using radar).

The ideal is the top-right (identify everything and miss nothing).

As we saw before, for a single threshold value 0.5, we were able to compute a single recall and precision.

If we want to make an ROC curve we take a number of threshold values

```
import pandas as pd
from collections import OrderedDict

out_vals = []
for thresh_val in np.linspace(0.1, 0.9):
    thresh_img = cell_img > thresh_val
    for real_img_idx in [0, 1]:
        for pred_img_idx in [0, 1]:
            match_img = (thresh_img == pred_img_idx) & (
                cell_seg == real_img_idx)
            tp_title = tp_func(real_img_idx, pred_img_idx)
            out_results[tp_title] = np.sum(match_img)
    out_vals += [
        OrderedDict(
            Threshold=thresh_val,
            Recall=out_results['True Positive'] /
            (out_results['True Positive']+out_results['False Negative']),
            Precision=(out_results['True Positive'] /
                       (out_results['True Positive']+out_results['False Positive'])),
            False_Positive_Rate=(out_results['False Positive'] /
                       (out_results['False Positive']+out_results['True Negative'])),
            **out_results
        )]

roc_df = pd.DataFrame(out_vals)
roc_df.head(3)
```

```
   Threshold    Recall  Precision  False_Positive_Rate  True Negative  \
0   0.100000  0.823512   0.180903             0.344148         118050
1   0.116327  0.823512   0.180903             0.344148         118050
2   0.132653  0.823512   0.180903             0.344148         118050

   False Positive  False Negative  True Positive  (True Negative, green)  \
0           61945            2932          13681                      0
1           61945            2932          13681                      0
2           61945            2932          13681                      0

   (False Positive, red)  (False Negative, red)  (True Positive, green)
0                 179995                      0                   16613
1                 179995                      0                   16613
2                 179995                      0                   16613
```

Making ROC Curves Easier

ROC curves are a very common tool for analyzing the performance of binary classification systems and there are a large number of tools which can automatically make them. Here we show how it is done with scikit-image.

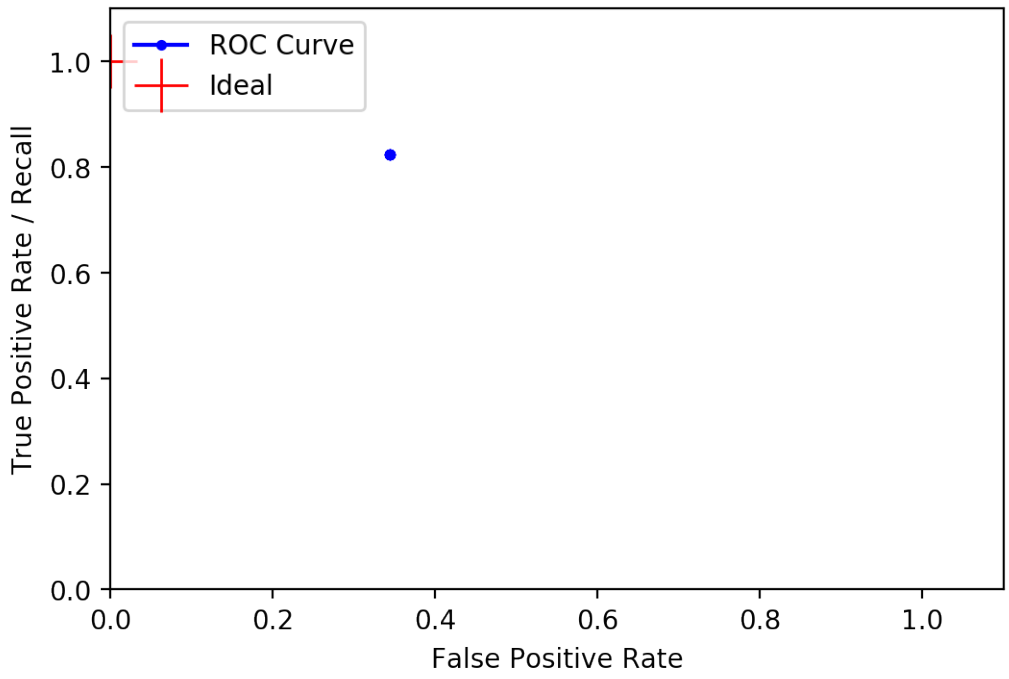
Another way of showing the ROC curve (more common for machine learning rather than medical diagnosis) is using the True positive rate and False positive rate

- **True Positive Rate** (recall)= $TP/(TP+FN)$
- **False Positive Rate** = $FP/(FP+TN)$

These show very similar information with the major difference being the goal is to be in the upper left-hand corner. Additionally random guesses can be shown as the slope 1 line. Therefore for a system to be useful it must lie above the random line.

```
fig, ax1 = plt.subplots(1, 1, dpi=200)
ax1.plot(roc_df['False_Positive_Rate'], roc_df['Recall']  , 'b.-', label='ROC Curve')
ax1.plot(0, 1.0, 'r+', markersize=20, label='Ideal')
ax1.set_xlim(0, 1.1)
ax1.set_ylim(0, 1.1)
ax1.set_ylabel('True Positive Rate / Recall')
ax1.set_xlabel('False Positive Rate')
ax1.legend(loc=2)
```

```
<matplotlib.legend.Legend at 0x1c181c51d0>
```

