Quantitative Big Imaging - Basic segmentation

Anders Kaestner

May 11, 2023

CONTENTS

This is the lecture notes for the 4th lecture of the Quantitative big imaging class given during the spring semester 2022 at ETH Zurich, Switzerland.

0.1 Image segmentation and discrete structures

Quantitative Big Imaging ETHZ: 227-0966-00L

Part 1: Image formation and thresholding

0.1.1 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

0.1.2 Load some modules

```
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
import pandas as pd
from skimage.morphology import ball
import tifffile as tiff
import plotsupport as ps

%matplotlib inline

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

0.1.3 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.

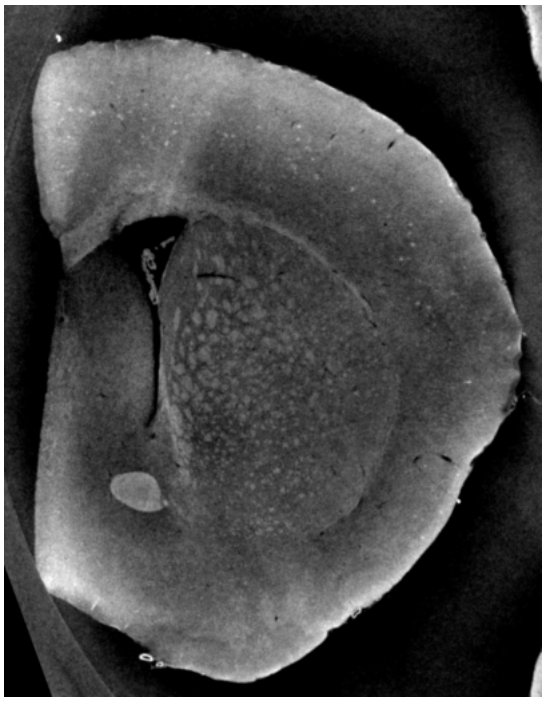


Fig. 1: An x-ray CT slice of the cortex.

- 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

0.1.4 Literature / Useful References

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

Models / ROC Curves

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

0.2 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.

0.2.1 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

0.2.2 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?

0.2.3 What we are looking at?

0.2.4 To test a hypothesis

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

We have performed an experiment that produced heaps of data to analyze. For example a using tomography.

At the beginning we have 2560 x 2560 x 2160 x 32 bits = 56GB / sample! Then we apply some filtering and preprocessing to prepare the data for analysis. After 20h of computer time we still have 56GB of data (it is however nicer to work with). This still way to much data to handle, we need to reduce it in some way.

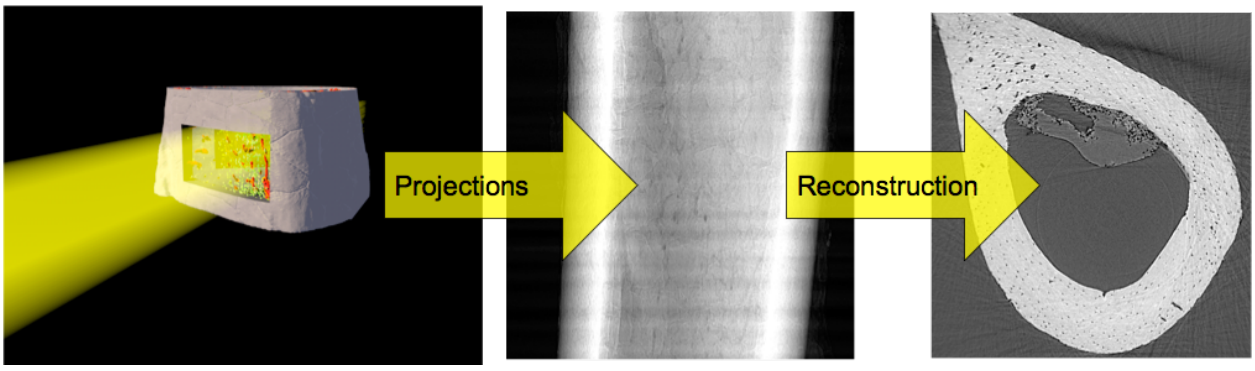


Fig. 1: Acquisition workflow to obtain CT slices of a specimen.

↓

Way too much data, we need to reduce

0.2.5 What did we want in the first place?

***Single numbers*:**

- Volume *fraction*,
- Cell *count*,
- Average cell *stretch*,
- Cell volume *variability*

  These are all **measurable** metrics!

0.2.6 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.

0.2.7 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

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

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)$$

0.2.8 What we get from the imaging modality

To demonstrate what we get from a modality, we load rubber duck radiograph as a toy example.

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

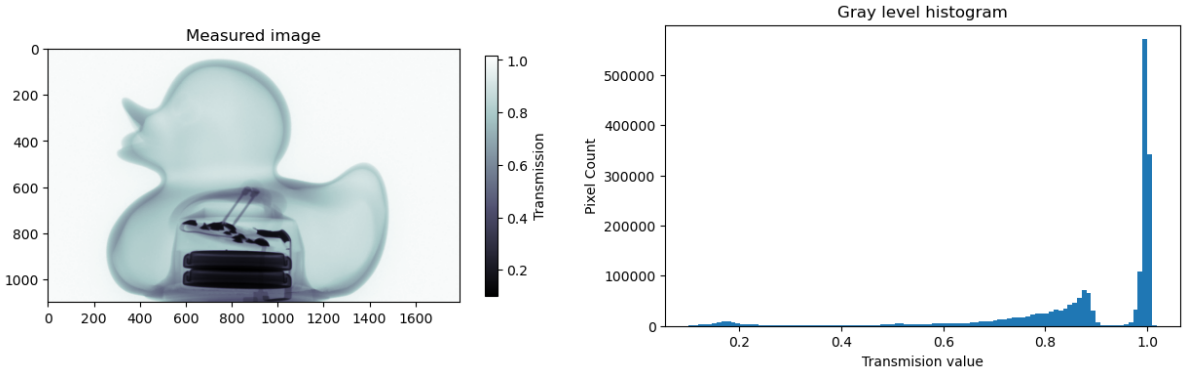
```
fig,ax=plt.subplots(1,figsize=(12,7))
dkimg = imread("figures/duck/normalized.tif")
ax.imshow(dkimg, cmap = 'bone');
ax.set_xticks([]); ax.set_yticks([]);
```

0.3 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 = tiff.imread("figures/duck/normalized.tif")
fig, (ax_img, ax_hist) = plt.subplots(1, 2, figsize = (15,4))

m_show_obj = ax_img.imshow(dkimg, cmap = 'bone')
cb_obj = fig.colorbar(m_show_obj,ax=ax_img,shrink=0.8)
cb_obj.set_label('Transmission'), ax_img.set_title('Measured image')

ax_hist.hist(dkimg.ravel(),bins=100)
ax_hist.set_xlabel('Transmision value')
ax_hist.set_ylabel('Pixel Count'), ax_hist.set_title('Gray level histogram');
```



0.3.1 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: “batteries in the bottom”, “solder spots on PCB”, “Thin skin”

0.3.2 Morphometrics

- Trace the outline of the object (or sub-structures)

0.4 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

0.4.1 How to approach the segmentation task

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.

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.

0.4.2 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.

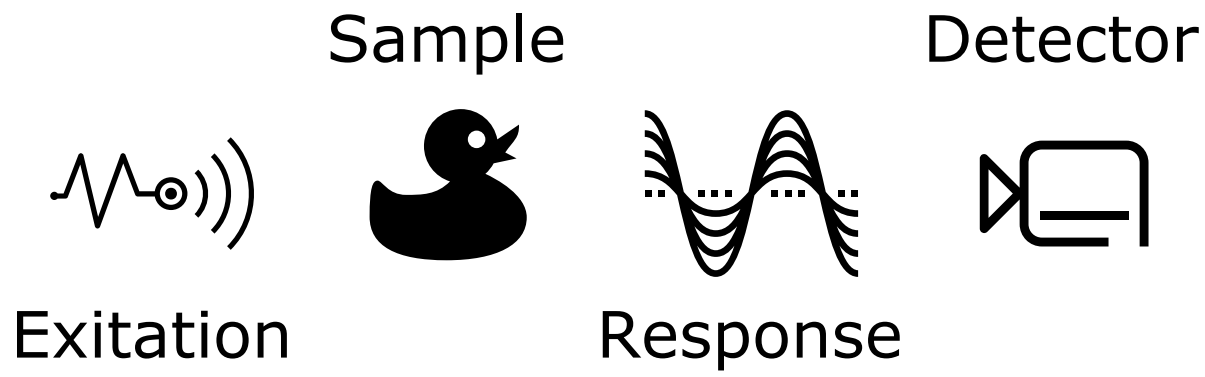


Fig. 1: The elements of the image formation process.

- **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:micro-CT to MRI to Confocal to Light-field to AFM.
- Similarities in underlying equations, but different *coefficients*, *units*, and *mechanism*

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

Direct Imaging (simple)

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

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

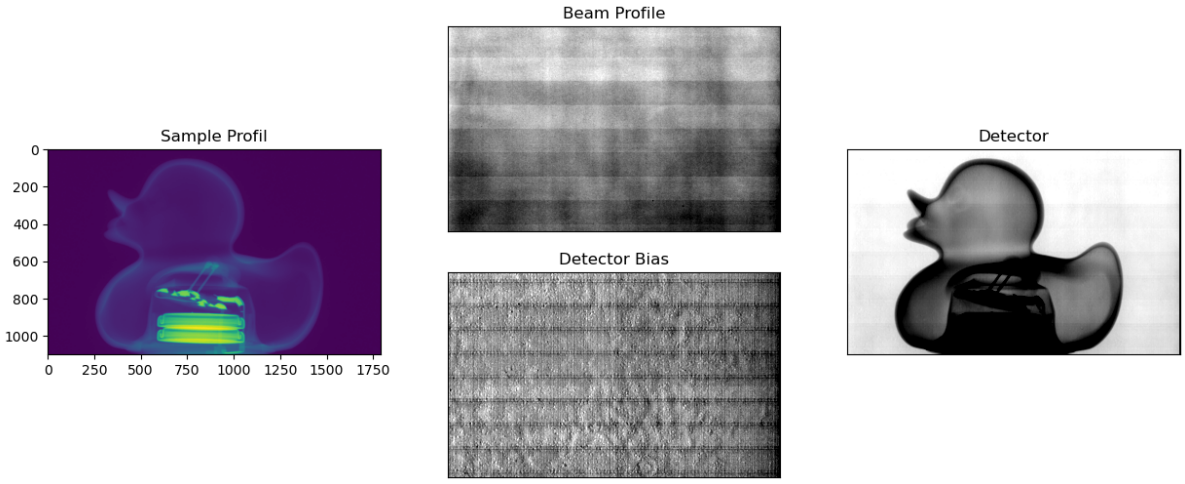
$$I_{stimulus} = \text{Beam}_{profile}$$

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

Let's look at a radiograph where beam profile that is penetrates 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
```

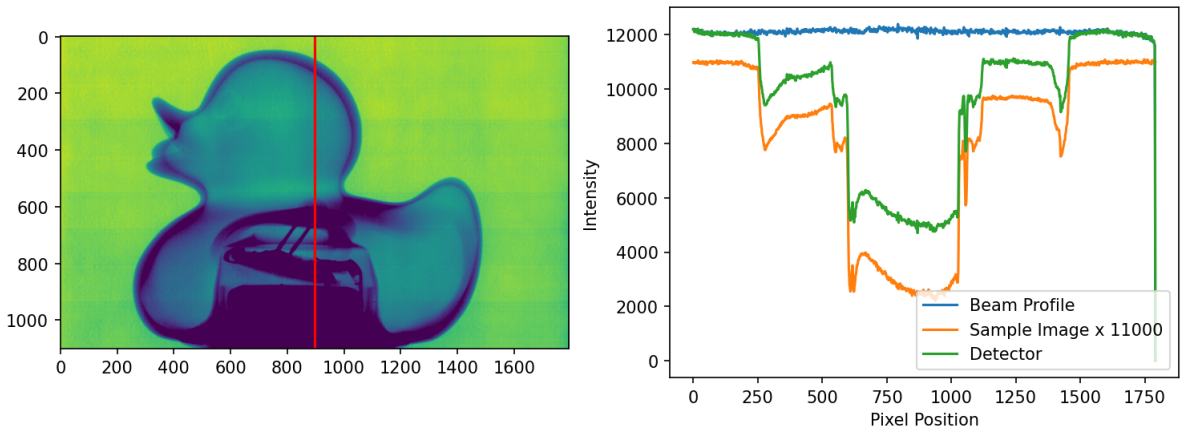
```
duck_img      = tiff.imread("figures/duck/neglognorm.tif")
duck_imgn     = tiff.imread("figures/duck/normalized.tif")
beam_img      = tiff.imread("figures/duck/ob.tif")
detector_bias = tiff.imread("figures/duck/dc.tif")
detector_img  = tiff.imread("figures/duck/duck90.tif")

fig =plt.figure(figsize=(15,6))
ax_beam = plt.subplot2grid(shape=(2,3 ), loc=(0, 1))
ax_img  = plt.subplot2grid(shape=(1,3),  loc=(0, 0))
ax_det  = plt.subplot2grid(shape=(1,3),  loc=(0, 2))
ax_bias = plt.subplot2grid(shape=(2,3),  loc=(1, 1))

m,s = (beam_img-detector_bias).mean(), (beam_img-detector_bias).std()
ax_beam.imshow(beam_img-detector_bias, clim=[m-s,m+s], cmap = 'gray'); ax_beam.set_
 ↪title('Beam Profile')
ax_beam.set_xticks([]), ax_beam.set_yticks([]);
m,s = detector_bias.mean(), detector_bias.std()
ax_bias.imshow(detector_bias, clim=[m-s,m+s], cmap = 'gray'); ax_bias.set_title(
 ↪'Detector Bias')
ax_bias.set_xticks([]), ax_bias.set_yticks([]);
ax_img.imshow(duck_img,    cmap = 'viridis'); ax_img.set_title('Sample Profil')
ax_beam.set_xticks([]), ax_beam.set_yticks([]);
m,s = detector_img.mean(), detector_img.std()
ax_det.imshow(detector_img, clim=[m-0.7*s,m+0.7*s],cmap = 'gray'); ax_det.set_title(
 ↪'Detector');
ax_det.set_xticks([]), ax_det.set_yticks([]);
```



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)
m,s=detector_img.mean(),detector_img.std()
ax[0].imshow(detector_img,clim=[m-s,m+s]);
ax[0].axvline(beam_img.shape[1]//2,color='red')
ax[1].plot(beam_img[beam_img.shape[1]//2], label = 'Beam Profile')
ax[1].plot(11000*duck_imgn[beam_img.shape[1]//2], label = 'Sample Image x 11000')
ax[1].plot(detector_img[detector_img.shape[1]//2], label = 'Detector')
ax[1].set_ylabel('Intensity'); ax[1].set_xlabel('Pixel Position');ax[1].legend(loc=
↪"lower right");
```



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.

