Quantitative Big Imaging - Shape Analysis

Anders Kaestner

Mar 26, 2025

CONTENTS

This is the lecture notes fot the 6th lecture of the Quantitative big imaging class given during the spring semester 2025 at ETH Zurich, Switzerland.

0.1 Shape analysis

0.1.1 Learning Objectives

Motivation (Why and How?)

- How do we quantify where and **how big** our objects are?
- How can we say something about the **shape**?
- How can we compare objects of **different sizes**?
- How can we **compare two images** on the basis of the shape as calculated from the images?
- How can we put objects into an
  - finite element simulation?
  - or make pretty renderings?

0.1.2 Outline

- Motivation (Why and How?)
- Object Characterization
- Volume
- Center and Extents
- Anisotropy

0.1.3 Motivation

- We have dramatically simplified our data, but there is still too much.
- We perform an experiment to see how big the bone cells are inside the tissue:
- The work flow until now is

Aquisition →
2560 x 2560 x 2160 x 32 bit

- 56GB / sample

Filtering →
2560 x 2560 x 2160 x 32 bit

- Still 56GB of less noisy data

Segmentation
2560 x 2560 x 2160 x 1 bit = 1.75GB / sample

- 1.75GB is still an aweful lot of data

… but we are not quantifying anything yet!

What did we want in the first place?

Single numbers :

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

0.1.4 Metrics

- Surface Area
- Counting
- Shape Tensor
- Principal Component Analysis
- Ellipsoid Representation
- Anisotropy, Oblateness
- Scale-free metrics
- Meshing
- Marching Cubes
- Isosurfaces

0.1.5 Literature / Useful References

General

- Jean Claude, *Morphometry with R*
- John C. Russ, *The Image Processing Handbook*, Boca Raton, CRC Press

Principal Component Analysis

- Venables, W. N. and B. D. Ripley (2002). *Modern Applied Statistics with S*, Springer-Verlag

Shape Tensors

- Doube, M.,et al. (2010). *BoneJ: Free and extensible bone image analysis in ImageJ*. Bone, 47, 1076–9.
- Mader, K. , et al. (2013). *A quantitative framework for the 3D characterization of the osteocyte lacunar system*. Bone, 57(1), 142–154.
- Wilhelm Burger, Mark Burge (2009). *Principles of Digital Image Processing: Core Algorithms*. Springer-Verlag
- B. Jähne (2005). *Digital Image Processing*. Springer-Verlag
- T. H. Reiss (1993). *Recognizing Planar Objects Using Invariant Image Features*, from Lecture notes in computer science, p. 676. Springer

- Image moments

0.1.6 Let's load some modules for python

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.collections import PatchCollection
from matplotlib.patches     import Rectangle

from matplotlib.animation import FuncAnimation
from IPython.display      import HTML
from skimage.morphology   import label
from skimage.morphology   import erosion, disk
from skimage.measure      import regionprops
from skimage.io           import imread
from skimage.morphology   import label

from IPython.display        import Markdown, display
from sklearn.neighbors      import KNeighborsClassifier
from sklearn.decomposition  import PCA
import webcolors
import sys
sys.path.append('../common')
import plotsupport as ps