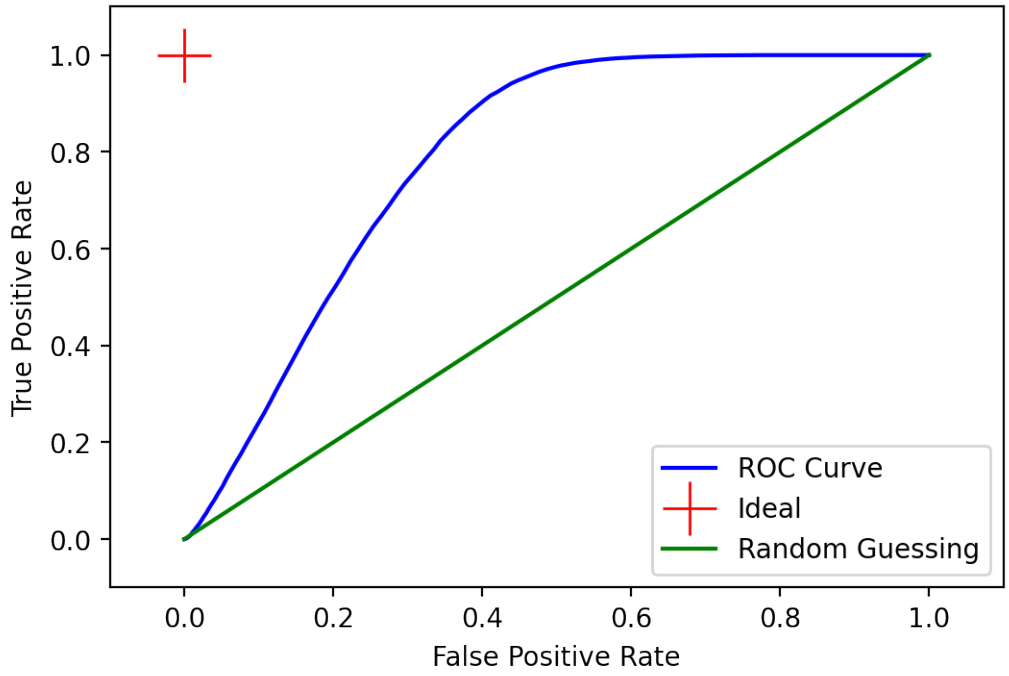



```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(cell_seg.ravel().astype(int),
                                 cell_img.ravel())

fig, ax1 = plt.subplots(1, 1, dpi=200)
ax1.plot(fpr, tpr, 'b.-', markersize=0.01,  label='ROC Curve')
```

(continues on next page)

(continued from previous page)

```
ax1.plot(0.0, 1.0, 'r+', markersize=20, label='Ideal')
ax1.plot([0, 1], [0, 1], 'g-', label='Random Guessing')
ax1.set_xlim(-0.1, 1.1)
ax1.set_ylim(-0.1, 1.1)
ax1.set_xlabel('False Positive Rate')
ax1.set_ylabel('True Positive Rate')
ax1.legend(loc=0);
```

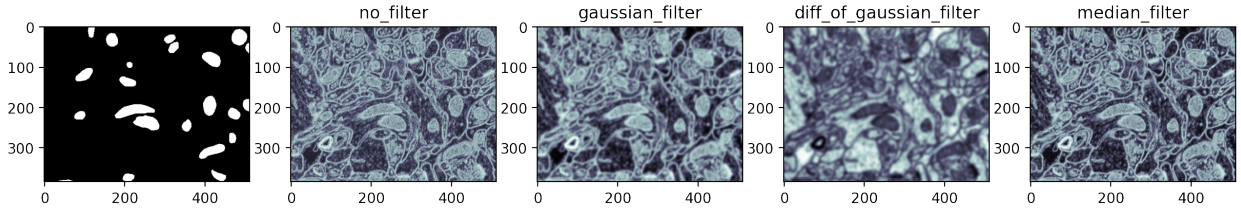



```
from skimage.filters import gaussian, median


def no_filter(x):
    return x


def gaussian_filter(x):
    return gaussian(x, sigma=2)


def diff_of_gaussian_filter(x):
    return   -gaussian(x, sigma=3)


def median_filter(x):
    return median(x, np.ones((3, 3)))


fig, m_axs = plt.subplots(1, 5, figsize=(15, 3), dpi=200)
m_axs[0].imshow(cell_seg, cmap='gray')
for c_filt, c_ax in zip([no_filter, gaussian_filter, diff_of_gaussian_filter, median_
↪filter], m_axs[1:]):
    c_ax.imshow(c_filt(cell_img), cmap='bone')
```

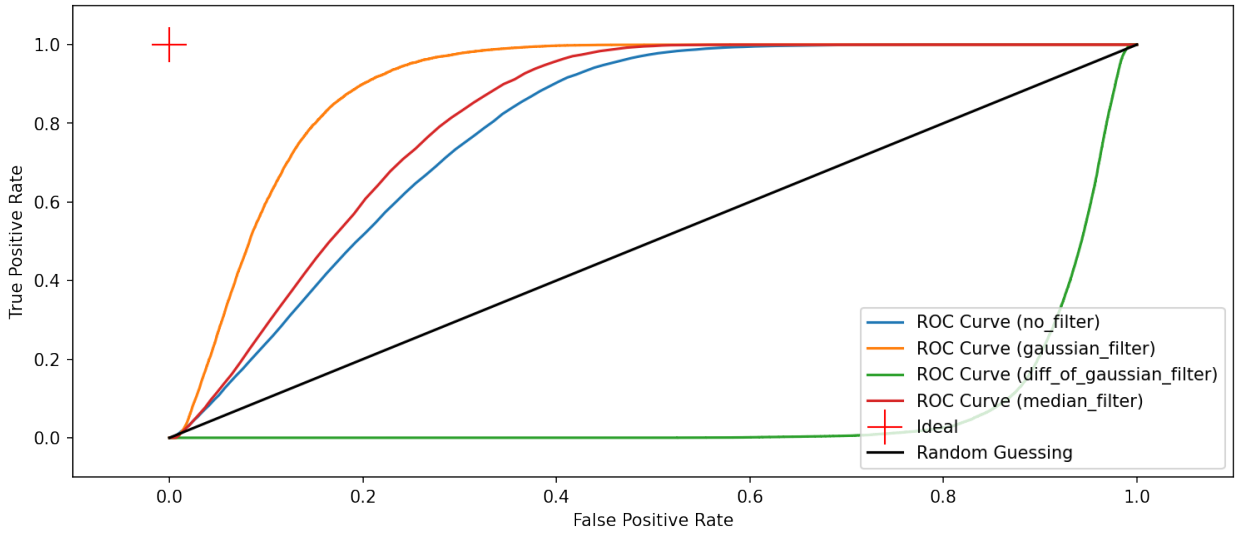
(continues on next page)

(continued from previous page)

```
    c_ax.set_title(c_filt.__name__)
```




```
fig, ax1 = plt.subplots(1, 1,figsize=(12,5), dpi=150)
for c_filt in [no_filter, gaussian_filter, diff_of_gaussian_filter, median_filter]:
    fpr, tpr, thresholds = roc_curve(cell_seg.ravel().astype(int),
                                     c_filt(cell_img).ravel())
    ax1.plot(fpr, tpr, '-', markersize=0.01,
             label='ROC Curve ({})'.format(c_filt.__name__))

ax1.plot(0.0, 1.0, 'r+', markersize=20, label='Ideal')
ax1.plot([0, 1], [0, 1], 'k-', label='Random Guessing')
ax1.set_xlim(-0.1, 1.1)
ax1.set_ylim(-0.1, 1.1)
ax1.set_xlabel('False Positive Rate')
ax1.set_ylabel('True Positive Rate')
ax1.legend(loc="lower right", fontsize=10);
```



We can then use this ROC curve to compare different filters (or even entire workflows), if the area is higher the approach is better.