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
import numpy as np
import pandas as pd
```

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, 0.50, 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
66   0.617795  material 1    0.539132
67   1.531738  material 1    0.216160
231  3.264320  material 3    0.038223
```

(continues on next page)

(continued from previous page)

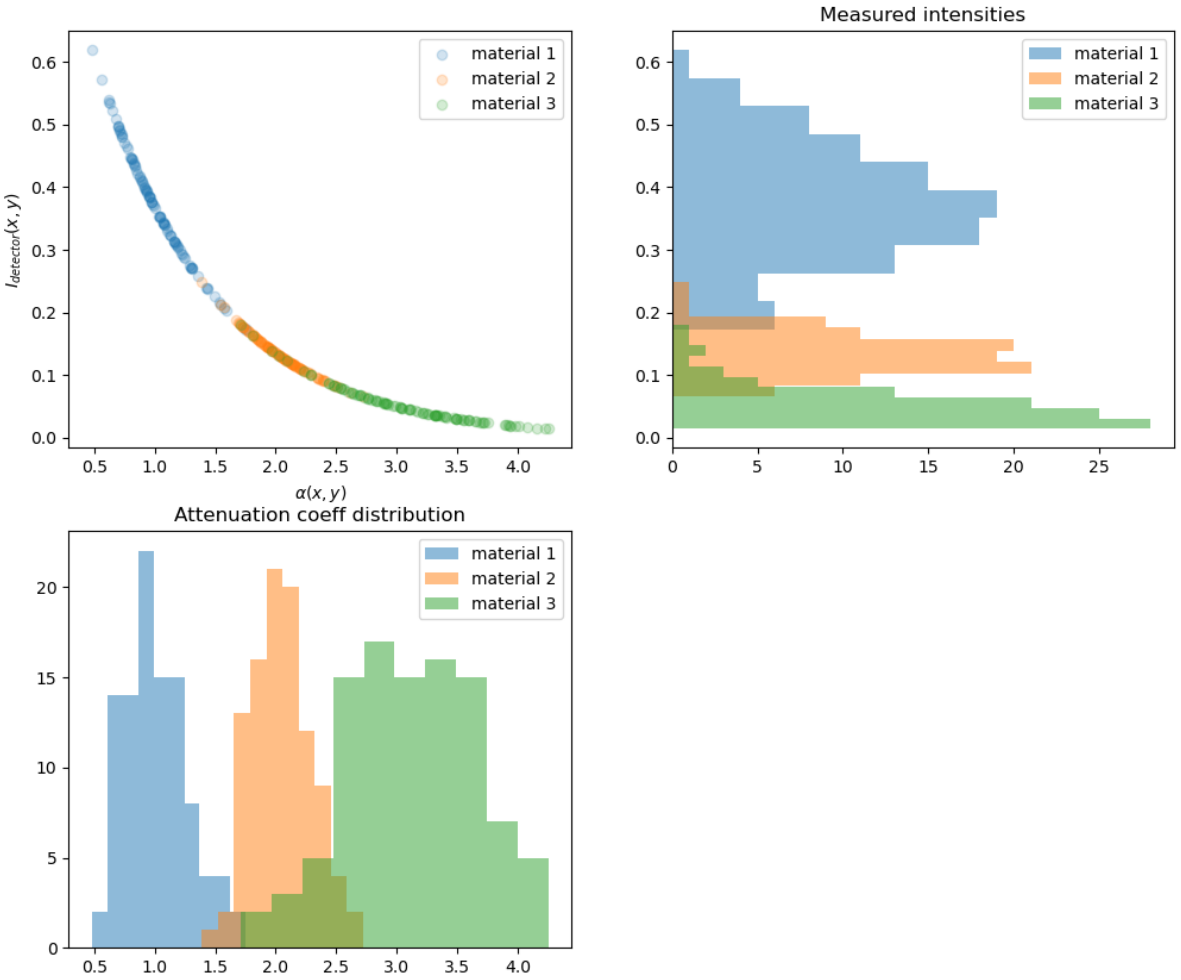
```
226  3.150598  material 3    0.042826
205  3.098720  material 3    0.045107
```

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 = (12, 10))
for c_mat, c_df in abs_df.groupby('material'):
    ax1.scatter(x = c_df['alpha'],
                y = c_df['I_detector'],
                label = c_mat,alpha=0.2)
    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('$\\alpha(x,y)$');
ax1.set_ylabel('$I_{detector}(x,y)$')
ax1.legend();
ax2.legend();
ax2.set_title('Measured intensities')
ax3.legend(loc = 0);
ax3.set_title('Attenuation coeff distribution')
ax4.axis('off');
```



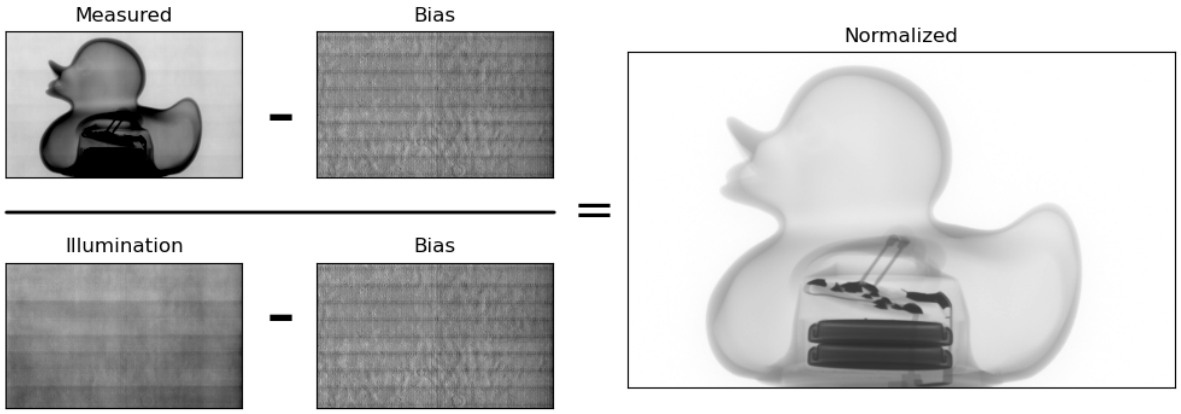
The material are differently represented!

0.4.3 Flatten a transmission image

- A transmission image can be described by Beer-Lambert's law
- Each image has a bias introduced by the detector

$$T = \frac{I_{Measured} - I_{Bias}}{I_{Illumination} - I_{Bias}} = e^{-\int \alpha(x)dx}$$

```
ps.visualize_normalization(detector_img,beam_img,detector_bias,duck_imgn)
```



T is an image normalized between 0 and 1

$$\begin{cases} T = 1 & \text{No sample between source and detector} \\ 0 < T < 1 & \text{A sample attenuates the beam to some degree} \\ T = 0 & \text{The sample is opaque} \end{cases}$$

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.

0.5 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.

0.5.1 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

0.5.2 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)
```

Transmission image of the breast phantom

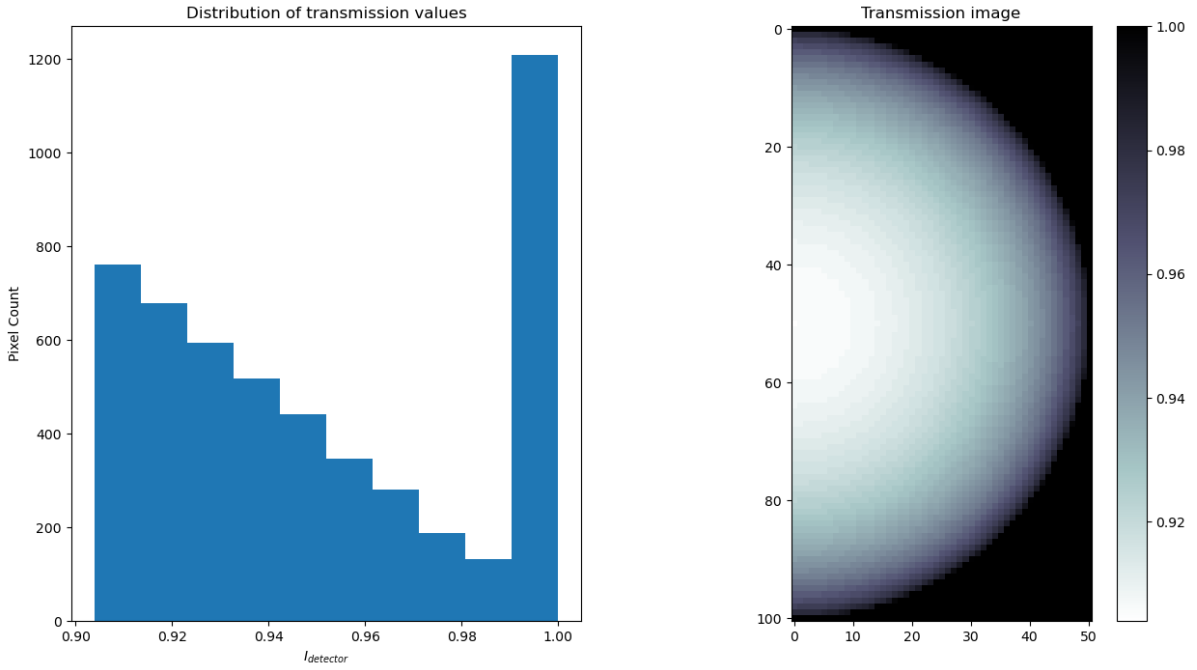
Our first step is to simulate a transmission image of the breast. This is done by

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,axis=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.

Compute the thickness

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

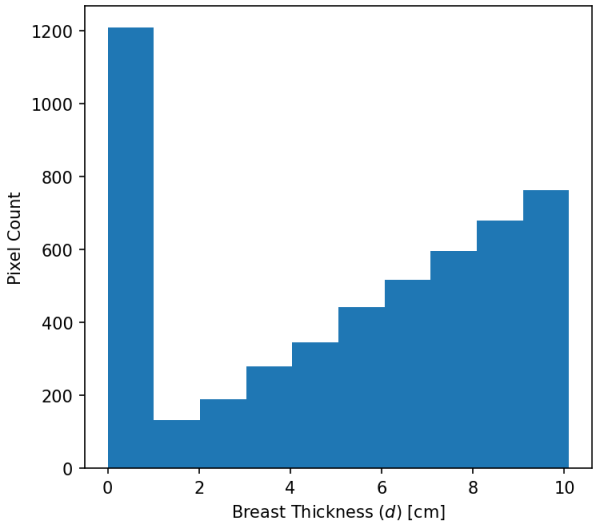
$$d = -\log(I_{detector})/\alpha$$

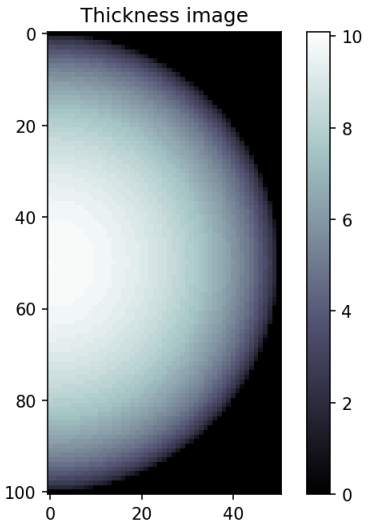
This is only valid because we have air ($\alpha = 0$) as the second component in the phantom. Otherwise, if it was a denser material we would have a material mixture.

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

```
breast_thickness = -np.log(i_detector)/breast_alpha # Compute the thickness
```

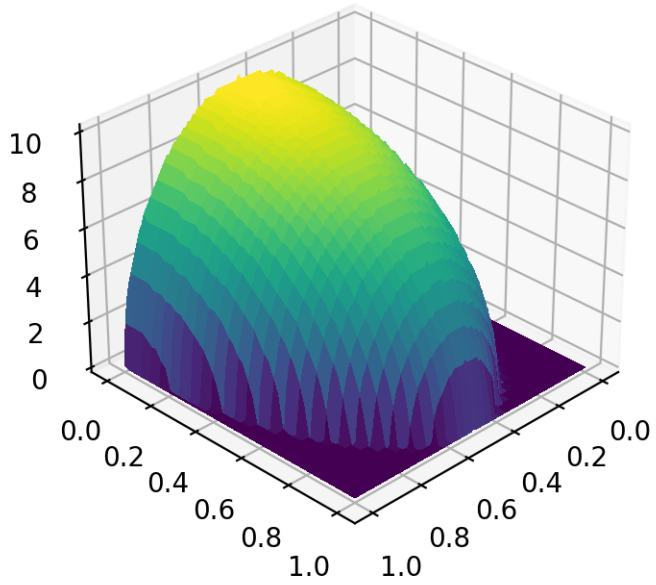
```
fig, (ax_hist, ax_breast) = plt.subplots(1, 2, figsize = (12,5), dpi=150)

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');
```





Visualizing the thickness

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize = (8, 4), dpi = 200)
# ax  = fig.gca(projection='3d')
ax = fig.add_subplot(1, 1, 1, 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');
```



0.5.3 What if alpha is not constant?

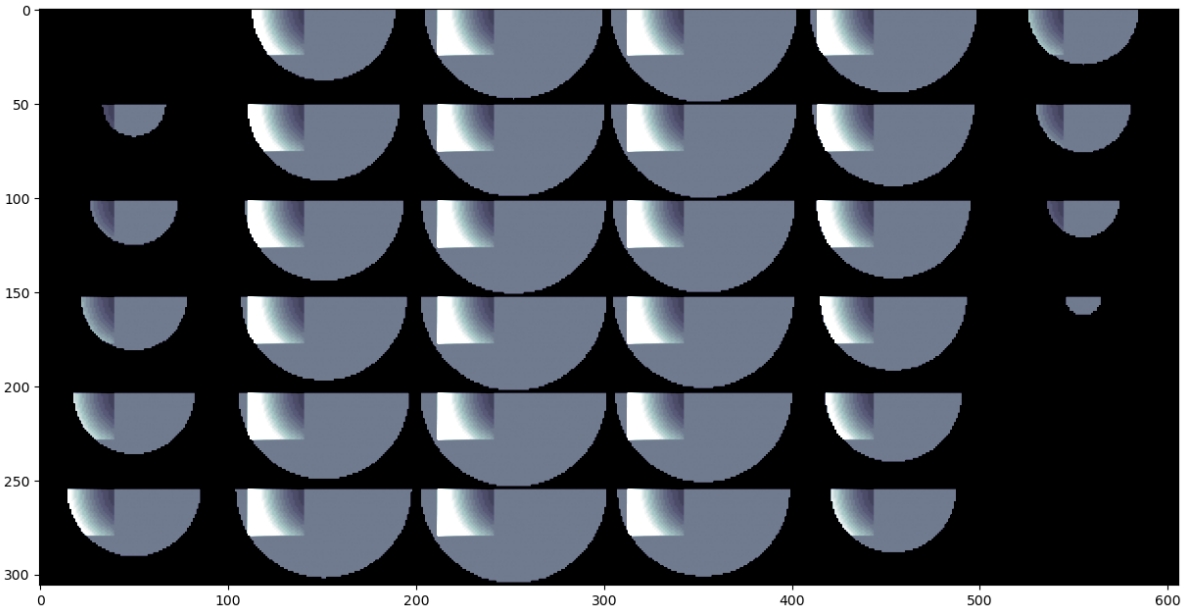
We run into problems when the α is no longer constant.

- 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 = (15, 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);
```

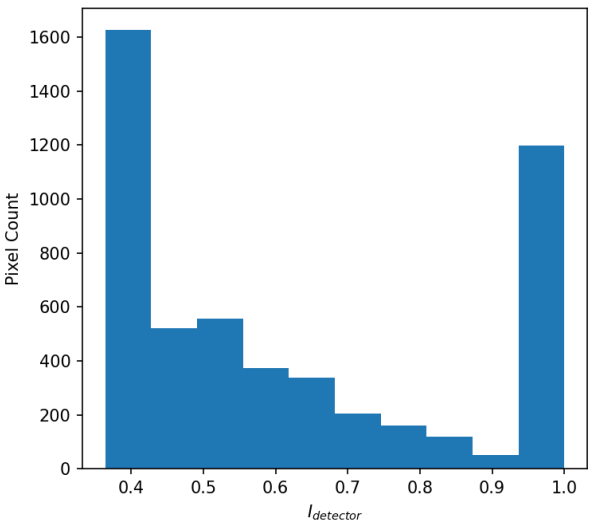


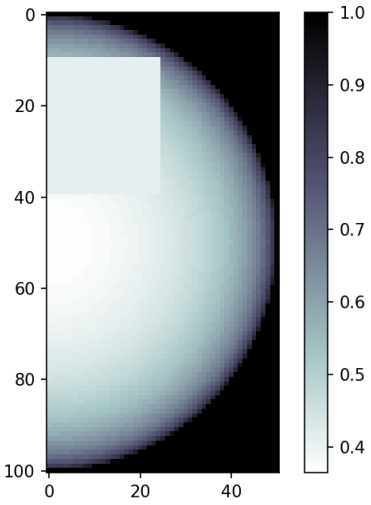
Looking at the thickness again

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)) # Compute what the detector sees
```