from collections import defaultdict

%matplotlib inline
```

0.2 Component labelling

Segmentation

- Segmentation identified pixels belonging to some class
  - a single set containing all pixels!

To measure objects in an image, they need to be uniquely identified.

Once we have a clearly segmented image, it is often helpful to identify the sub-components of this image. Connected component labeling is one of the first labeling algorithms you come across. This is the easist method for identifying these subcomponents which again uses the neighborhood $\mathcal{N}$ as a criterion for connectivity. The principle of the this algorithm class is that it is it puts labels on all pixels that touch each other. This means that the algorithm searches the neighborhood of each pixel and checks if it is marked as an object. If, "yes" then the neighbor pixel will be assigned the same class a the pixels we started from. This is an iterative process until each object in the image has one unique label.

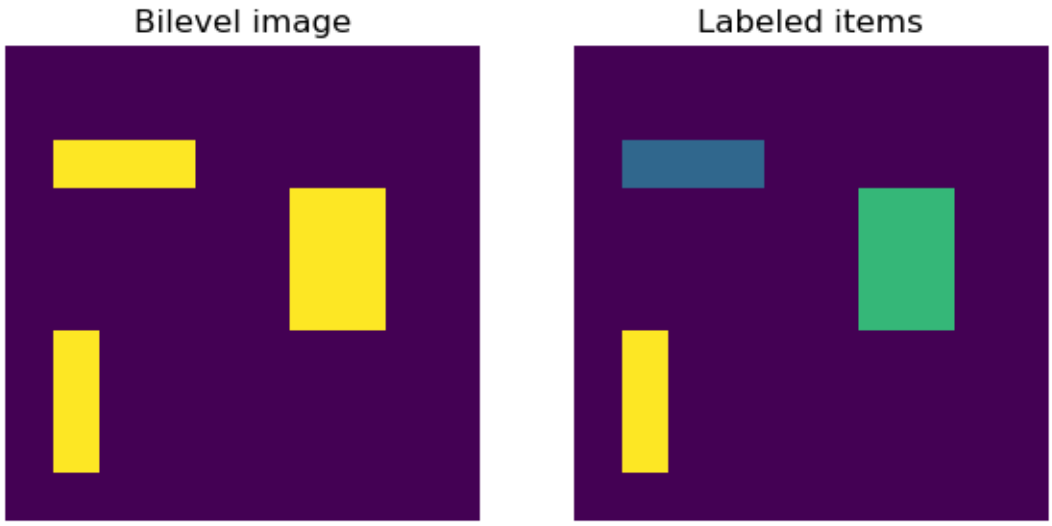
In general, the approach works well since usually when different regions are touching, they are related. It runs into issues when you have multiple regions which agglomerate together, for example a continuous pore network (1 object) or a cluster of touching cells.

- Basic component labelling
  - give the same label to all pixels touching each other.

- has its drawbacks… touching item are treated as one

```
img = np.zeros((100,100))
img[20:30,10:40]=1
img[30:60,60:80]=1
img[60:90,10:20]=1

lbl = label(img)
```

```
# visualization
fig,ax=plt.subplots(1,2,figsize=(7,4))
ax[0].imshow(img, interpolation='none'), ax[0].axis('off')
ax[0].set_title('Bilevel image');
ax[1].imshow(lbl,interpolation='none'),ax[1].axis('off')
ax[1].set_title('Labeled items');
```

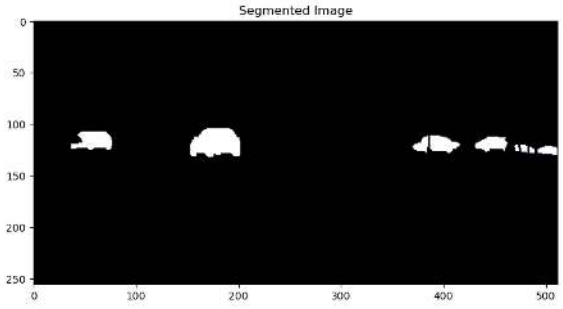


0.2.1 A cityscape image

To demonsstrate the connected components labeling we need an image. Here, we show some examples from Cityscape Data taken in Aachen (https://www.cityscapes-dataset.com/). The picture represents a street with cars in a city. The cars are also provided as a segmentation mask. This saves us the trouble of finding them in the first place.

```
car_img = imread('figures/aachen_img.png')
seg_img = imread('figures/aachen_label.png')[::4, ::4] == 26
print('image dimensions', car_img.shape, seg_img.shape)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 8))
ax1.imshow(car_img)
ax1.set_title('Input Image')

ax2.imshow(seg_img, cmap='bone')
ax2.set_title('Segmented Image');
```

```
image dimensions (256, 512, 4) (256, 512)
```



0.2.2 Connected component labeling in python

```
from skimage.morphology import label
help(label)
```

```
Help on function label in module skimage.measure._label:

label(label_image, background=None, return_num=False, connectivity=None)
    Label connected regions of an integer array.

    Two pixels are connected when they are neighbors and have the same value.
    In 2D, they can be neighbors either in a 1- or 2-connected sense.
    The value refers to the maximum number of orthogonal hops to consider a
    pixel/voxel a neighbor::

      1-connectivity     2-connectivity     diagonal connection close-up

            [ ]           [ ]  [ ]  [ ]             [ ]
             |              \  |  /                |  <- hop 2
      [ ]--[x]--[ ]      [ ]--[x]--[ ]        [x]--[ ]
             |              /  |  \              hop 1
            [ ]           [ ]  [ ]  [ ]

    Parameters
    ----------
    label_image : ndarray of dtype int
        Image to label.
    background : int, optional
        Consider all pixels with this value as background pixels, and label
        them as 0. By default, 0-valued pixels are considered as background
        pixels.
    return_num : bool, optional
        Whether to return the number of assigned labels.
    connectivity : int, optional
        Maximum number of orthogonal hops to consider a pixel/voxel
        as a neighbor.
        Accepted values are ranging from  1 to input.ndim. If ``None``, a full
        connectivity of ``input.ndim`` is used.