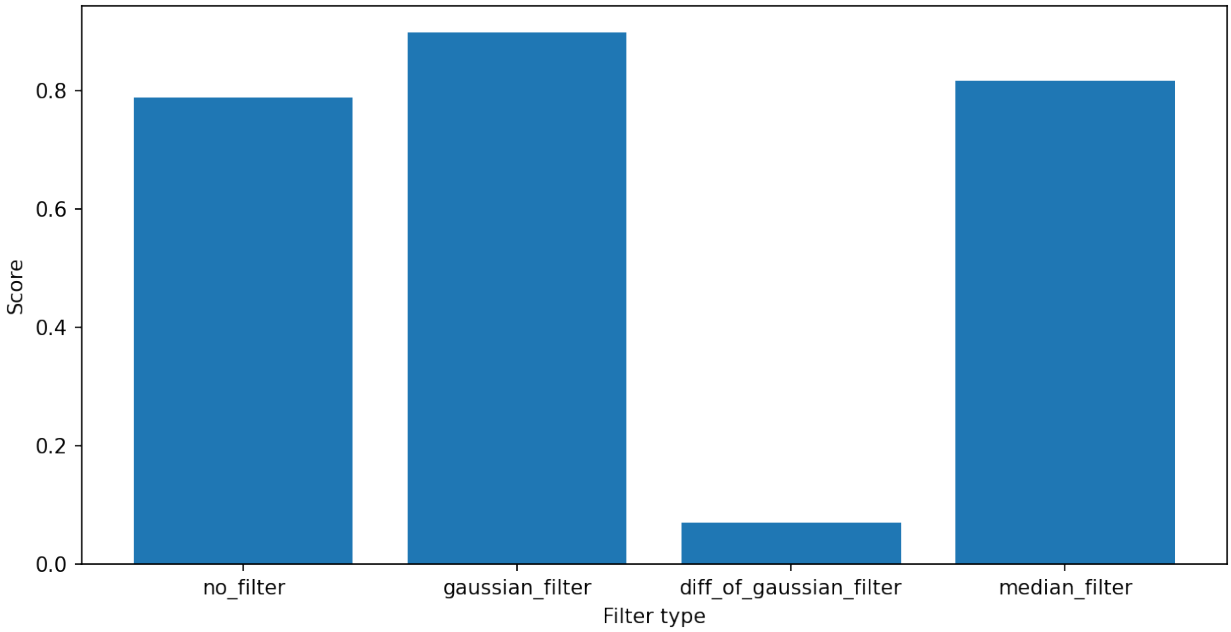
Different approaches can be compared by area under the curve

```
from sklearn.metrics import roc_auc_score
scores = []
names = ['no_filter', 'gaussian_filter', 'diff_of_gaussian_filter', 'median_filter']
for c_filt in [no_filter, gaussian_filter, diff_of_gaussian_filter, median_filter]:
    scores.append(roc_auc_score(cell_seg.ravel().astype(int),  c_filt(cell_img).
↪ravel()))
#    print('%s - %2.2f' % (c_filt.__name__, roc_auc_score(cell_seg.ravel().
↪astype(int),
```

(continues on next page)

(continued from previous page)

```
#                                                    c_filt(cell_img).ravel())))

plt.figure(figsize=[10,5],dpi=150)
plt.bar(names,scores); plt.xlabel('Filter type'),plt.ylabel('Score');
```



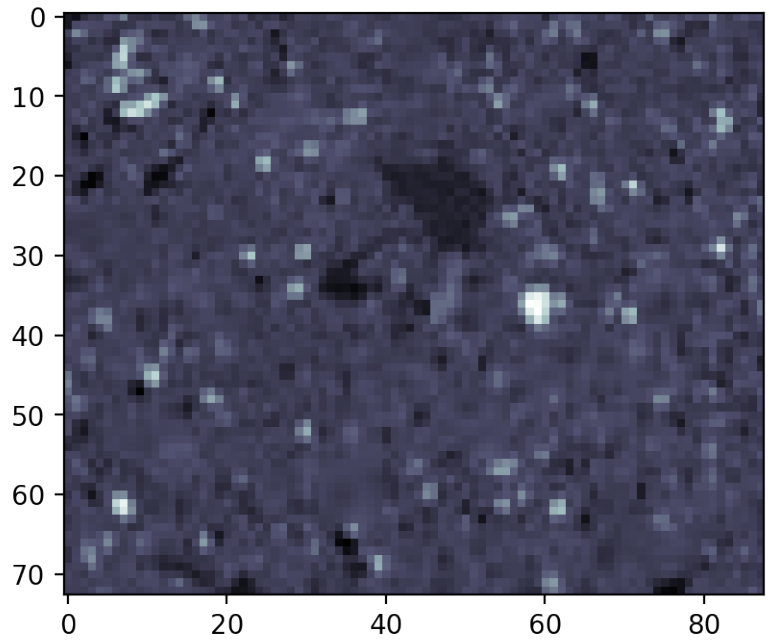
Evaluating Models

- https://github.com/jvns/talks/blob/master/pydatanyc2014/slides.md
- http://mathbabe.org/2012/03/06/the-value-added-teacher-model-sucks/

Multiple Phases: Segmenting Shale

- Shale provided from Kanitpanyacharoen, W. (2012). Synchrotron X-ray Applications Toward an Understanding of Elastic Anisotropy.
- Here we have a shale sample measured with X-ray tomography with three different phases inside (clay, rock, and air).
- The model is that because the chemical composition and density of each phase is different they will absorb different amounts of x-rays and appear as different brightnesses in the image