```
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');
```





An anomaly in the thickness reconstruction

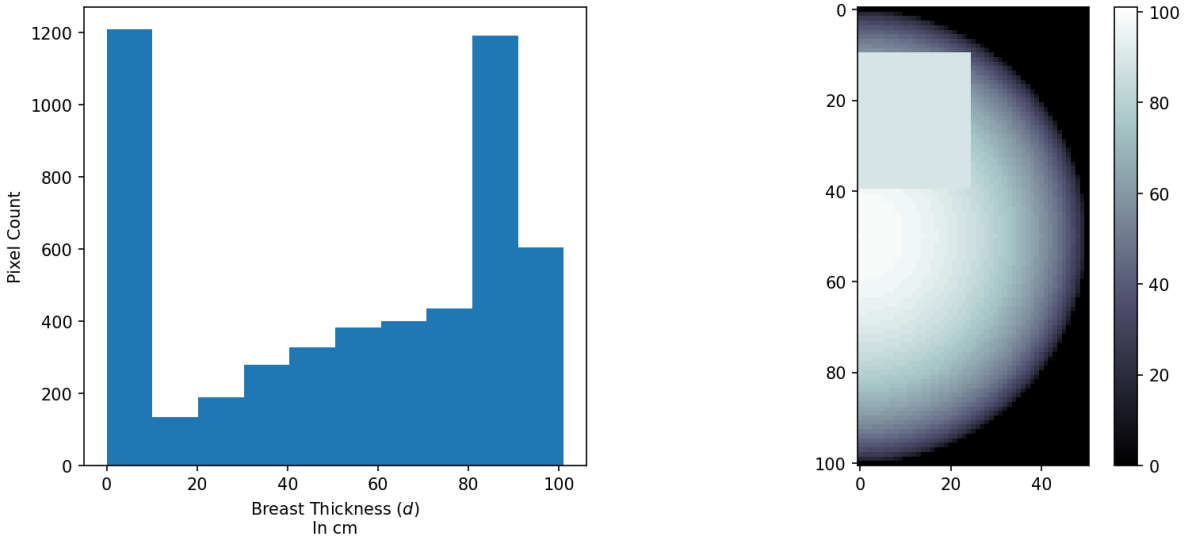
It appears as a flat constant region in the thickness reconstruction.

So we fundamentally from this single image cannot answer:

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

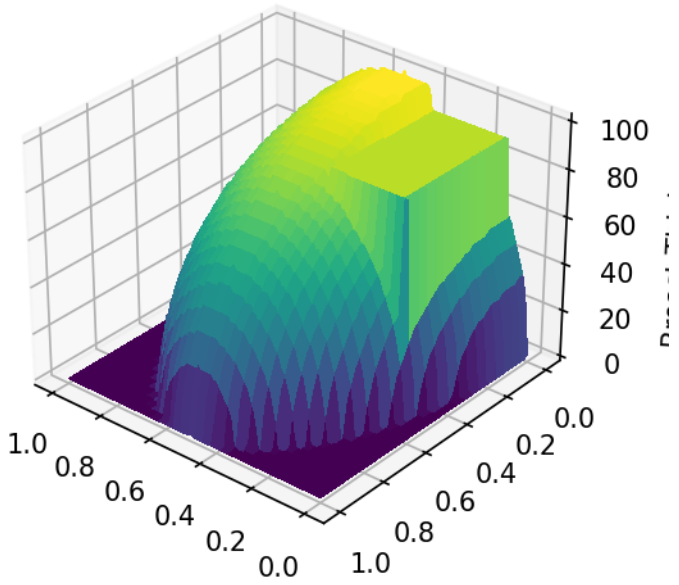
```
breast_thickness = -np.log(i_detector)/1e-2
```

```
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');
```



Looking at the thickness profile with lump

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize = (8, 4), dpi = 150)
ax  = fig.add_subplot(1,1,1,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');
```



0.5.4 Summary of the image formation process

- Images from the detector can mostly *not* be used directly.
- Normalization is needed.
- The information in transmission images can be difficult to interpret.

0.6 Segmentation

0.6.1 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 → 2560 x 2560 x 2160 x **1 bit** = 1.75GB / sample

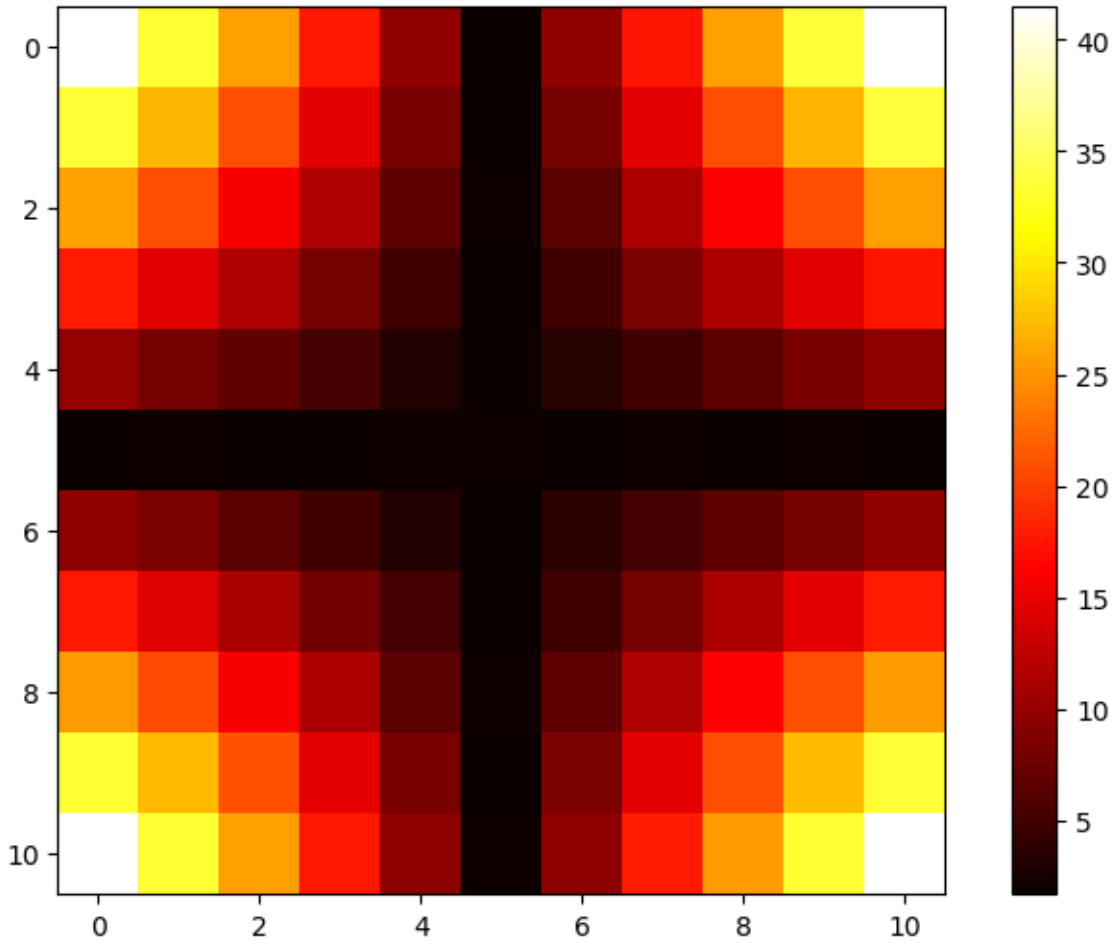
0.6.2 Basic segmentation: Applying a threshold to an image

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.

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

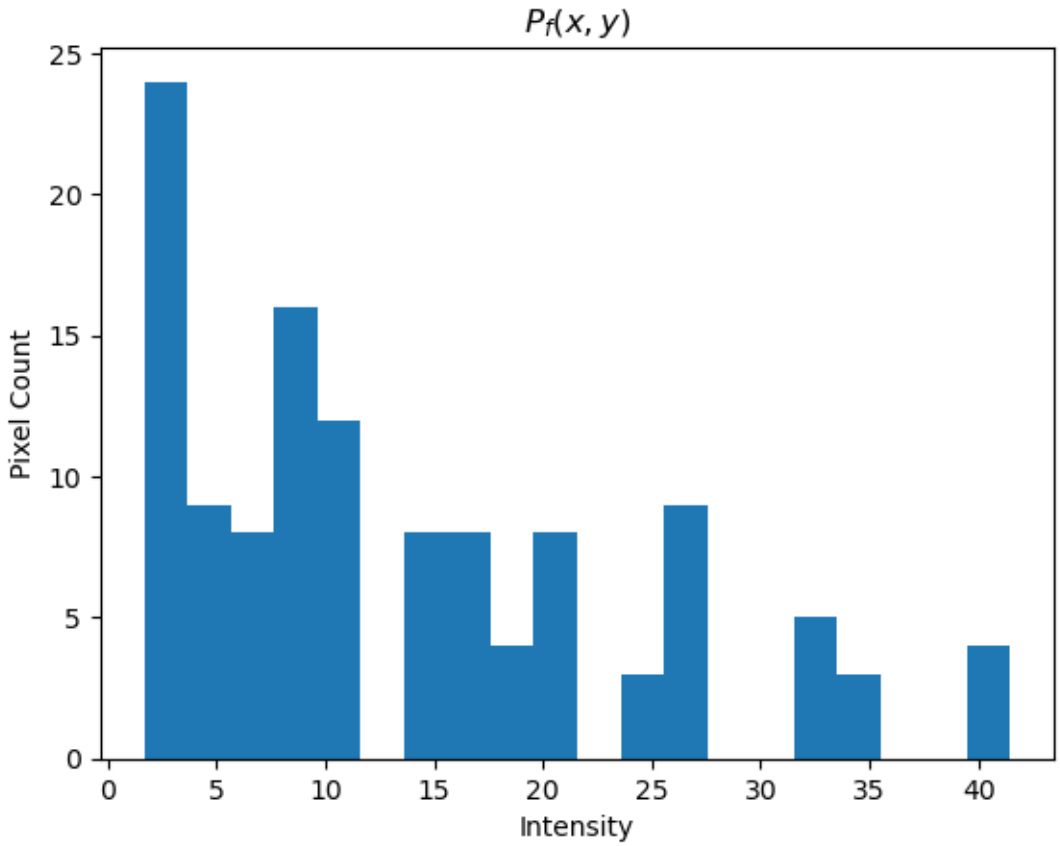
```
fig,ax = plt.subplots(1,1,figsize=(9,6))
im=ax.imshow(cross_im, cmap = 'hot')
fig.colorbar(im);
```



0.6.3 The histogram

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

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



0.6.4 Applying a threshold to an image

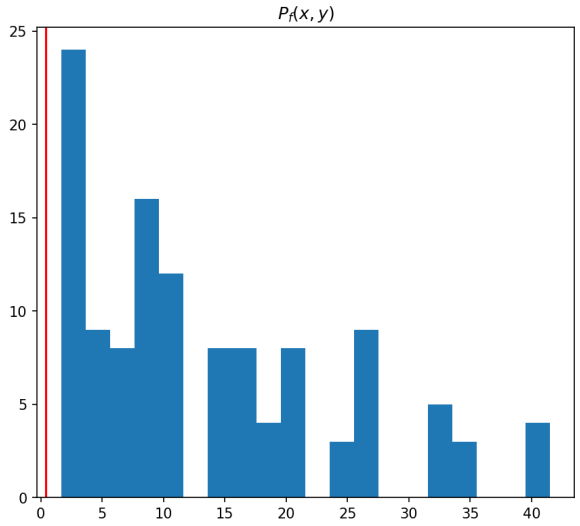
By examining the image and probability distribution function, we can *deduce* that the underyling model is a whitish phase that makes up the cross and the darkish background

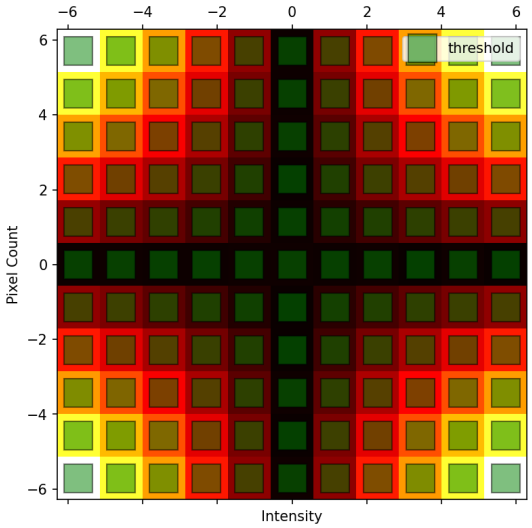
Applying the threshold is a deceptively simple operation

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

```
threshold  = 0.4
thresh_img = cross_im > threshold
```

```
fig, (ax2,ax1) = plt.subplots(1,2,figsize=(15,6),dpi=150)
ax1.matshow(cross_im, cmap = 'hot', extent = [xx.min(), xx.max(), yy.min(), yy.max()])

ax1.plot(xx[np.where(thresh_img)]*0.91, yy[np.where(thresh_img)]*0.91,
         'ks', markerfacecolor = 'green', alpha = 0.5, label = 'threshold',␣
 ↪markersize = 20)
ax1.legend();
ax2.hist(cross_im.ravel(), 20)
ax2.set_title('$P_f(x,y)$'); ax1.set_xlabel('Intensity'); ax1.set_ylabel('Pixel Count
 ↪');

ax2.axvline(x=0.4, color='r');
```



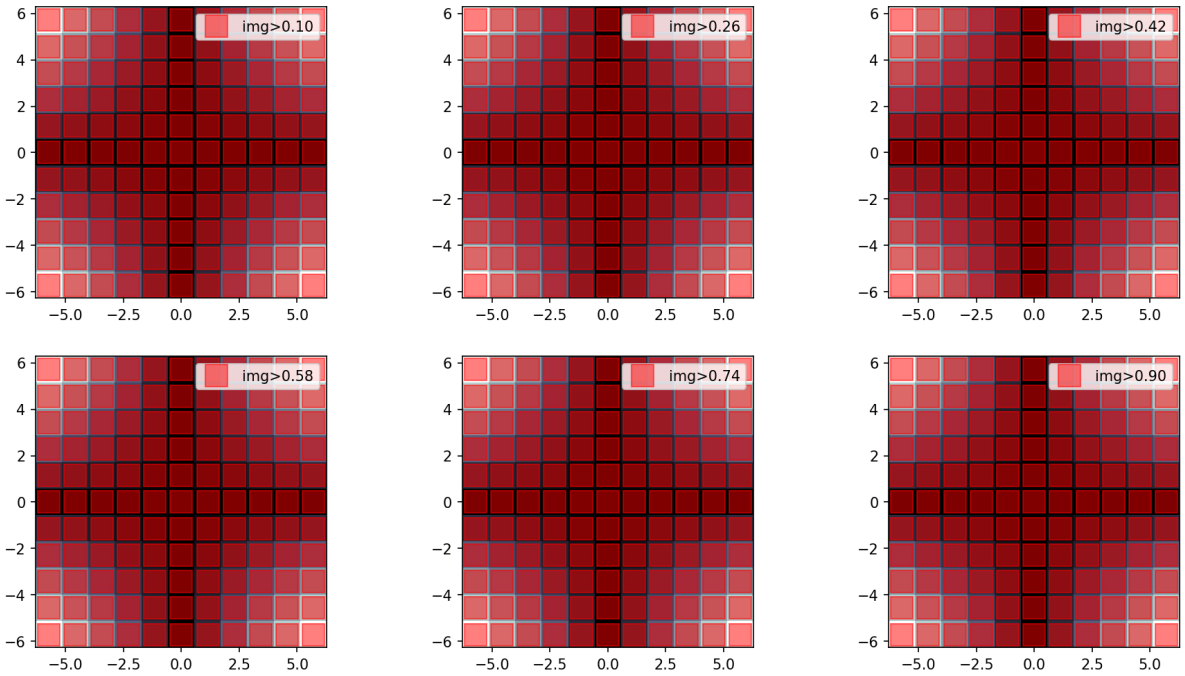


Various Thresholds

We can see the effect of choosing various thresholds

$\gamma \in \{0.1, 0.26, 0.42, 0.58, 0.74, 0.9\}$

```
fig, m_axs = plt.subplots(2,3,
                          figsize = (15, 8),dpi=150)
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)]*0.91, yy[np.where(thresh_img)]*0.91, 'rs',␣
 ↪alpha = 0.5, label = 'img>%2.2f' % c_thresh, markersize = 15)
    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.

0.7 Segmenting Cells

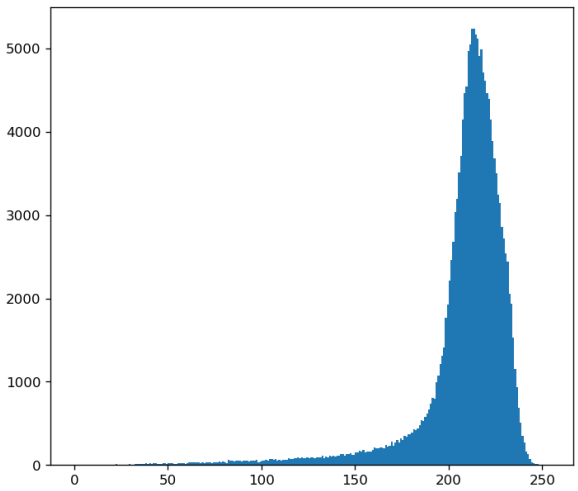
We can peform the same sort of analysis with this image of cells

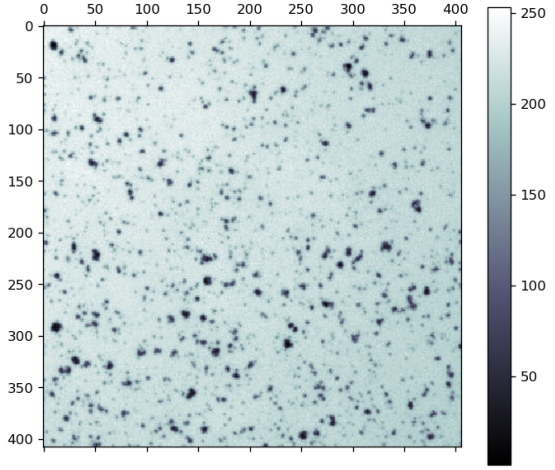
This time we can derive the model from the basic physics of the system

- 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
```