    Returns
    -------
    labels : ndarray of dtype int
        Labeled array, where all connected regions are assigned the
        same integer value.
```

(continues on next page)

(continued from previous page)

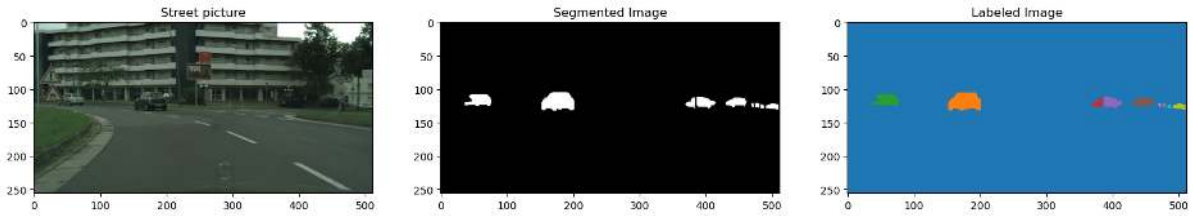
```
num : int, optional
    Number of labels, which equals the maximum label index and is only
    returned if return_num is `True`.

See Also
--------
skimage.measure.regionprops
skimage.measure.regionprops_table

References
----------
.. [1] Christophe Fiorio and Jens Gustedt, "Two linear time Union-Find
       strategies for image processing", Theoretical Computer Science
       154 (1996), pp. 165-181.
.. [2] Kensheng Wu, Ekow Otoo and Arie Shoshani, "Optimizing connected
       component labeling algorithms", Paper LBNL-56864, 2005,
       Lawrence Berkeley National Laboratory (University of California),
       http://repositories.cdlib.org/lbnl/LBNL-56864

Examples
--------
>>> import numpy as np
>>> x = np.eye(3).astype(int)
>>> print(x)
[[1 0 0]
 [0 1 0]
 [0 0 1]]
>>> print(label(x, connectivity=1))
[[1 0 0]
 [0 2 0]
 [0 0 3]]
>>> print(label(x, connectivity=2))
[[1 0 0]
 [0 1 0]
 [0 0 1]]
>>> print(label(x, background=-1))
[[1 2 2]
 [2 1 2]
 [2 2 1]]
>>> x = np.array([[1, 0, 0],
...               [1, 1, 5],
...               [0, 0, 0]])
>>> print(label(x))
[[1 0 0]
 [1 1 2]
 [0 0 0]]
```

0.2.3 Labels in the cityscape image

When we apply the `label` operation on the car mask, we see the that each car is assigned a color. There are however some cars that get multiple classes. This is because they were divided into several segments due to objects like tree and trafic signs.

```
from skimage.morphology import label
lab_img = label(seg_img)
```

```
fig, (ax0,ax1, ax2) = plt.subplots(1, 3, figsize=(20, 8), dpi=100)
ax0.imshow(car_img)
ax0.set_title('Street picture')
ax1.imshow(seg_img, cmap='bone')
ax1.set_title('Segmented Image')
ax2.imshow(lab_img, cmap=plt.cm.tab10,interpolation='none')
ax2.set_title('Labeled Image');
```



0.2.4 Area of each segment

Now, we can start making measurements in the image. A first thing that comes to mind is to compute the area of the objects. This is a very simple operation; we only have to count the number of pixels belonging to each label.

$$Area_i = \#\{x | f(x) \in i\}$$

Python coding

```
lbls = np.unique(lab_img)
areas = []
for L in lbls :
    areas.append(lab_img[lab_img==L].sum())