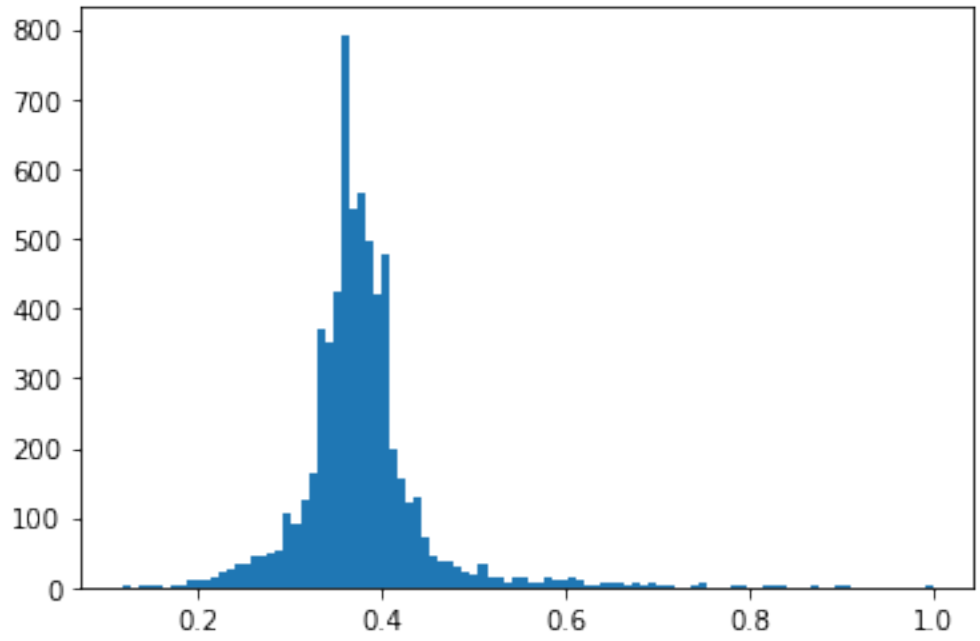
```
import numpy as np
import matplotlib.pyplot as plt
from skimage.color import rgb2gray
from skimage.io import imread
%matplotlib inline
shale_img = imread("figures/ShaleSample.jpg")/255.0
fig, ax1 = plt.subplots(1, 1, dpi=200)
ax1.imshow(shale_img, cmap='bone');
```



Ideally we would derive 3 values for the thresholds based on a model for the composition of each phase and how much it absorbs, but that is not always possible or practical.

- While there are 3 phases clearly visible in the image, the histogram is less telling (even after being re-scaled).

```
plt.hist(shale_img.ravel(), 100);
```

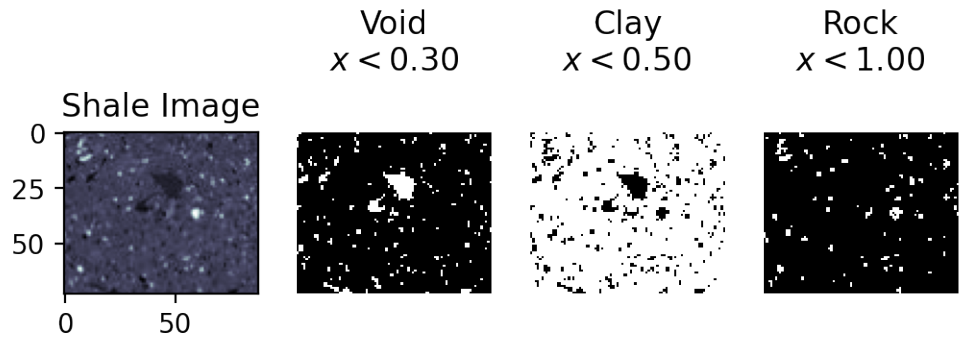


Multiple Segmentations

For this exercise we choose arbitrarily 3 ranges for the different phases and perform visual inspection

The relation can explicitly be written out as $$I(x) = \begin{cases} \text{Void}, & 0 \leq x \leq 0.3 \\ \text{Clay}, & 0.3 < x \leq 0.5 \\ \text{Rock}, & 0.5 < x \end{cases}$$

```
fig, m_axs = plt.subplots(1, 4, dpi=200, figsize=(6, 3))
m_axs[0].imshow(shale_img, cmap='bone')
m_axs[0].set_title('Shale Image')
used_vox = np.zeros_like(shale_img).astype(np.uint8)
for c_ax, c_max, c_title in zip(m_axs[1:], [0.3, 0.5, 1.0], ['Void', 'Clay', 'Rock']):
    c_slice = (shale_img < c_max)-used_vox
    c_ax.matshow(c_slice, cmap='bone')
    used_vox += c_slice
    c_ax.axis('off')
    c_ax.set_title('%s\n$x<%2.2f$' % (c_title, c_max))
```



Implementation

The implementations of basic thresholds and segmentations is very easy since it is a unary operation of a single image $$f(I(\vec{x}))$$ In mathematical terms this is called a map and since it does not require information from neighboring voxels or images it can be calculated for each point independently (*parallel*). Filters on the other hand almost always depend on neighboring voxels and thus the calculations are not as easy to seperate.

Implementation Code

Matlab / Python (numpy)

The simplist is a single threshold in Matlab:

```
thresh_img = gray_img > thresh
```

A more complicated threshold:

```
thresh_img = (gray_img > thresh_a) & (gray_img < thresh_b)
```

Python

```
thresh_img = map(lambda gray_val: gray_val>thresh,
                 gray_img)
```

Java

```
boolean[] thresh_img = new boolean[x_size*y_size*z_size];
for(int x=x_min;x<x_max;x++)
  for(int y=y_min;y<y_max;y++)
    for(int z=z_min;z<z_max;z++) {
      int offset=(z*y_size+y)*x_size+x;
      thresh_img[offset]=gray_img[offset]>thresh;
    }
```

In C/C++

```
bool* thresh_img = malloc(x_size*y_size*z_size * sizeof (bool));

for(int x=x_min;x<x_max;x++)
  for(int y=y_min;y<y_max;y++)
    for(int z=z_min;z<z_max;z++) {
      int offset=(z*y_size+y)*x_size+x;
      thresh_img[offset]=gray_img[offset]>thresh;
    }
```

Morphology

We can now utilize information from neighborhood voxels to improve the results. These steps are called morphological operations. We return to the original image of a cross

Like filtering the assumption behind morphological operations are

- nearby voxels in **real** images are related / strongly correlated with one another
- noise and imaging artifacts are less spatially correlated.