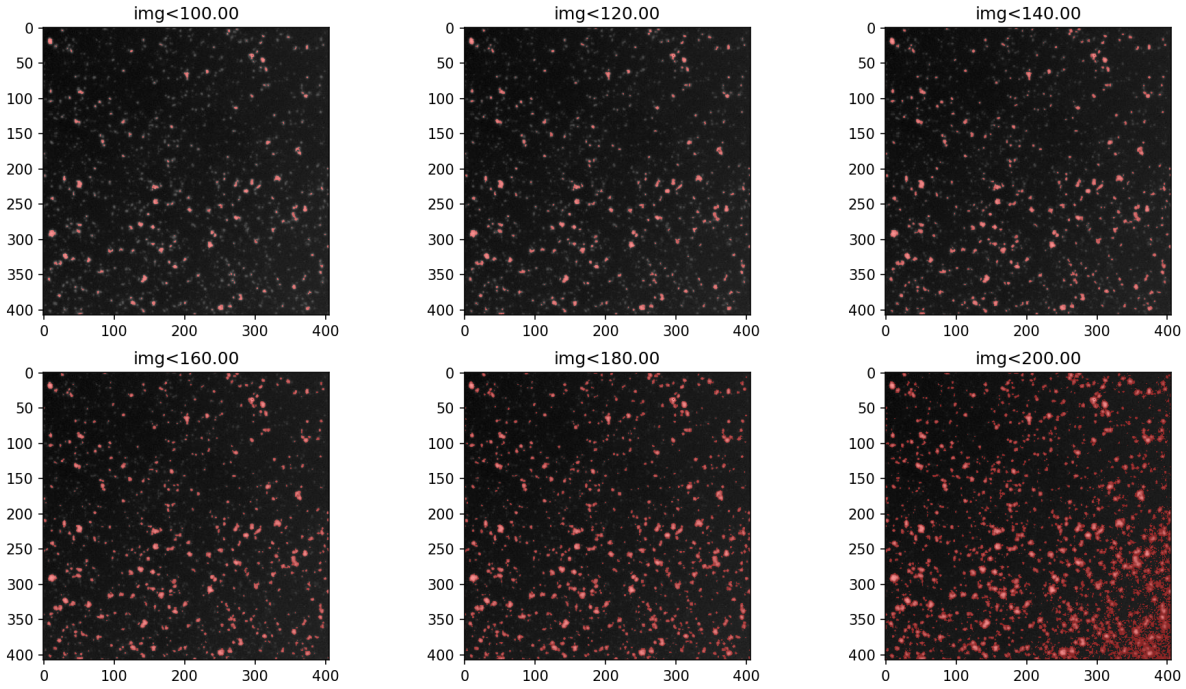
```
cell_img = imread("figures/Cell_Colony.jpg")

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);
```





0.7.1 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),
 ↪interpolation='None') # Rgb coding of image and mask
    ax1.set_title('img<%2.2f' % c_thresh)
```



There is a graylevel gradient in the image!

0.8 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.

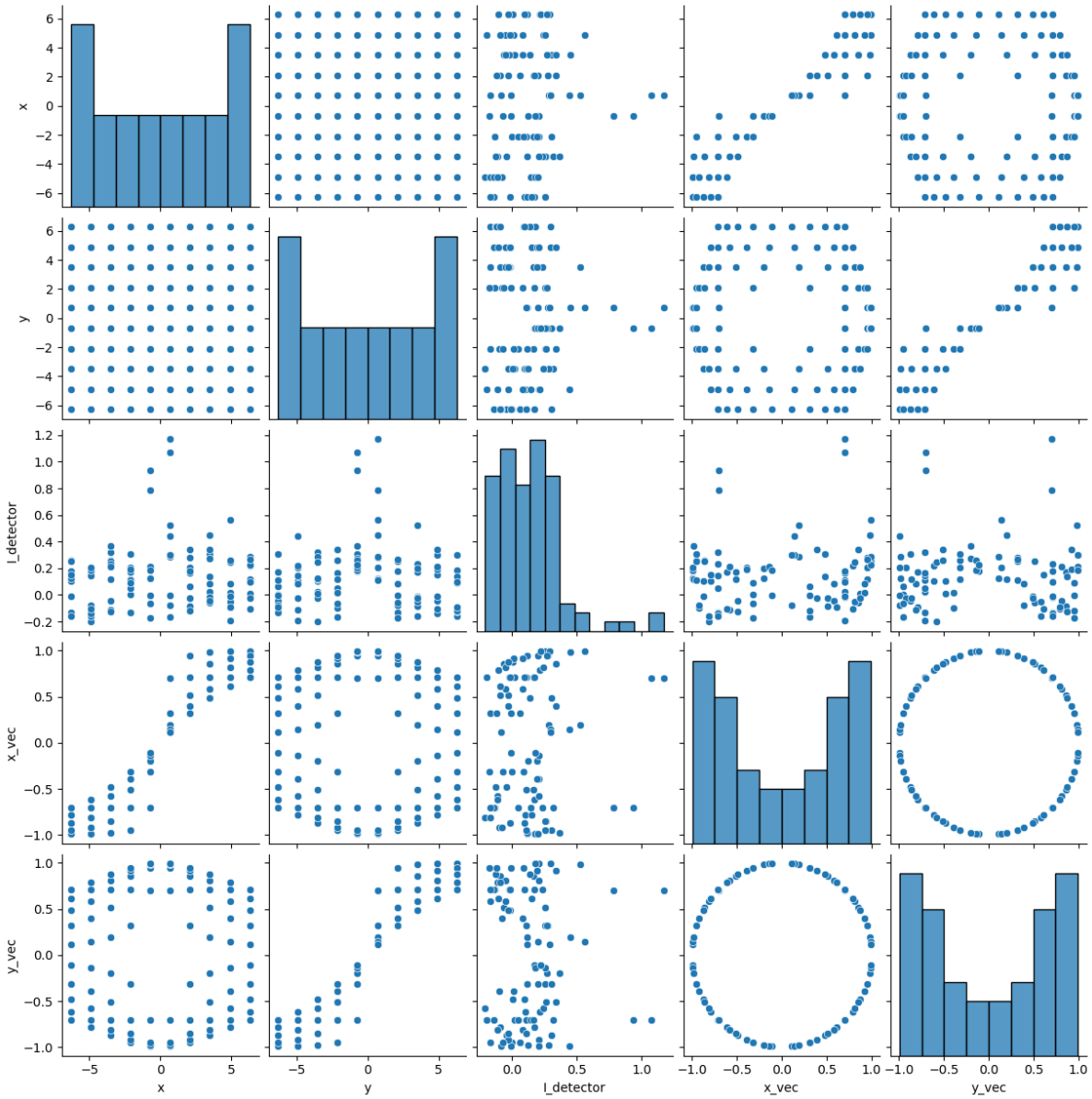
```
nx = 10
ny = 10
xx, yy = np.meshgrid(np.linspace(-2*np.pi, 2*np.pi, nx),
                     np.linspace(-2*np.pi, 2*np.pi, ny))

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
11	-4.886922	-4.886922	0.145787	-0.707033	-0.707033
96	2.094395	6.283185	-0.116856	0.316192	0.948575
17	3.490659	-4.886922	0.083704	0.581158	-0.813621
39	6.283185	-2.094395	0.115649	0.948575	-0.316192
46	2.094395	-0.698132	0.201720	0.947712	-0.315904

0.8.1 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.

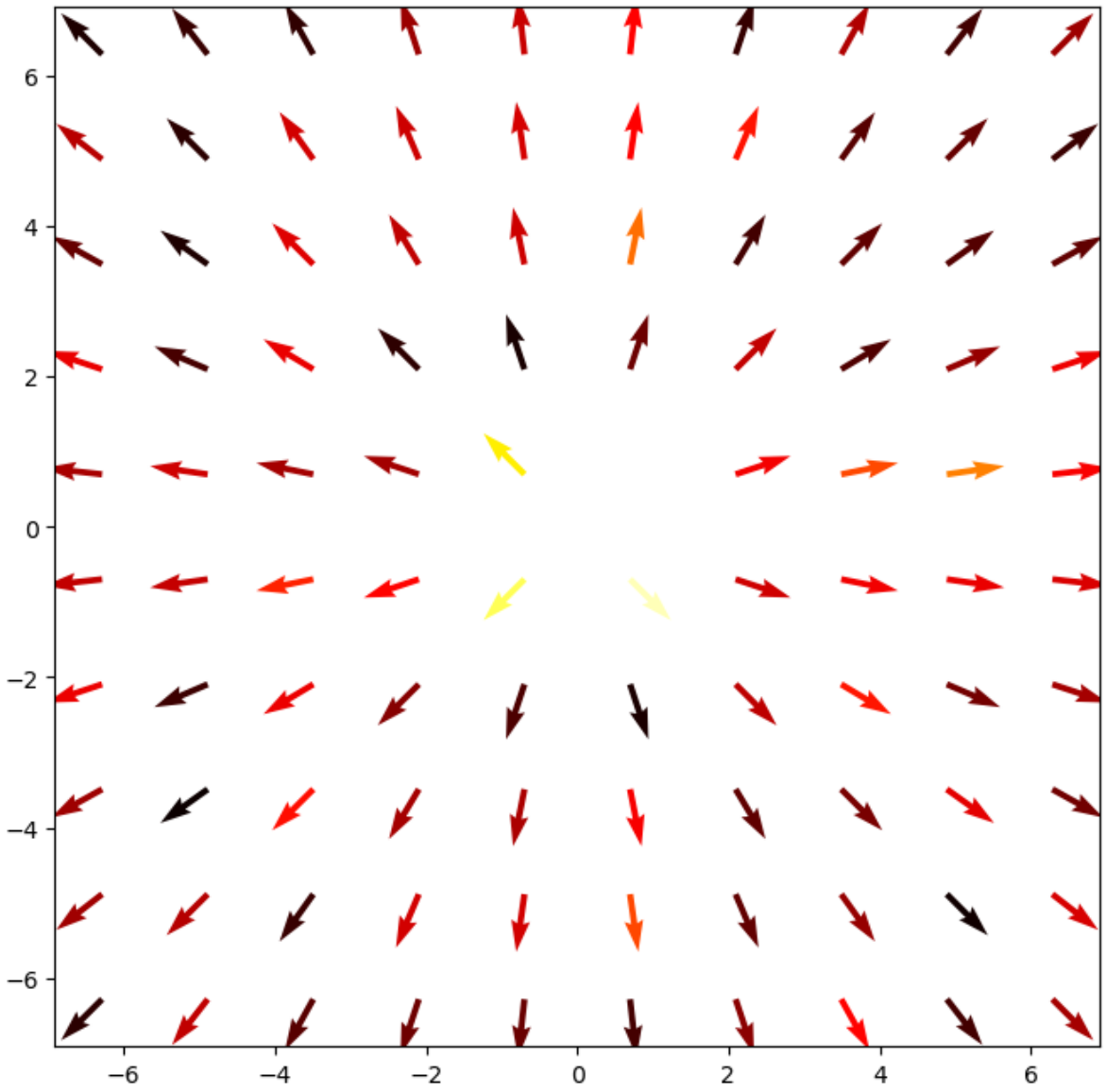
```
import seaborn as sns
sns.pairplot(base_df);
```



0.8.2 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.

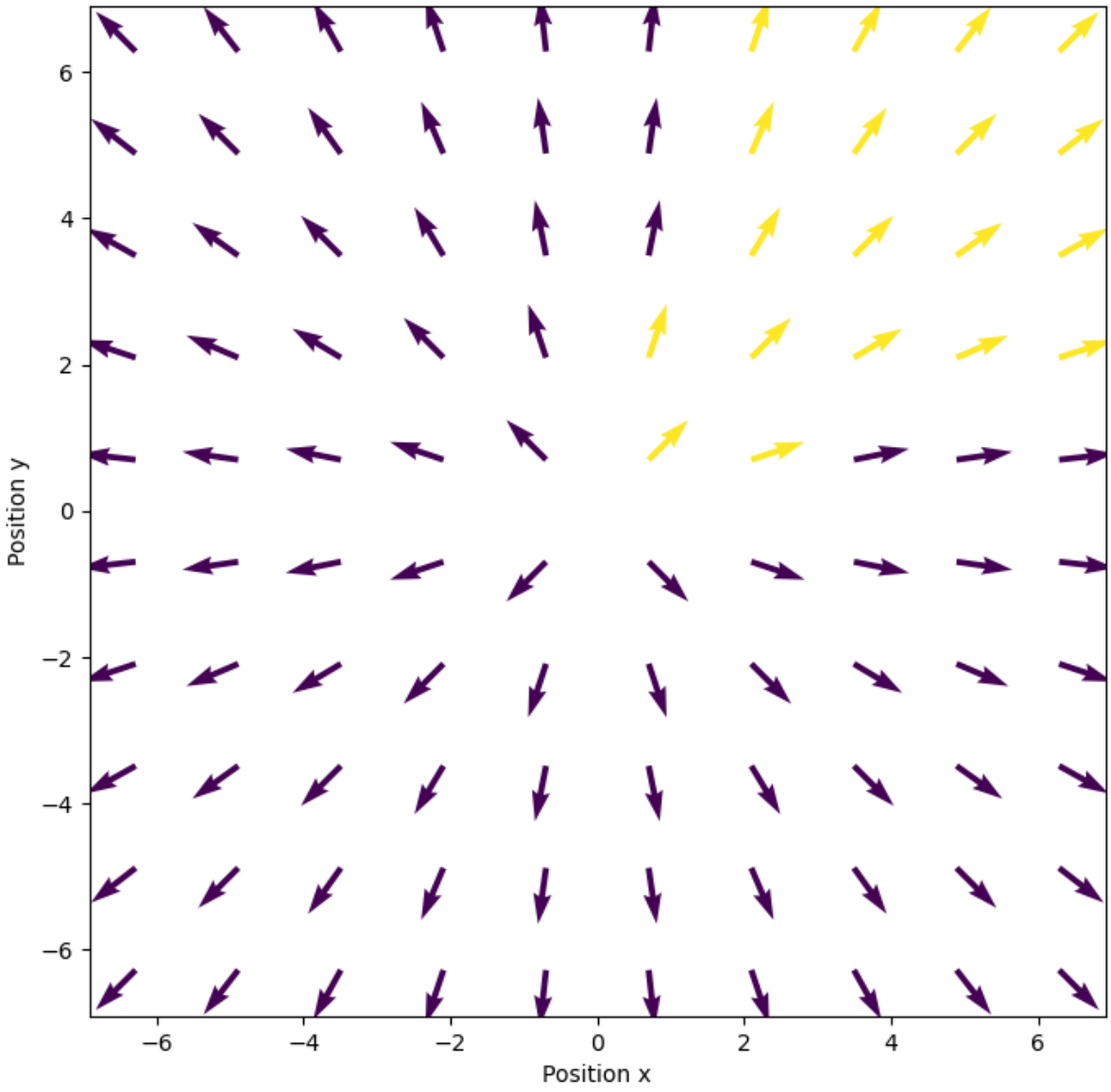
```
fig, ax1 = plt.subplots(1,1, figsize = (8, 8))
ax1.quiver(base_df['x'], base_df['y'], base_df['x_vec'], base_df['y_vec'], base_df['I_
↪detector'], cmap = 'hot');
```