df = pd.DataFrame({'lbl': lbls, 'area' : areas})
df.transpose()
```

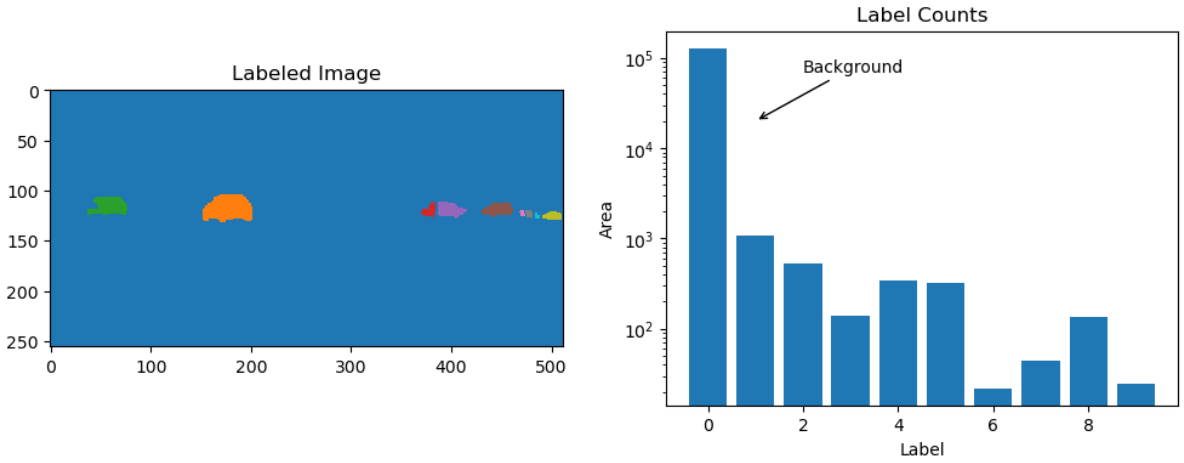
	0	1	2	3	4	5	6	7	8	9
lbl	0	1	2	3	4	5	6	7	8	9
area	0	1091	1070	420	1368	1615	132	308	1072	225

Using the histogram

Usually, we want to know the are of all items in the images. We could do this by writing a loop that goes through all labels in the image and compute each one of them. There is however an operation that does this much easier: the histogram.

We can use a histogram with the same number of bins are there are labels in the image. This would give us the size distribution in of the objects in one single operation.

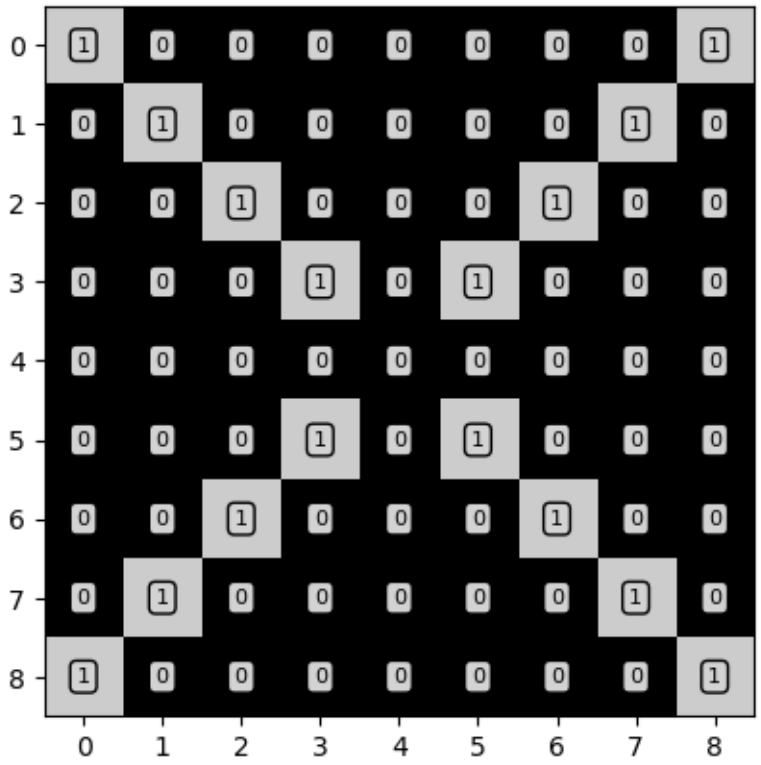
```
fig, ax = plt.subplots(1, 2, figsize=(12,4))

ax[0].imshow(lab_img, cmap=plt.cm.tab10,interpolation='none')
ax[0].set_title('Labeled Image');
# ax[1].hist(lab_img.ravel())

hb,ha = np.histogram(lab_img.ravel(),bins=np.arange(0,lab_img.max()-lab_img.min()+2))

plt.bar(ha[:-1],hb ,align="center")

ax[1].set_title('Label Counts')
ax[1].set_yscale('log')
ax[1].set_xlabel('Label')
ax[1].set_ylabel('Area');

ax[1].annotate("Background",xy=(1,2e4),xytext=(2,7e4),arrowprops={"arrowstyle":"->"});
```



0.3 A component labeling algorithm

We start off with all of the pixels in either foreground (1) or background (0)