Therefore these imaging problems can be alleviated by adjusting the balance between local and neighborhood values.

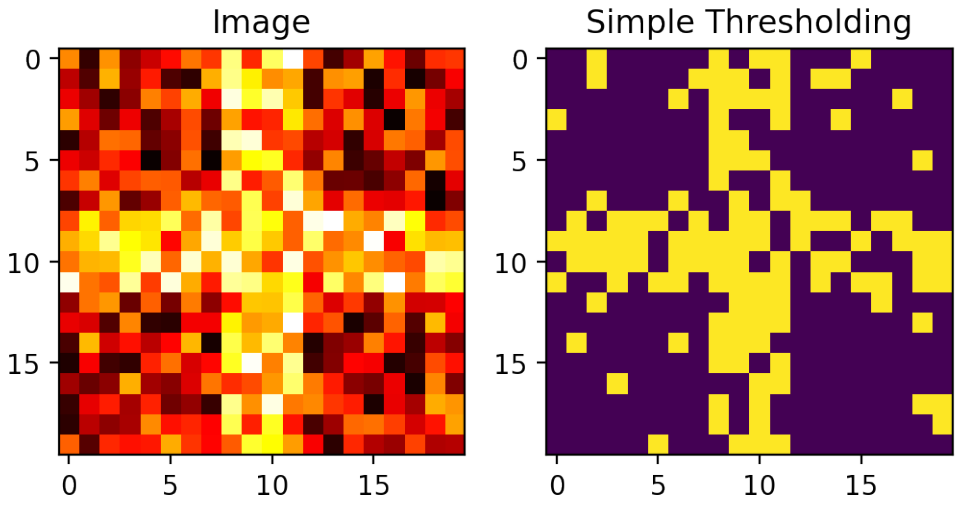
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

nx = 20
ny = 20
xx, yy = np.meshgrid(np.linspace(-10, 10, nx),
                     np.linspace(-10, 10, ny))
np.random.seed(2018)
cross_im = 1.1*((np.abs(xx) < 2)+(np.abs(yy) < 2)) + \
    np.random.uniform(-1.0, 1.0, size=xx.shape)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(6, 3.5), dpi=200)
```

(continues on next page)

(continued from previous page)

```
ax1.imshow(cross_im, cmap='hot')
ax1.set_title('Image')
ax2.imshow(cross_im > 0.8)
ax2.set_title('Simple Thresholding')
```

```
Text(0.5, 1.0, 'Simple Thresholding')
```



Fundamentals: Neighborhood

A neighborhood consists of the pixels or voxels which are of sufficient proximity to a given point. There are a number of possible definitions which largely affect the result when it is invoked.

- A large neighborhood performs operations over larger areas / volumes
- Computationally intensive
- Can *smooth* out features
- A small neighborhood performs operations over small areas / volumes
- Computationally cheaper
- Struggles with large noise / filling large holes

The neighborhood is important for a large number of image and other (communication, mapping, networking) processing operations:

- filtering
- morphological operations
- component labeling
- distance maps
- image correlation based tracking methods

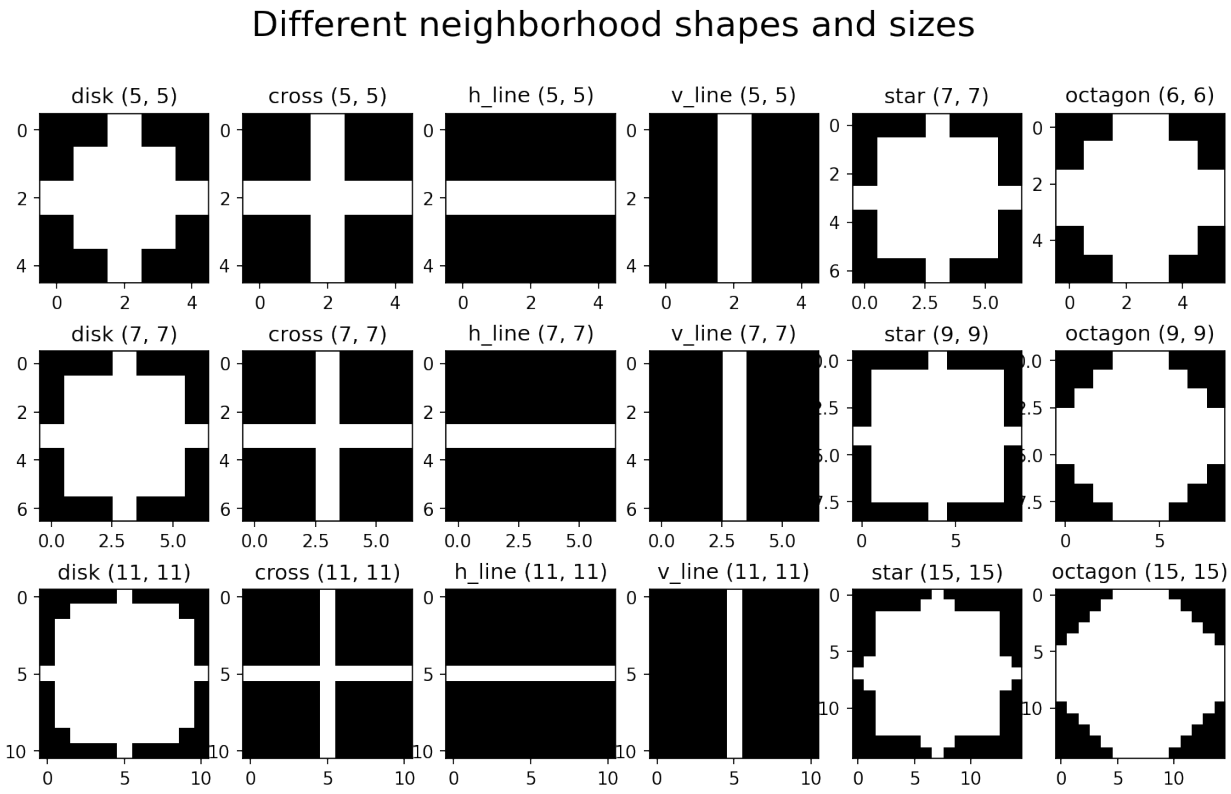
It is often called structuring element (or `selem` for sort / code), but has exactly the same meaning

Fundamentals: Neighbors in 2D

For standard image operations there are two definitions of neighborhood. The 4 and 8 adjacent neighbors shown below. Given the blue pixel in the center the red are the 4-adjacent and the red and green make up the 8 adjacent. We expand beyond this to disk, cross, vertical and horizontal lines