0.8.3 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');
```

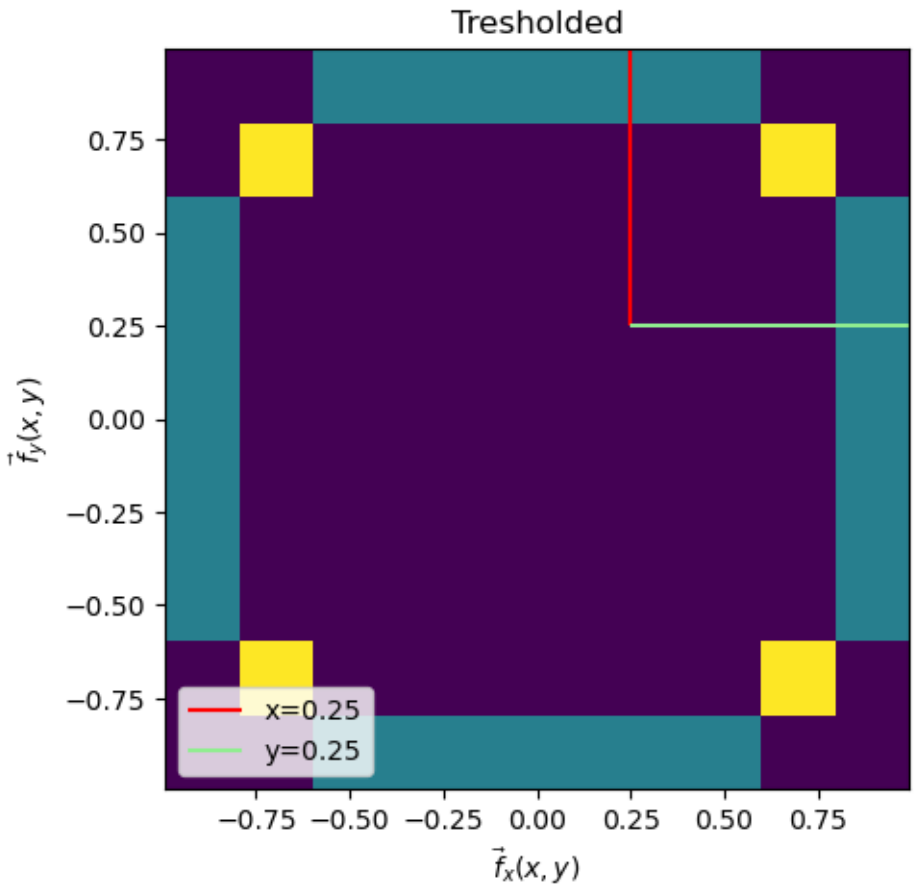


Histogram of the vectors

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))
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');
```



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.

$$I(x,y) = \begin{cases} 1, & |\vec{f}(x,y)\cdot\vec{i}| = 1 \\ 0, & \text{otherwise} \end{cases}$$

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

$$\vec{x}\cdot\vec{y} = |\vec{x}||\vec{y}|\cos(\theta_{x\to y})$$

$$I(x,y) = \begin{cases} 1, & \cos^{-1}(\vec{f}(x,y)\cdot\vec{i}) \leq \theta^\circ \\ 0, & \text{otherwise} \end{cases}$$

0.8.4 Summary of basic thresholding

- Thresholding is the first approach to segmentation
- The histogram guides the threshold selection
- Inhomogeneous illumination and noise make the task harder.

0.9 A Machine Learning Approach to Image Processing

0.9.1 Loading some modules

Let's load a collection of modules for this part.

```
import numpy as np
import matplotlib.pyplot as plt
from skimage.color import rgb2gray
from skimage.io import imread
import plotsupport as ps
from sklearn.metrics import roc_auc_score
import pandas as pd
from collections import OrderedDict
from sklearn.metrics import roc_curve

%matplotlib inline
```

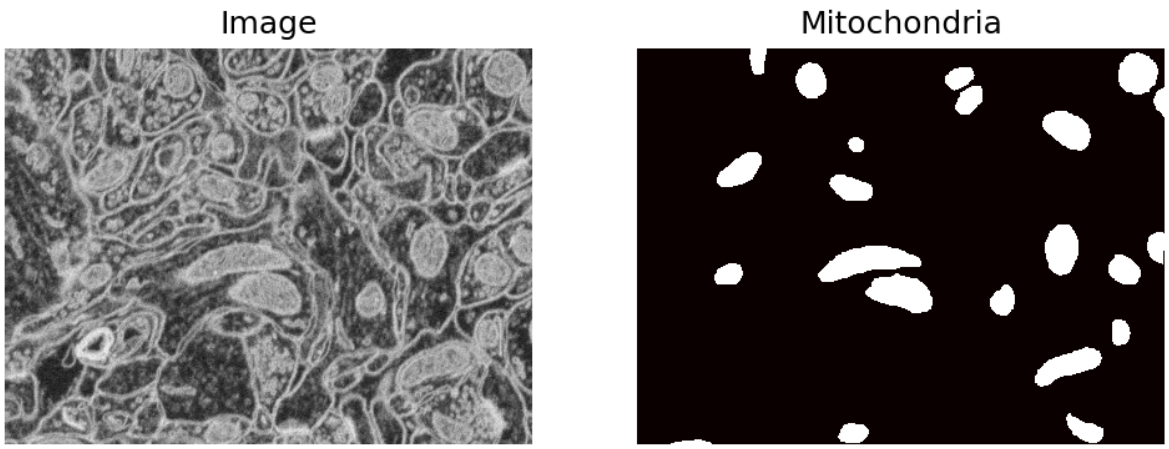
0.9.2 How to approach the analysis

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…

Identifying the mitochondria in the images like the one to the left in the figures below.

```
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',interpolation='None');  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*

This is a really tedious task and we want to automatize it. Here, segmentation is a good approach. The question is now how we can achieve this.

0.9.3 Which values can be assigned

- **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*

But we only know if it is right or wrong if we have a ground truth (lecture 2).

Appying a threshold

We can then apply a threshold to the image to determine the number of points in each category

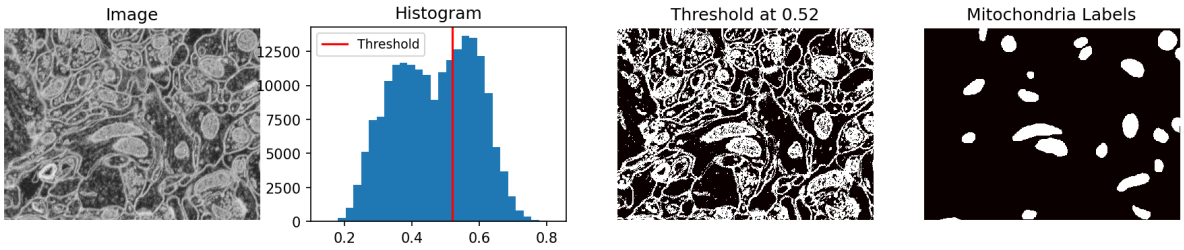
```
thresh     = 0.52
thresh_img = cell_img > thresh # Apply a single threshold to the image
```

```
# Visualization
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=30); ax[1].set_title('Histogram')
ax[1].axvline(thresh,color='r',label='Threshold');
ax[1].legend(fontsize=9)

ax[2].imshow(thresh_img, cmap='hot',interpolation='none');
```

(continues on next page)

(continued from previous page)

```
ax[2].set_title('Threshold at {0}'.format(thresh));
ax[2].axis('off')

ax[3].imshow(cell_seg,   cmap='hot',interpolation='none');
ax[3].set_title('Mitochondria Labels');
ax[3].axis('off');
```



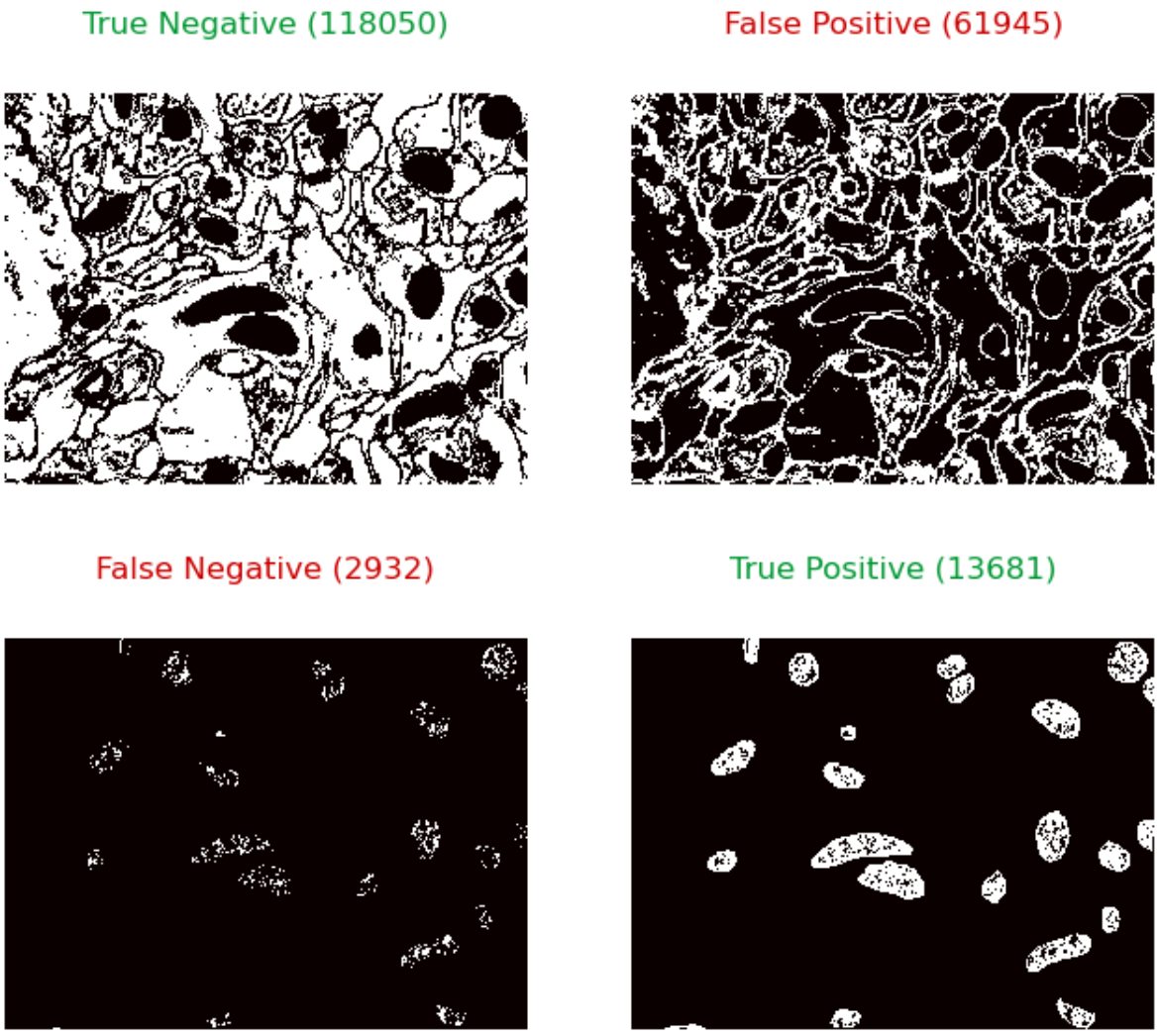
In this example we can see that it is clearly not sufficient to apply a single threshold as we have tried before. When we compare the thresholded image to the provided mask, we can see the that there are plenty more structures marked as foreground and also that there are holes within the mitochondria.

0.9.4 Check the performance of the thresholding

Let's create a confusion matrix to visualize the performance of the segmentation. A first step is to compute how many hits and misses our segmentation resulted in. In particual, looking at the four different cases that can occur in a binarization.

```
# 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','#009933'
    if real_img_idx == 0 and pred_img_idx == 0:
        return 'True Negative','#009933'
    if real_img_idx == 0 and pred_img_idx == 1:
        return 'False Positive','#cc0000'
    if real_img_idx == 1 and pred_img_idx == 0:
        return 'False Negative','#cc0000'
```

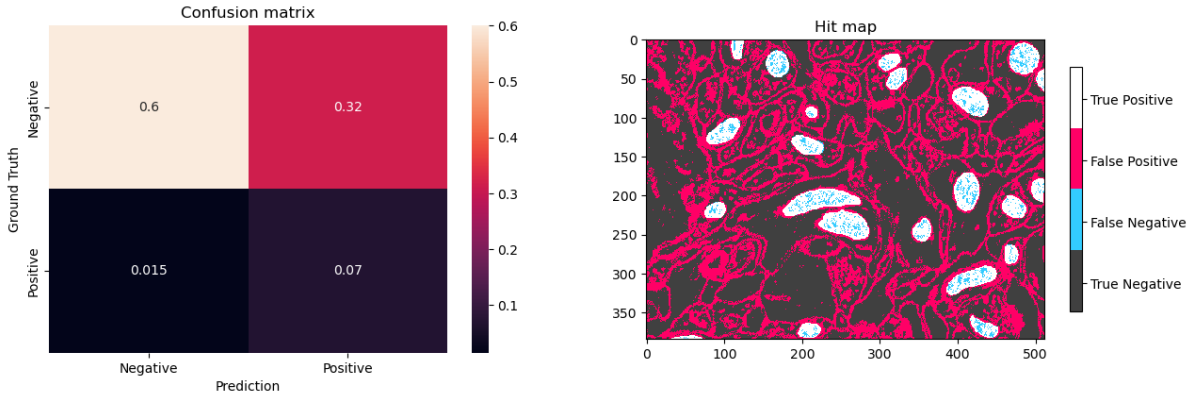
```
out_results = {}
fig, m_ax = plt.subplots(2, 2, figsize=(8, 7), dpi=100)
for real_img_idx, n_ax in zip([0, 1], m_ax):
    for pred_img_idx, c_ax in zip([0, 1], n_ax):
        match_img = (thresh_img == pred_img_idx) & (cell_seg == real_img_idx)
        (tp_title,color) = tp_func(real_img_idx, pred_img_idx)
        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')
```



0.9.5 The confusion matrix (revisited)

From the counts in the previus slide, we can now create a Confusion matrix and also look at the combined image of all the cases. In the hit map we can see white and gray as true segmentation and blue and magenta as false segmentations.

```
fig,ax=plt.subplots(1,2,figsize=(15,4.5))
ps.showHitMap(cell_seg,thresh_img,ax=ax) # this is a handy support function provided␣
↪with the notebook
```



0.9.6 Apply Precision and Recall

We can use two further metrics to measure the performance of a segmentation method. These are based on the information in the confusion matrix like

Recall (sensitivity) $\$\frac{TP}{TP+FN}\$$

Precision $\$\frac{TP}{TP+FP}\$$

Recall is the sum the true positive relative to the number of positives in the mask or also as written here the sum of true positives and false negatives. Recall tells us how good the method is to find the correct label within the mask.

Precision is the sum of true positives relative to the total number of positives provided by our segmentation method. The precision tells us how much our method over segments the image.

Both recall and precision are scalar numbers in the interval $0 < m \leq 1$ where '1' is the ideal condition.

Let's compute precision and recall for our mitochonria example.

```
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
```

This result tells us that our segmentation was relatively good at finding the mitochondria, but also that this happened at the cost of many false positives.

0.10 Reciever Operating Characteristic (ROC)

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.

- The concept of the ROC curve was first developed for WW2 soldiers detecting objects in battlefields using radar.
- As we saw before, for a single threshold value 0.5, we were able to compute a single recall and precision.
- The ROC shows the releation between recall and precision for a segmentation model.

Let's compute the hit and miss statistics…

If we want to make an ROC curve we take a number of threshold values and compute the corresponding precision and recall values for each threshold. In the example below we scan threshold values from 0.1 to 0.9 and compute the hit and miss statistics to calculate the precision and recall.