```
from skimage.morphology import disk, octagon as oct_func, star


def h_line(n):
    return np.pad(np.ones((1, 2*n+1)), [[n, n], [0, 0]], mode='constant', constant_
↪values=0).astype(int)


def v_line(n):
    return h_line(n).T


def cross(n):
    return ((h_line(n)+v_line(n)) > 0).astype(int)


def octagon(n):
    return oct_func(n, n)
```

```
neighbor_functions = [disk, cross, h_line, v_line, star, octagon]
sizes = [2, 3, 5]
fig, m_axs = plt.subplots(len(sizes),len(neighbor_functions),
                          figsize=(12, 7),dpi=150)
for c_dim, c_axs in zip(sizes, m_axs):
    for c_func, c_ax in zip(neighbor_functions, c_axs):
        c_ax.imshow(c_func(c_dim), cmap='bone', interpolation='none')
        c_ax.set_title('{} {}'.format(c_func.__name__, c_func(c_dim).shape))

plt.suptitle('Different neighborhood shapes and sizes', fontsize=20);
```



Erosion and Dilation

Erosion

If any of the voxels in the neighborhood are 0/false than the voxel will be set to 0

- Has the effect of peeling the surface layer off of an object

---

Dilation

If any of the voxels in the neigbhorhood are 1/true then the voxel will be set to 1

- Has the effect of adding a layer onto an object (dunking an strawberry in chocolate, adding a coat of paint to a car)

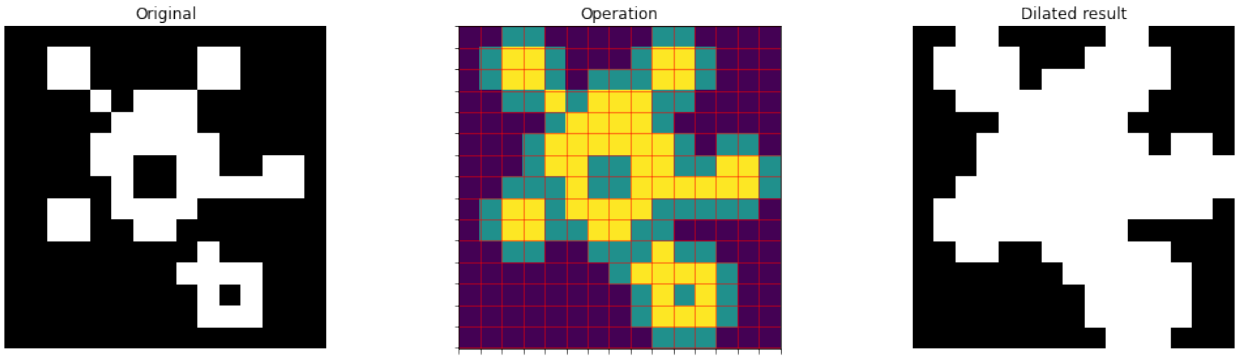
Applied Erosion and Dilation

```
import numpy as np
import matplotlib.pyplot as plt
import skimage.morphology as morph

img=np.load('data/morphimage.npy')

oimg=morph.opening(img,np.array([[0,1,0],[1,1,1],[0,1,0]]))
cimg=morph.closing(img,np.array([[0,1,0],[1,1,1],[0,1,0]]))
s=255.0
cmap = [[230/s,230/s,230/s],
        [255/s,176/s,159/s],
        [0.0/s,0.0/s,0.0/s]]
```

Dilation

We can use dilation to expand objects, for example a too-low threshold value leading to disconnected components

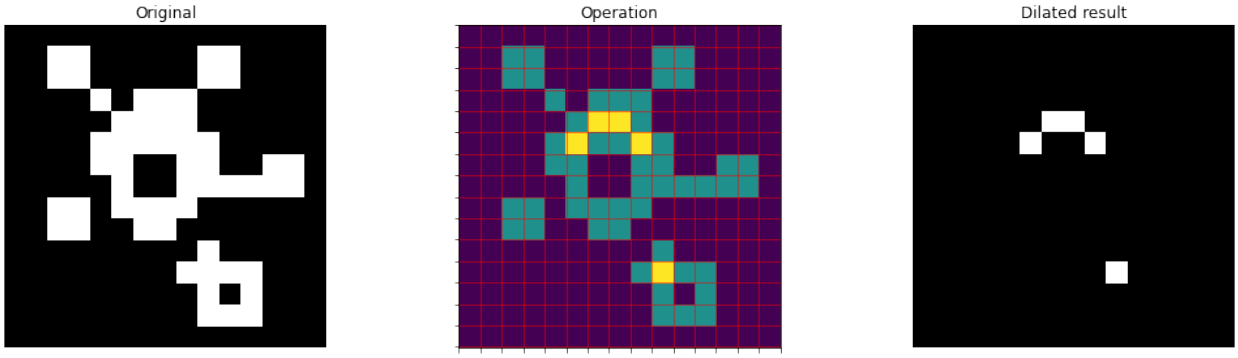
```
dimg=morph.dilation(img,[[0,1,0],[1,1,1],[0,1,0]])

fig, ax = plt.subplots(1,3,figsize=(15,4))

ax[0].imshow(img,cmap='gray'); ax[0].set_title('Original'); ax[0].axis('off');

ax[1].imshow(img+dimg,cmap='viridis');
ax[1].set_xticks(np.arange(-0.5,img.shape[1],1)); ax[1].set_xticklabels([]);ax[1].set_
↪yticks(np.arange(-0.55,img.shape[0],1)); ax[1].set_yticklabels([])
ax[1].grid(color='red', linestyle='-', linewidth=0.5); ax[1].grid(True);ax[1].set_
↪title('Operation')

ax[2].imshow(dimg,cmap='gray'); ax[2].set_title('Dilated result');ax[2].axis('off');
plt.tight_layout()
```



Erosion

Erosion performs the opposite task reducing the size

```
eimg=morph.erosion(img,[[0,1,0],[1,1,1],[0,1,0]])
fig, ax = plt.subplots(1,3,figsize=(15,4))

ax[0].imshow(img,cmap='gray'); ax[0].set_title('Original'); ax[0].axis('off');

ax[1].imshow(img+eimg,cmap='viridis');
ax[1].set_xticks(np.arange(-0.5,img.shape[1],1)); ax[1].set_xticklabels([]);ax[1].set_
↪yticks(np.arange(-0.55,img.shape[0],1)); ax[1].set_yticklabels([])
ax[1].grid(color='red', linestyle='-', linewidth=0.5); ax[1].grid(True);ax[1].set_
↪title('Operation')

ax[2].imshow(eimg,cmap='gray'); ax[2].set_title('Dilated result');ax[2].axis('off');
plt.tight_layout()
```



Opening and Closing

Opening

An erosion followed by a dilation operation

- Peels a layer off and adds a layer on
- Very small objects and connections are deleted in the erosion and do not return the image is thus __open__ed
- A cube larger than several voxels will have the exact same volume after (conservative)

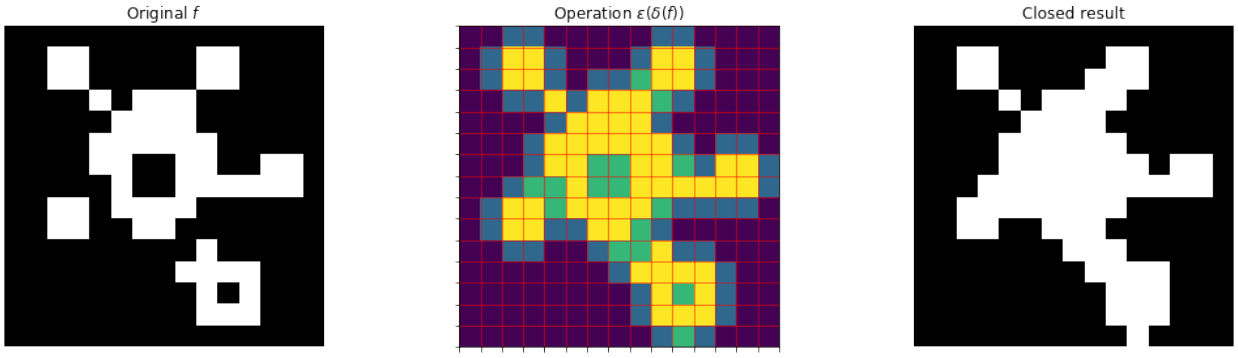
Closing

A dilation followed by an erosion operation

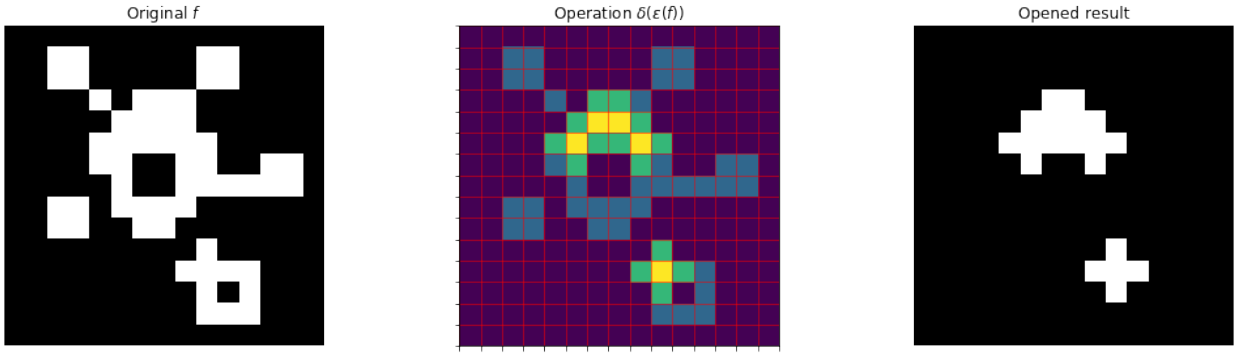
- Adds a layer and then peels a layer off
- Objects that are very close are connected when the layer is added and they stay connected when the layer is removed thus the image is __close__d
- A cube larger than one voxel will have the exact same volume after (conservative)

Morphological Closing

```
plt.figure(figsize=[15,4])
plt.subplot(1,3,1)
plt.imshow(img,cmap='gray')
plt.axis('off')
plt.title('Original $f$')
plt.subplot(1,3,2)
plt.imshow(img+dimg+cimg,cmap='viridis')
plt.title('Operation $\epsilon(\delta(f))$')
plt.xticks(np.arange(-0.5,img.shape[1],1),labels=[])
plt.yticks(np.arange(-0.55,img.shape[0],1),labels=[])
plt.grid(color='red', linestyle='-', linewidth=0.5)
plt.grid(True)
plt.subplot(1,3,3)
plt.imshow(cimg,cmap='gray')
plt.title('Closed result')
plt.axis('off')
plt.tight_layout()
```