```
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, #009933)  \
0           61945            2932          13681                         0
1           61945            2932          13681                         0
2           61945            2932          13681                         0

   (False Positive, #cc0000)  (False Negative, #cc0000)  \
0                     179995                          0
1                     179995                          0
2                     179995                          0

   (True Positive, #009933)
```

(continues on next page)

(continued from previous page)

```
0                       16613
1                       16613
2                       16613
```

… and plot the table.

0.10.1 Making ROC Curves Easier

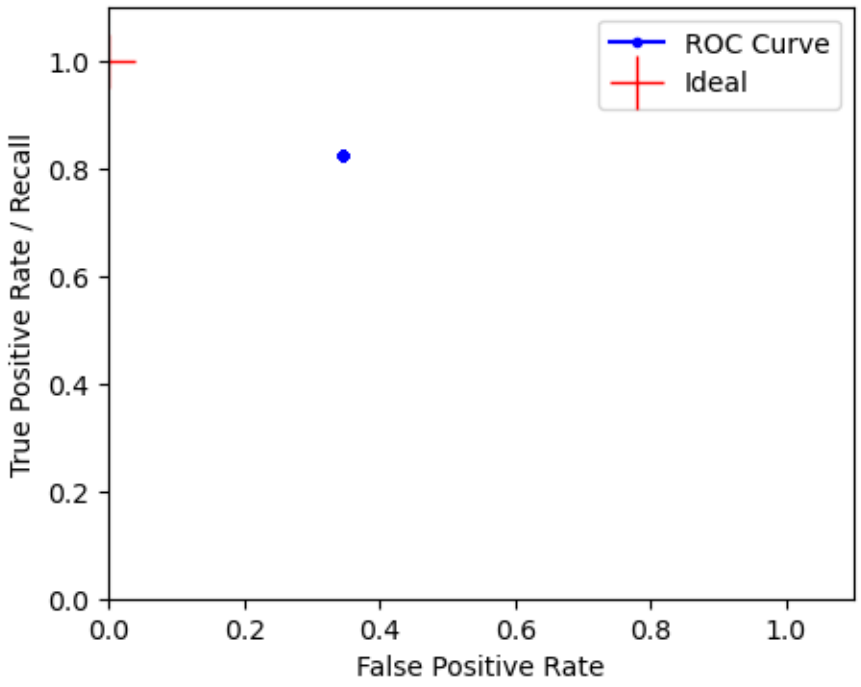
Here we show how it is done with scikit-image.

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.

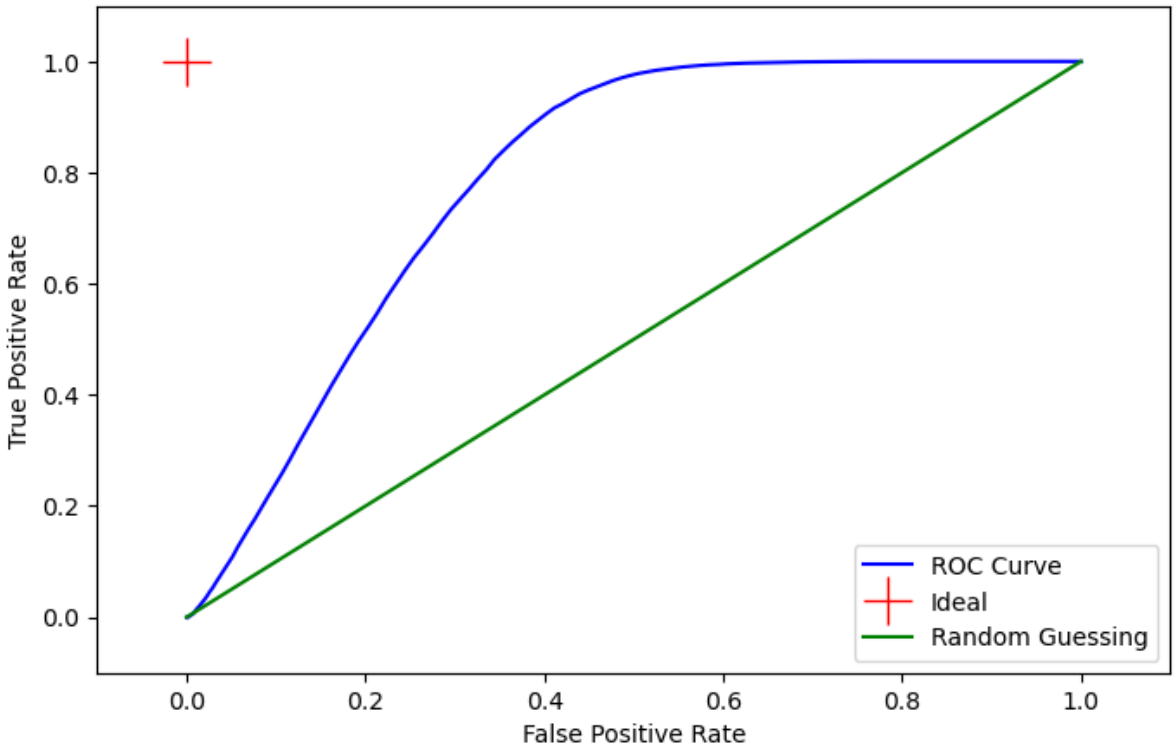
```
fig, ax1 = plt.subplots(1, 1, figsize=(5,4))
ax1.plot(roc_df['False_Positive_Rate'], roc_df['Recall']  , 'b.-', label='ROC Curve')
ax1.plot(0, 1.0, 'r+', markersize=20, label='Ideal'); ax1.set_xlim(0, 1.1); ax1.set_
↪ylim(0, 1.1);
ax1.set_ylabel('True Positive Rate / Recall')
ax1.set_xlabel('False Positive Rate')
ax1.legend(loc='upper right');
```



ROC curve for mitochondria image segmentation

```
fpr, tpr, thresholds = roc_curve(cell_seg.ravel().astype(int),
                                 cell_img.ravel())
```

```
fig,ax1 = plt.subplots(1,figsize=(8,5))
ax1.plot(fpr, tpr, 'b.-', markersize=0.01,  label='ROC Curve')
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);
```



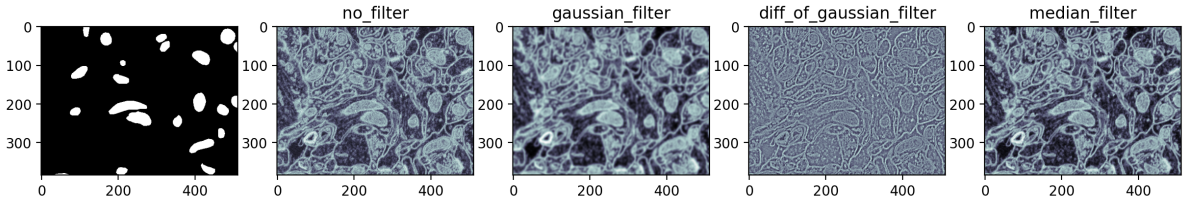
0.10.2 Explore the impact of different filters on the ROC

We have already seen what the ROC curve looks like for the original data. Some weeks ago we learnt about a lot of filters and now it is time to see how these can be used in an attempt to improve the ROC. In this example we will compare the unfiltered image to:

- Gaussian filter ($\sigma = 2$)
- Difference of Gaussian $x - G_{sigma=3} * x$
- Median size 3x3 And see what performance improvements we can achieve

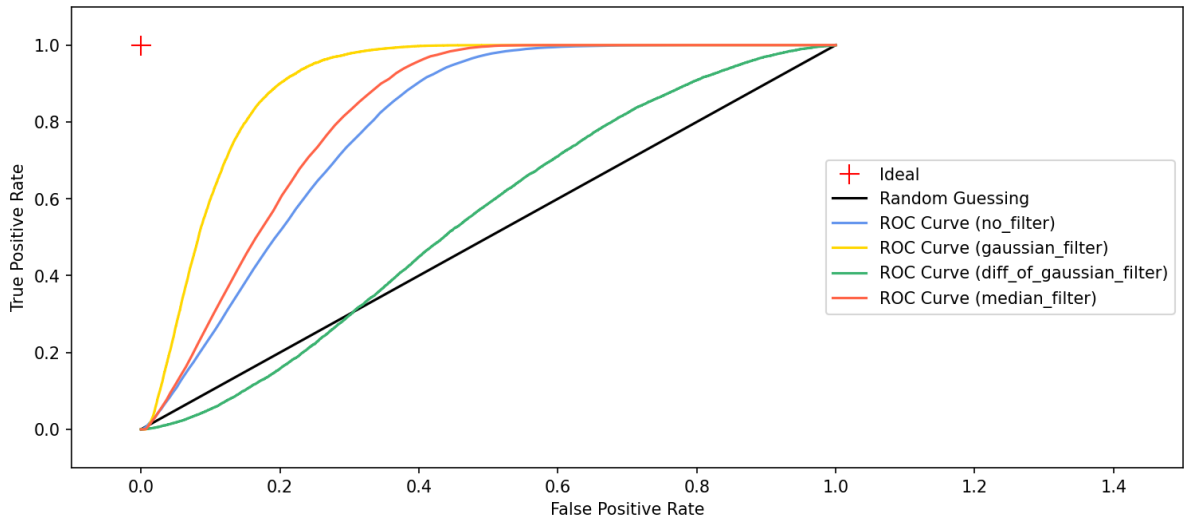
Let's produce some filtered images:

```
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  x-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')
    c_ax.set_title(c_filt.__name__)
```



ROC curves of filtered images

```
fig, ax1 = plt.subplots(1, 1,figsize=(12,5), dpi=150)

ax1.plot(0.0, 1.0, 'r+', markersize=12, label='Ideal')
ax1.plot([0, 1], [0, 1], 'k-', label='Random Guessing')

colors = ['cornflowerblue','gold','mediumseagreen','tomato']
for color,c_filt in zip(colors,[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__),color=color)

# Decorations
ax1.set_xlim(-0.1, 1.5); ax1.set_ylim(-0.1, 1.1)
ax1.set_xlabel('False Positive Rate'); ax1.set_ylabel('True Positive Rate')
ax1.legend(loc="center right", fontsize=10);
```

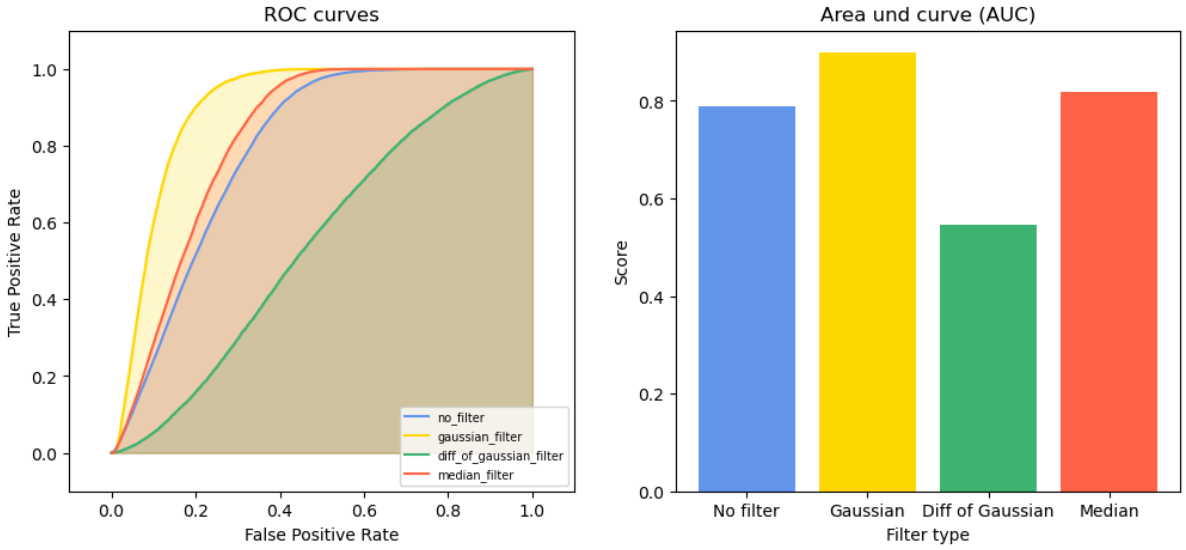


0.10.3 Area Under the Curve (AUC)

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* (AUC) which is a scalar.

```
fig, ax = plt.subplots(1, 2,figsize=(12,5), dpi=100)
colors = ['cornflowerblue','gold','mediumseagreen','tomato']
scores = []
for color, c_filt in zip(colors,[no_filter, gaussian_filter, diff_of_gaussian_filter,␣
 ↪median_filter]):
    fimg = c_filt(cell_img).ravel()
    fpr, tpr, thresholds = roc_curve(cell_seg.ravel().astype(int), fimg)
    scores.append(roc_auc_score(cell_seg.ravel().astype(int),  fimg))
    ax[0].plot(fpr, tpr, '-', markersize=0.01, color=color,label='{}'.format(c_filt.__
 ↪name__))
    ax[0].fill_between(fpr, tpr, 0, alpha=0.2, color=color)

ax[0].set_xlim(-0.1, 1.1); ax[0].set_ylim(-0.1, 1.1)
ax[0].set_xlabel('False Positive Rate'); ax[0].set_ylabel('True Positive Rate');␣
 ↪ax[0].set_title('ROC curves')
ax[0].legend(loc="lower right", fontsize=7);
names = ['No filter', 'Gaussian', 'Diff of Gaussian', 'Median']
ax[1].bar(names,scores, color=colors); plt.xlabel('Filter type'),ax[1].set_ylabel(
 ↪'Score'); ax[1].set_title('Area und curve (AUC)');
```



Armed with these tools we are ready to analyze the performance of the segmentation methods we develop to solve our image analysis tasks.

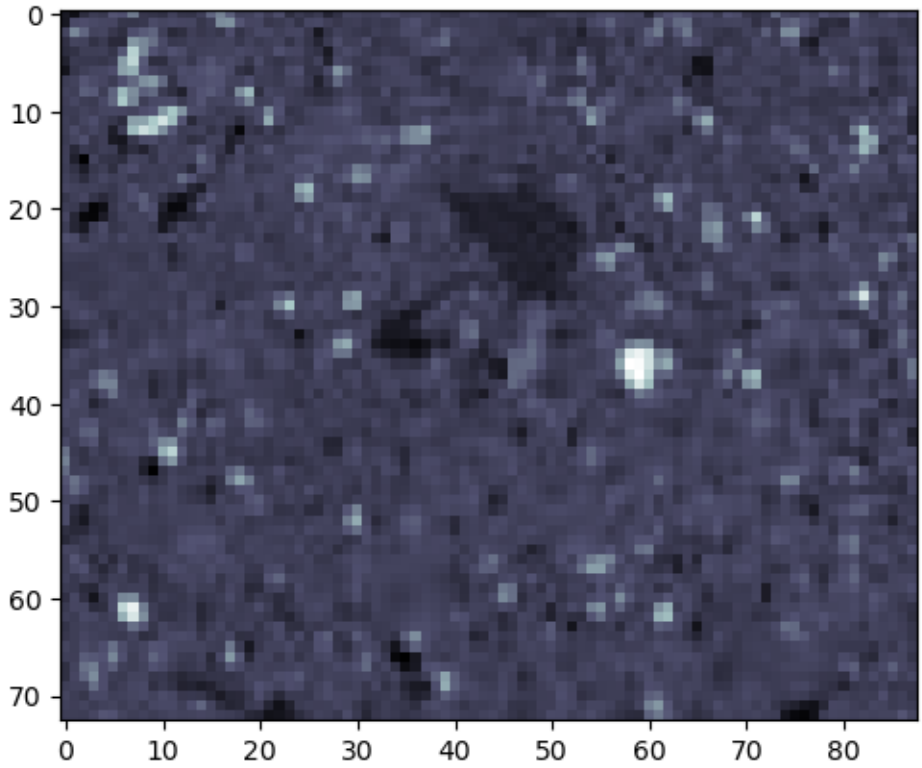
0.11 Segmenting multiple phases

0.11.1 Multiple Phases example: 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)
ax1.imshow(shale_img, cmap='bone');
```



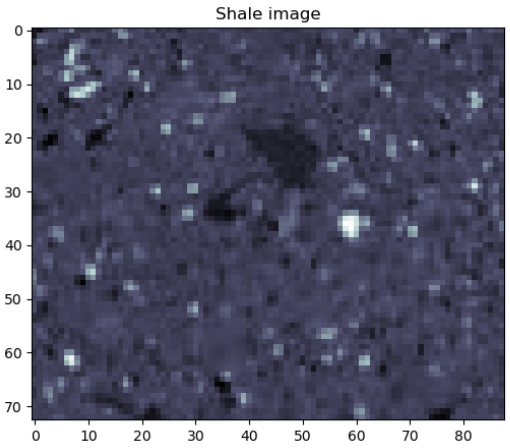
Finding three categories

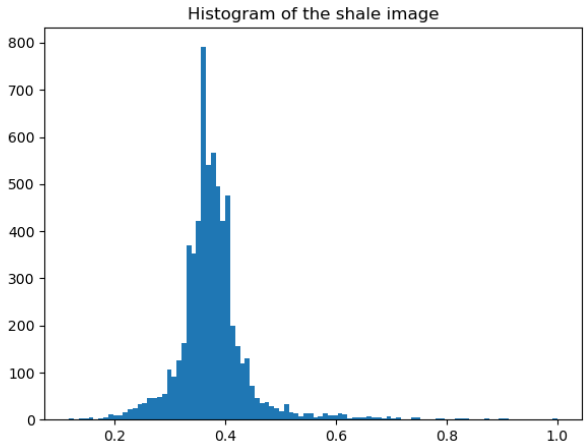
Let's take a look at the histogram as always when we face a segmentation task…

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).

```
fig, ax=plt.subplots(1,2, figsize=(15,5))
ax[0].imshow(shale_img, cmap='bone'),
ax[0].set_title('Shale image')
ax[1].hist(shale_img.ravel(), 100);
ax[1].set_title('Histogram of the shale image');
```



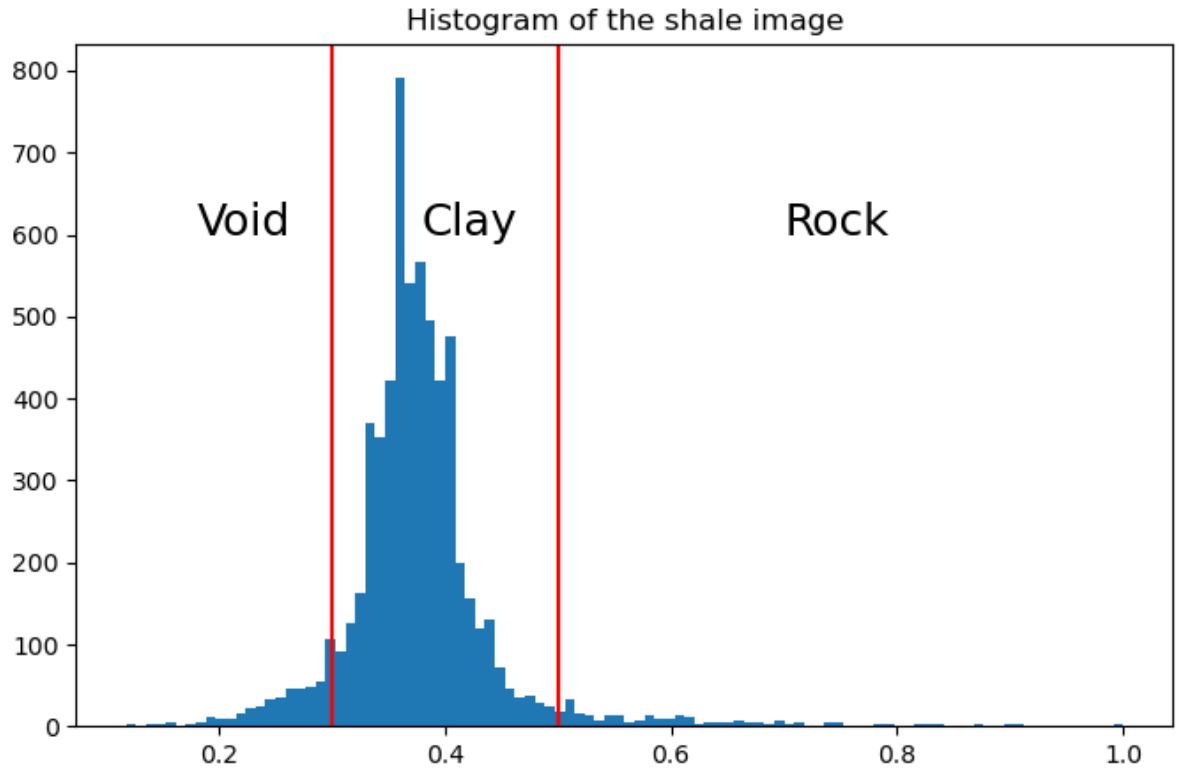


0.12 Multiple Segmentations

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

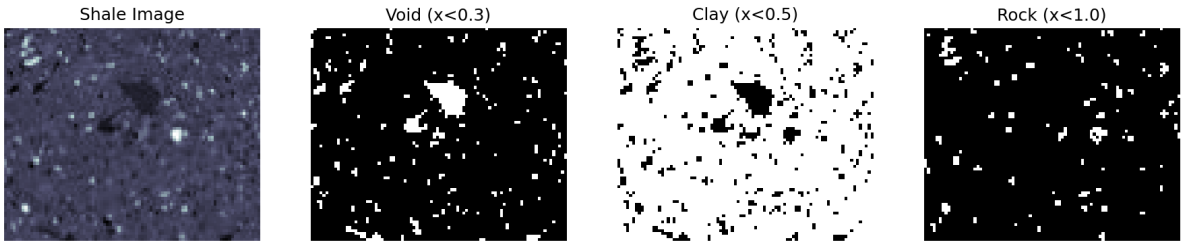
The relation can explicitly be written out as $$I(x) = \begin{cases} \text{Void}, & 0 \leq x \leq 0.3 \\ \text{Clay}, & 0.3 < x \leq 0.5 \\ \text{Rock}, & 0.5 < x \end{cases}$$

```
fig, ax=plt.subplots(1,1, figsize=(8,5))
ax.hist(shale_img.ravel(), 100);
ax.set_title('Histogram of the shale image');

thresholds = [0.3, 0.5]
ax.axvline(thresholds[0],color='r');
ax.axvline(thresholds[1],color='r');
fs=18; ypos=600; ax.text(0.18,ypos,'Void', fontsize=fs)
ax.text(0.38,ypos,'Clay', fontsize=fs)
ax.text(0.7,ypos,'Rock', fontsize=fs);
```



0.12.1 Segmentation result

```
fig, m_axs = plt.subplots(1, 4, dpi=150, figsize=(15, 5))
m_axs[0].imshow(shale_img, cmap='bone')
m_axs[0].set_title('Shale Image'); m_axs[0].axis('off')
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.imshow(c_slice, cmap='bone')
    used_vox += c_slice
    c_ax.axis('off')
    c_ax.set_title('{0} (x<{1:0.1f})'.format(c_title, c_max))
```



Segmenting multiple phases is a non-trivial problem. In particular, when the edges in the image as smooth and at low SNR. We will look into these problems next week.

0.13 Implementation of thresholding

The implementations of basic thresholds and segmentations is very easy since it is a unary operation of a single image

$$f(I(\vec{x}))$$

How is this implemented with using a programming language?

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.

0.13.1 Implementation using script languages

Python (numpy) and Matlab

The simplest is a single threshold in numpy and Matlab:

```
thresh_img = gray_img > thresh
```

A more complicated threshold:

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

Python

The task is slightly more complicated when you use standard python. Here, you have to define a mapping function with a lambda to perform the thresholding.

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

0.13.2 Implementation using traditional programming languages

In traditional programming languages you have to write some moer code as there are no predefined functions that operate directly on arrays. This means, you'll have to implement the loops yourself.

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;
    }
```

C++

```
bool* thresh_img = new bool[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;
    }
```

Alternative solution

- Image are stored as a sequence of numbers, not matter the number of dimensions.
- The pixel neighborhood is not considered.
- A single loop can be used!

0.13.3 Summary segmentation

- Validation methods from machine learning help us to understand segmentation performance
- We need a ground truth for the validation
- Multiple thresholds are needed for multi-phase materials
- The implementation of the segmentation depends on the chosen language

0.14 Morphological image processing

The segmentation is rarely perfect!

By comparing with neighborhood voxels we can improve the results.

These steps are called morphological operations.

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.

0.14.1 Noisy segmentation

We return to the original image of a cross:

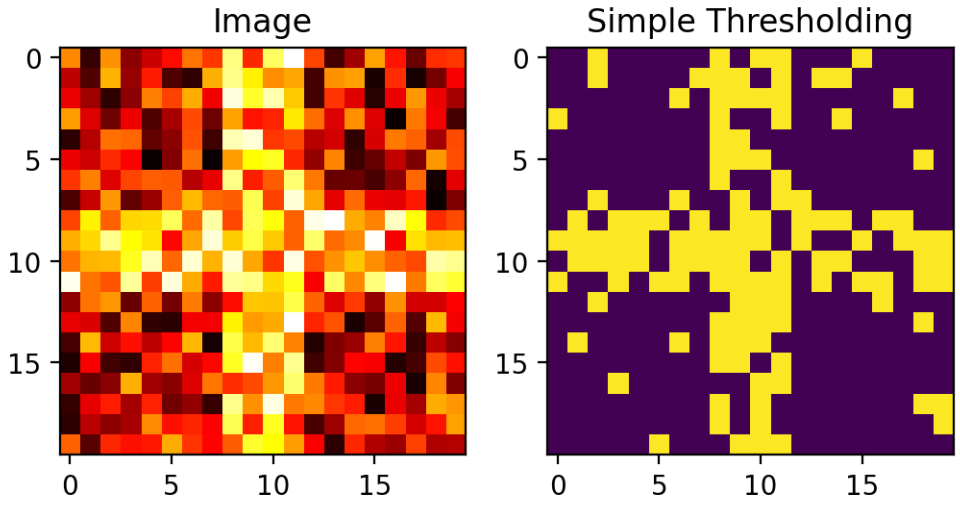
```
nx = 20
ny = 20
xx, yy = np.meshgrid(np.linspace(-10, 10, nx),
                     np.linspace(-10, 10, ny))
np.random.seed(2018)
cross_im = 1.1*((np.abs(xx) < 2)+(np.abs(yy) < 2)) + \
```

(continues on next page)

(continued from previous page)

```
    np.random.uniform(-1.0, 1.0, size=xx.shape)

thimg = cross_im > 0.8 # Let's apply a threshold
```

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(6, 3.5), dpi=200)
ax1.imshow(cross_im, cmap='hot')
ax1.set_title('Image')
ax2.imshow(thimg)
ax2.set_title('Simple Thresholding');
```



We have a lot of misclassified pixels here!

0.14.2 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

Why do we need neighborhods

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.

Fig. 1: 4-connected pixel neighborhood for 2D images.

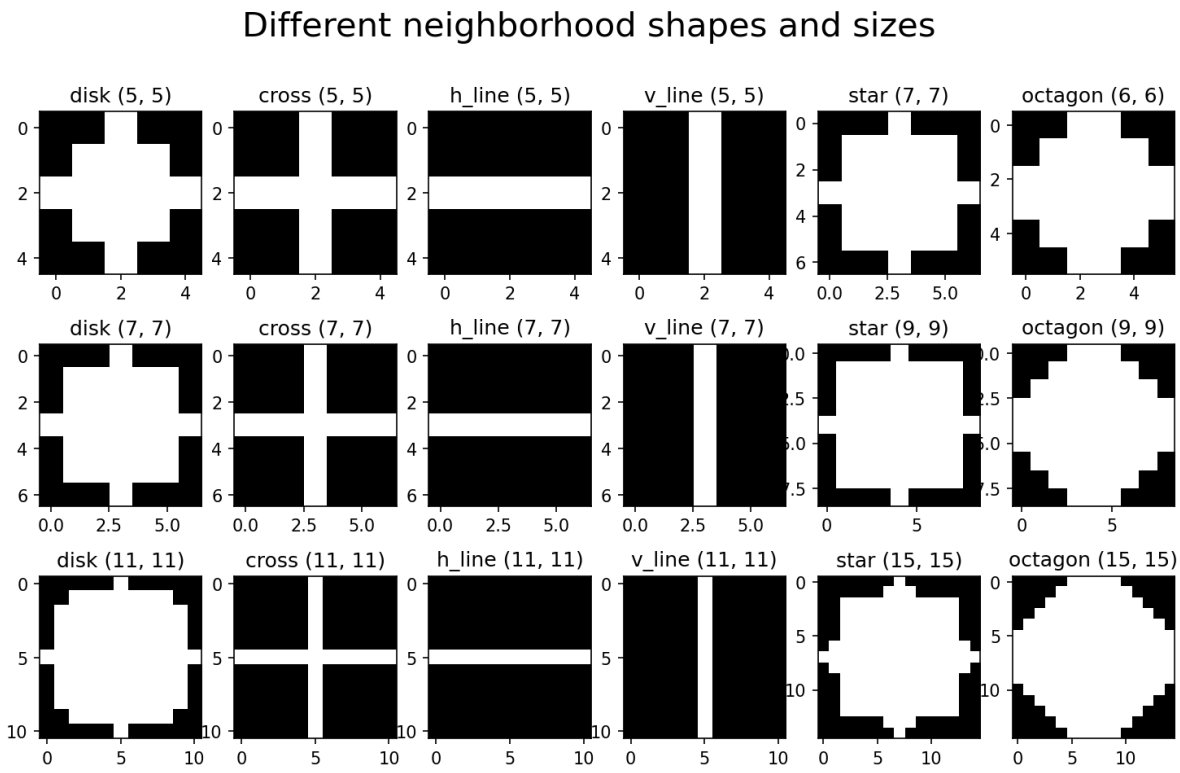
Fig. 2: 8-connected pixel neighborhood for 2D images.

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

More neighborhood shapes

```
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);
```



Fundamentals: Neighbors in 3D

Neighborhoods in 3D include the planes above and below the center pixel in addition to the neighbors in the same plane.

Fig. 3: 6-connected pixel neighborhood for 2D images.

Fig. 4: 8-connected pixel neighborhood for 2D images.

0.14.3 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

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]]
```

```
import skimage.morphology as morph
img=np.load('data/morphimage.npy')

selem = [[0,1,0],
         [1,1,1],
         [0,1,0]]

dimg=morph.dilation(img,selem)
eimg=morph.erosion(img,selem)
```

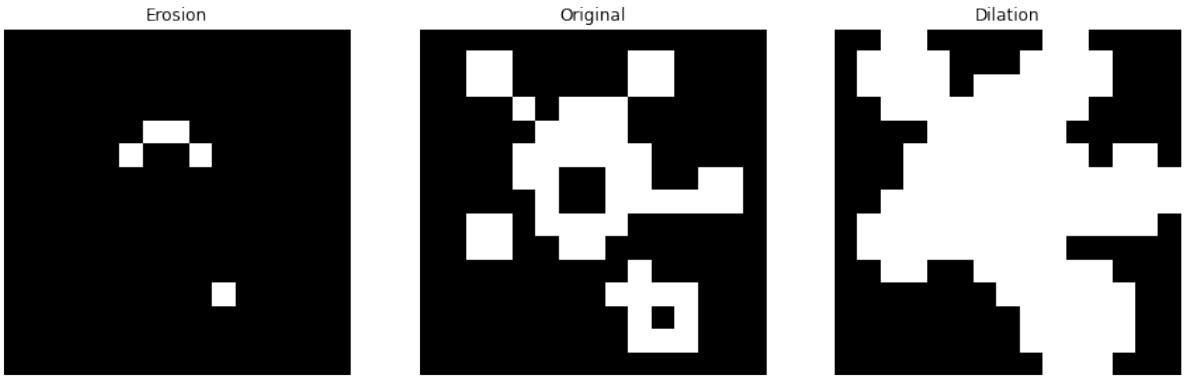
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[26], line 8
      2 img=np.load('data/morphimage.npy')
      4 selem = [[0,1,0],
      5          [1,1,1],
      6          [0,1,0]]
----> 8 dimg=morph.dilation(img,selem)
      9 eimg=morph.erosion(img,selem)

File ~/miniconda3/lib/python3.9/site-packages/skimage/morphology/misc.py:38, in␣
 ↪default_footprint.<locals>.func_out(image, footprint, *args, **kwargs)
     36 if footprint is None:
     37     footprint = ndi.generate_binary_structure(image.ndim, 1)
---> 38 return func(image, footprint=footprint, *args, **kwargs)

File ~/miniconda3/lib/python3.9/site-packages/skimage/morphology/gray.py:304, in␣
 ↪dilation(image, footprint, out, shift_x, shift_y)
    301 if out is None:
    302     out = np.empty_like(image)
--> 304 if _footprint_is_sequence(footprint):
    305     # shift and invert (see comment below) each footprint
    306     footprints = tuple(
    307         (_invert_footprint(_shift_footprint(fp, shift_x, shift_y)), n)
    308         for fp, n in footprint
    309     )
    310     return _iterate_gray_func(ndi.grey_dilation, image, footprints, out)

File ~/miniconda3/lib/python3.9/site-packages/skimage/morphology/footprints.py:37,␣
 ↪in _footprint_is_sequence(footprint)
     35 if isinstance(footprint, Sequence):
     36     if not all(_validate_sequence_element(t) for t in footprint):
---> 37         raise ValueError(
     38             "All elements of footprint sequence must be a 2-tuple where "
     39             "the first element of the tuple is an ndarray and the second "
```

(continues on next page)