```
plt.figure(figsize=[15,4])
plt.subplot(1,3,1)
plt.imshow(img,cmap='gray')
plt.axis('off')
plt.title('Original $f$')
plt.subplot(1,3,2)
plt.imshow(img+eimg+oimg,cmap='viridis')
plt.xticks(np.arange(-0.5,img.shape[1],1),labels=[])
plt.yticks(np.arange(-0.55,img.shape[0],1),labels=[])
plt.grid(color='red', linestyle='-', linewidth=0.5)
plt.grid(True)
plt.title('Operation $\delta(\epsilon(f))$')
plt.subplot(1,3,3)
plt.imshow(oimg,cmap='gray')
plt.axis('off')
plt.title('Opened result')
plt.tight_layout()
```



Pitfalls with Segmentation

Partial Volume Effect

- The partial volume effect is the name for the effect of discretization on the image into pixels or voxels.
- Surfaces are complicated, voxels are simple boxes which make poor representations
- Many voxels are only partially filled, but only the voxels on the surface
- Removing the first layer alleviates issue

When is a sphere really a sphere?

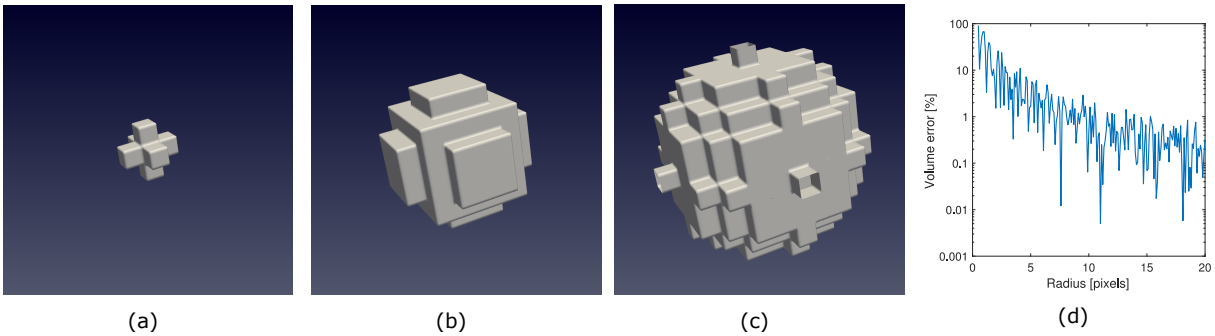


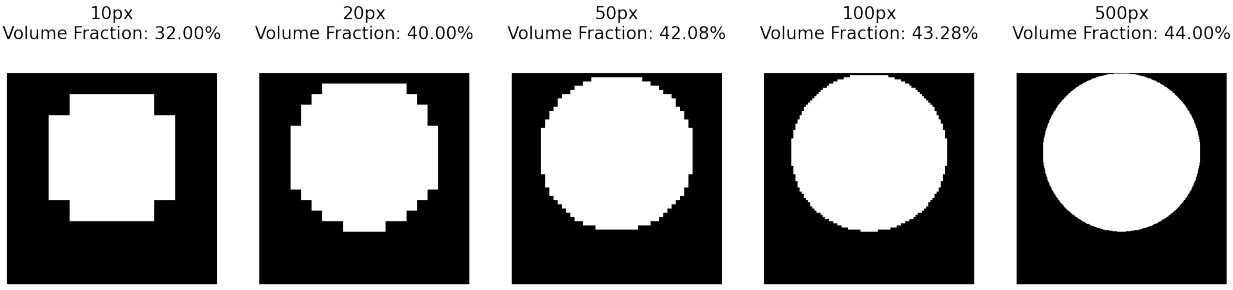
Fig. 10.1: Discrete spheres with increasing radius.

```
from scipy.ndimage import zoom
import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imread
%matplotlib inline
step_list = [10, 20, 50, 100, 500]
fig, m_axs = plt.subplots(1, len(step_list), figsize=(15, 5), dpi=200)
for c_ax, steps in zip(m_axs, step_list):
    x_lin = np.linspace(-1, 1, steps)
    xy_area = np.square(np.diff(x_lin)[0])
    xx, yy = np.meshgrid(x_lin, x_lin)
    test_img = (np.square(xx)+np.square(yy+0.25)) < np.square(0.75)
```

(continues on next page)

(continued from previous page)

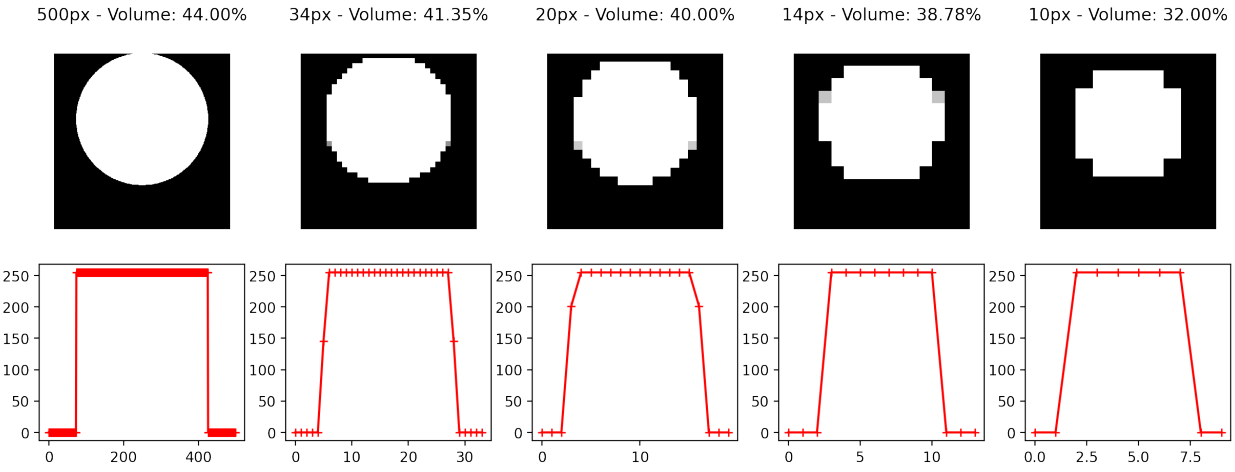
```
    c_ax.matshow(test_img,cmap='gray')
    c_ax.set_title('%dpx\nVolume Fraction: %2.2f%%' %
                   (steps, 100*np.sum(test_img)/np.prod(test_img.shape)))
    c_ax.axis('off')
```



Rescaling

We see the same effect when we rescale images from 500x500 down to 15x15 that the apparent volume fraction changes

```
zoom_level = [1, 0.067, 0.039, 0.029, 0.02]
fig, m_axs = plt.subplots(2, len(zoom_level), figsize=(15, 5), dpi=200)
for (c_ax, ax2), c_zoom in zip(m_axs.T, zoom_level):
    c_img = zoom(255.0*test_img, c_zoom, order=1)
    c_ax.matshow(c_img,cmap='gray')
    c_ax.set_title('%dpx - Volume: %2.2f%%' %
                   (c_img.shape[0], 100*np.sum(c_img > 0.5)/np.prod(c_img.shape)))
    c_ax.axis('off')
    ax2.plot(c_img[c_img.shape[0]//2], 'r+-')
```



Summary