(continued from previous page)

```
     40             "is an integer indicating the number of iterations."
     41         )
     42 else:
     43     raise ValueError("footprint must be either an ndarray or Sequence")

ValueError: All elements of footprint sequence must be a 2-tuple where the first␣
 ↪element of the tuple is an ndarray and the second is an integer indicating the␣
 ↪number of iterations.
```

```
fig, ax = plt.subplots(1,3,figsize=(15,6))

ax[0].imshow(eimg,cmap='gray'); ax[0].set_title('Erosion'),  ax[0].axis('off');
ax[1].imshow(img,cmap='gray');  ax[1].set_title('Original'), ax[1].axis('off');
ax[2].imshow(dimg,cmap='gray'); ax[2].set_title('Dilation'), ax[2].axis('off');
```

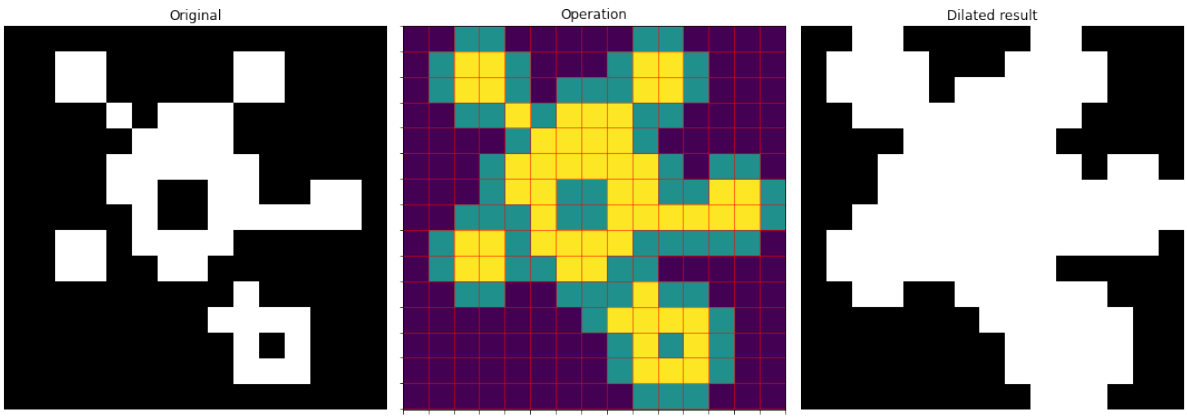


Dilation

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

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

```
fig, ax = plt.subplots(1,3,figsize=(15,6))

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()
```

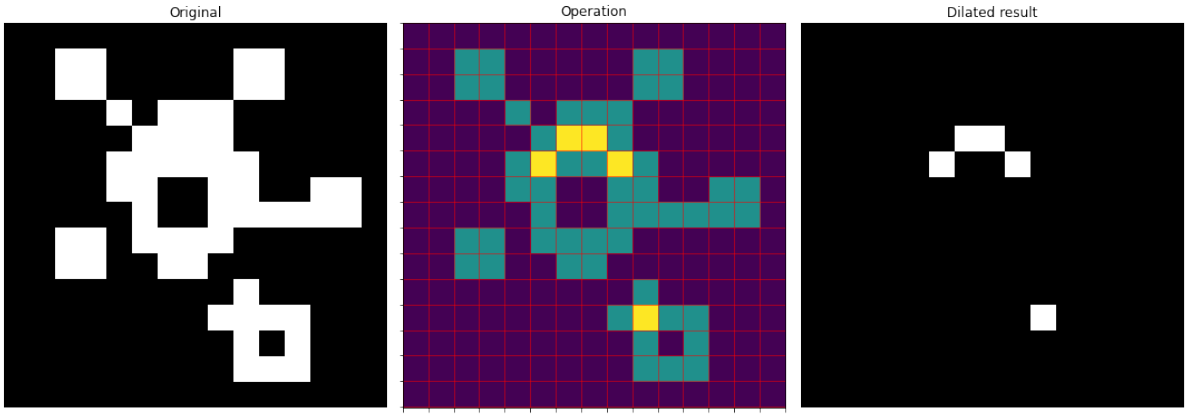


Erosion

Erosion performs the opposite task to the dilation by reducing the size of the objects in the image

```
selem= [[0,1,0],
        [1,1,1],
        [0,1,0]]
eimg=morph.erosion(img,selem)
```

```
fig, ax = plt.subplots(1,3,figsize=(15,6))

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()
```



0.14.4 Opening and Closing

Erosion and dilation removes and adds a layer of pixels to the objects in the image.

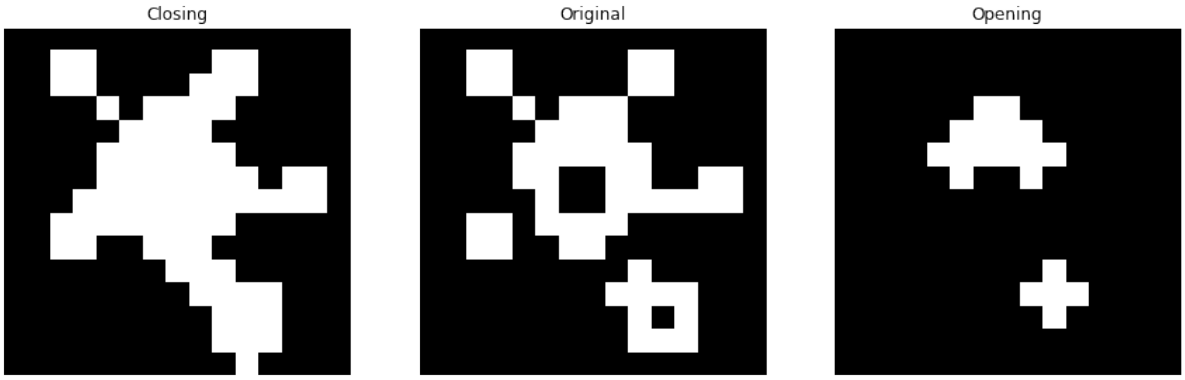
This is not desired, combining them gives two new operations:

- Opening $\delta(\epsilon(f))$
- Closing $\epsilon(\delta(f))$

These operation rebuilds most of the objects after removing unwanted features.

```
selem = np.array([[0,1,0],[1,1,1],[0,1,0]])
oimg = morph.opening(img,selem)
cimg = morph.closing(img,)
```

```
fig, ax = plt.subplots(1,3,figsize=(15,6))
ax[0].imshow(cimg,cmap='gray'); ax[0].set_title('Closing'), ax[0].axis('off');
ax[1].imshow(img,cmap='gray'); ax[1].set_title('Original'), ax[1].axis('off');

ax[2].imshow(oimg,cmap='gray'); ax[2].set_title('Opening'), ax[2].axis('off');
```



Morphological Closing

A dilation followed by an erosion operation

$$\epsilon(\delta(f))$$

- 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)

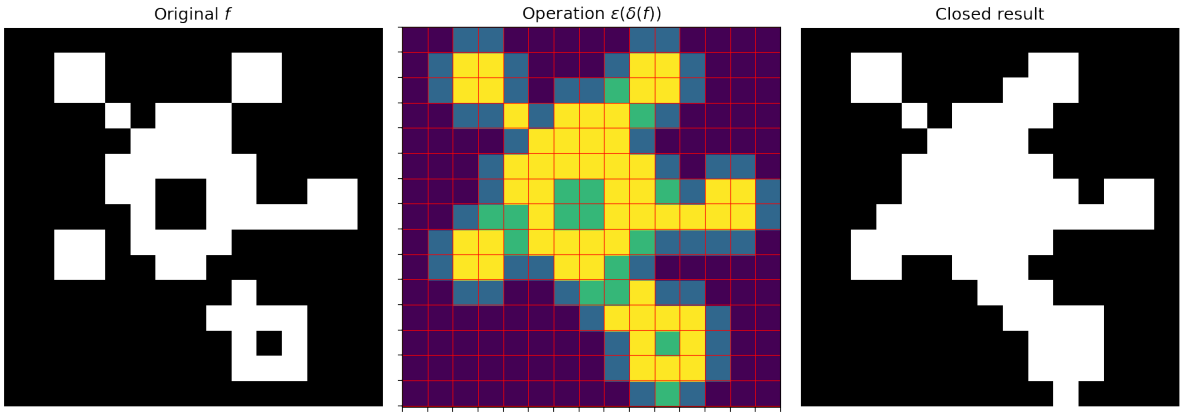
Closing is an operation you apply to remove false negatives in your image. The effect is that small holes in the objects are filled and gaps between large objects are connected.

```
fig, ax = plt.subplots(1,3,figsize=[12,6], dpi=150)

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

ax[1].imshow(img+dimg+cimg,cmap='viridis'); ax[1].set_title('Operation $\epsilon(\
↪delta(f))$')
```

(continues on next page)

(continued from previous page)

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



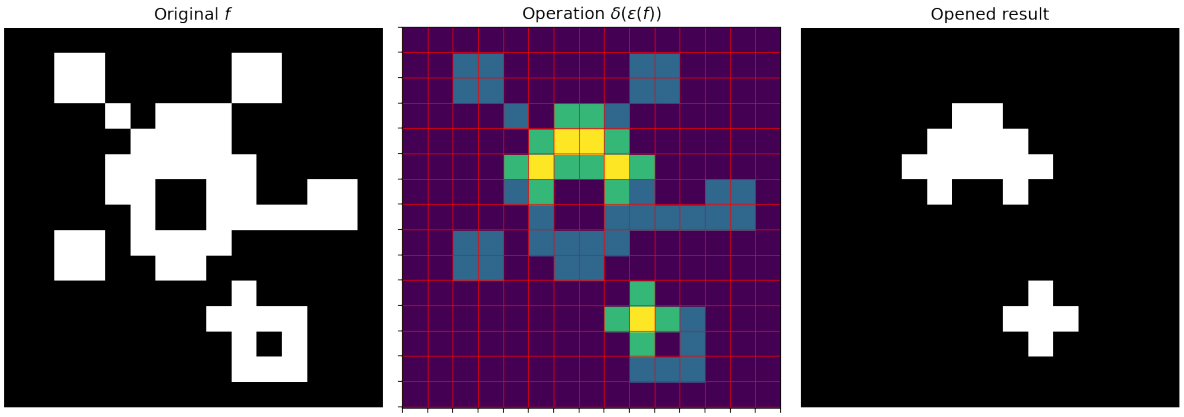
Morphological opening

An erosion followed by a dilation operation $\delta(\epsilon(f))$

- 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)

Opening is an operation you apply to remove false positives in your image. The effect is that small objects are erased and connections between large objects are removed.

```
fig,ax = plt.subplots(1,3,figsize=[12,6],dpi=150)

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

ax[1].imshow(img+eimg+oimg,cmap='viridis'); ax[1].set_title('Operation $\delta(\
↪epsilon(f))$')
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[2].imshow(oimg,cmap='gray'); ax[2].axis('off')
ax[2].set_title('Opened result')

plt.tight_layout()
```



0.15 Pitfalls with Segmentation

0.15.1 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

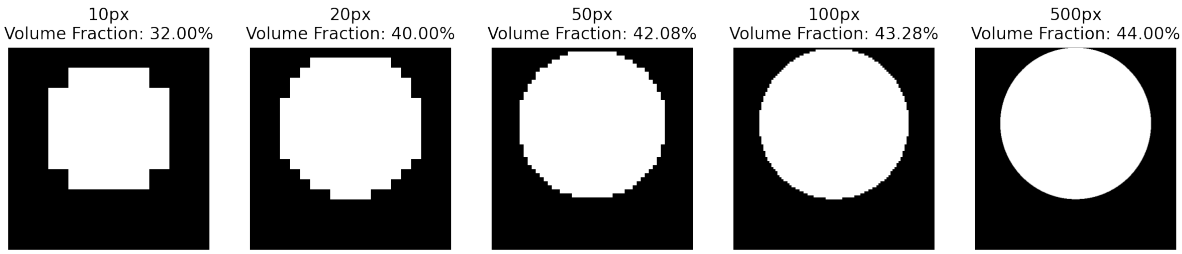
0.15.2 Thresholding structures

What happens when we threshold objects of different sizes?

In this example we create a series of spheres on different grid sizes form 10 up to 500 pixels.

```
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)
    c_ax.imshow(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')
```



Here you can see that the small objects are very pixeled and almost doesn't resemble a disk. When object size increases we see that the object looks more and more like a round disk. We also see that the volume fraction increases towards the value the resembles the volume of a true disc.

0.15.3 When is a sphere really a sphere?

We just saw that a 2D disc can be very pixelated for small radii. The same applies in 3D. In this example, you can see what a sphere looks like. The first two examples doesn't really look like a sphere, while the last one starts to look like a sphere. The plot in the last panel shows the volume error for different discrete spheres. At a raduis of about 10 pixels the error is below one percent.

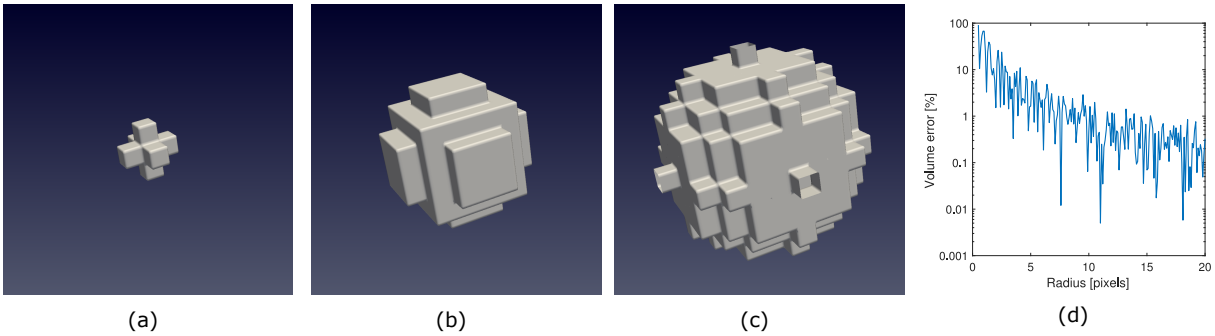


Fig. 1: Discrete spheres with increasing radius.

$$V_{error} = \frac{V_{discrete}}{V_{Analytical}}$$

Kaestner et al. 2017

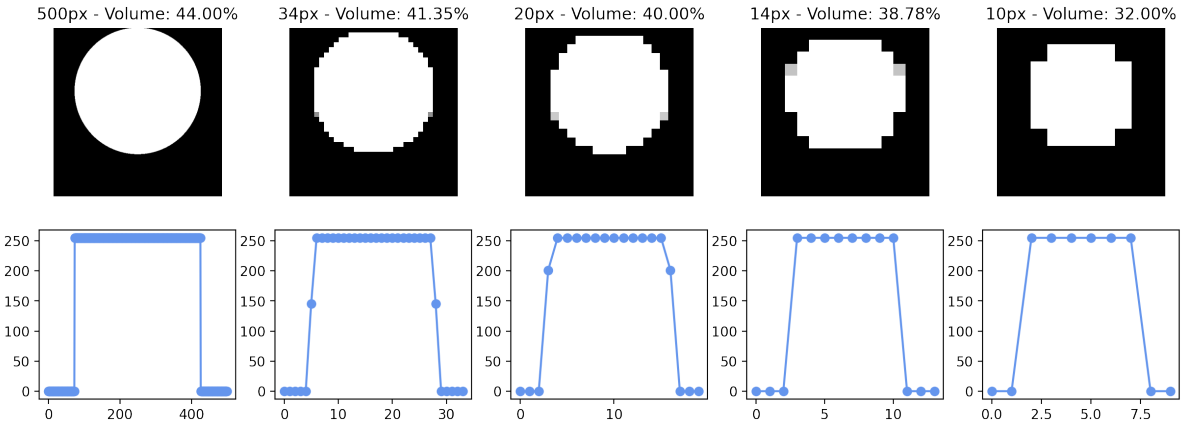
What we are learning from this study is that there is a difference between detecting a basic feature and really representing it true shape. Detection should in principle be posible withing a few pixels if the SNR is sufficiently high.

0.15.4 Rescaling

Sometimes, we want to downscale the image when it is too large. This is mostly done due to problems of fitting the images into memory or to speed up the processing. Rescaling should generally be done on gray scale images to avoid visible partial volume effects, which means that pixels done have only two values anymore at the edges.

In this example we rescale images from 500x500 down to 15x15 that the apparent volume fraction changes at the edges in some positions.

```
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.imshow(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], 'o-', color='cornflowerblue')
```



The intermediate values are in particular visible in the profiles from down sizing from 500 pixel to 20 and 34 pixels.

0.16 Summary

In todays lecture we have looked into

- The image formation process and how it relates to the segmentation problem.
- How the histogram can be used to decide how to segment an image.
- Evaluation of segmentation performance.
- The basic operations of morphological image processing
  - Using morphological operations to clean up segmented images
- Pitfall with segmentation - partial volume effects.