```
seg_img = np.eye(9, dtype=int)
seg_img[4, 4] = 0
seg_img += seg_img[::-1]
ps.heatmap(seg_img,cmap='nipy_spectral',fontsize=8,precision=0);
```

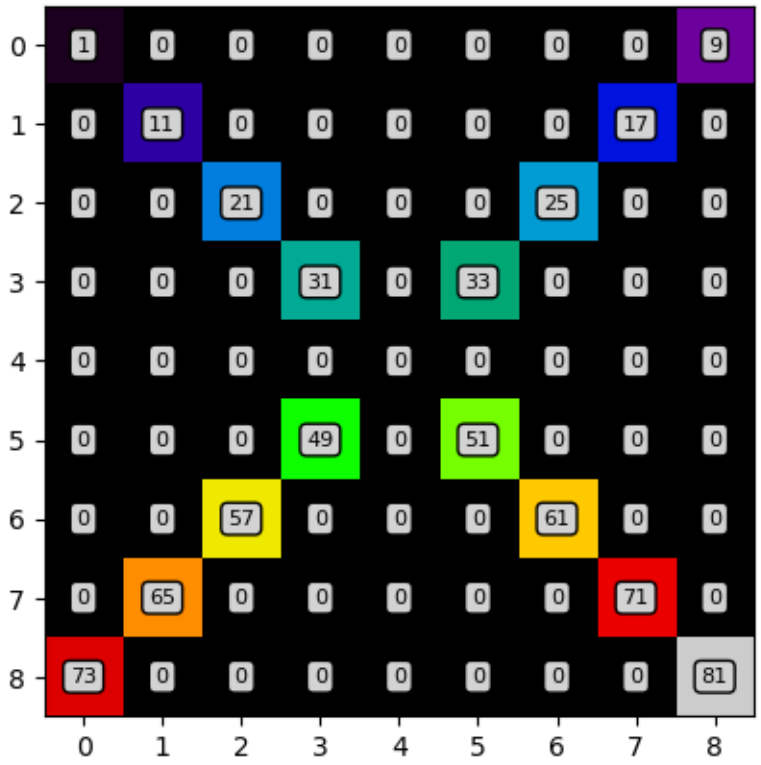


0.3.1 Labeling initialization

Give each point in the image a unique label

- For each point $(x, y) \in$ Foreground
- Set value to $I_{x,y} = x + y * width + 1$

```
idx_img = np.zeros_like(seg_img)

x,y = np.meshgrid(np.arange(0,seg_img.shape[0]),np.arange(0,seg_img.shape[1]))

nonzero = seg_img>0
idx_img[nonzero] = x[nonzero]+y[nonzero]*seg_img.shape[0]+1

ps.heatmap(idx_img,cmap='nipy_spectral',fontsize=8,precision=0);
```



0.3.2 A brushfire labeling algorithm

In a brushfire-style algorithm

- For each point $(x, y) \in$ Foreground
  - For each point $(x', y') \in \mathcal{N}(x, y)$
  - if $(x', y') \in$ Foreground
    - Set the label to $\max(I_{x,y}, I_{x',y'})$
- Repeat until no more labels have been changed

  Note: Local maximum is the same as gray level dilation.

Implementation of the brush fire algorithm

```
import skimage.morphology as morph

def brushfire(img,footprint=np.ones([3,3]),Niterations=4):
    last_img = img.copy()
    img_list = [last_img]

    mask = last_img > 0
    it_list = []
    for iteration in range(Niterations):
        cur_img = morph.dilation(last_img,footprint=footprint) * mask
```

(continues on next page)

(continued from previous page)

```
        img_list.append(cur_img)

        if (cur_img == last_img).all():
            print('Done')
            break
        else:
            print('Iteration', iteration,
                  'Groups', len(np.unique(cur_img[cur_img > 0].ravel())),
                  'Changes', np.sum(cur_img != last_img))
            it_list.append({'Iteration': iteration, 'Groups' : len(np.unique(cur_
↪img[cur_img > 0].ravel())), 'Changes' : np.sum(cur_img != last_img)})
            last_img = cur_img

    return img_list,it_list
```

Implementing the brushfire as described above would require four nested loops which is quite hard to understand.

Therefore we implement the algorithm using the grayscale dilation operation, which provides a local maximum for each pixel in the image. The problem with the dilation is that it acts on all pixels, not only the pixels belongning to the structure. This results in an image with more foreground pixels than before. The solution is to mask the result with the foreground pixels.

Looking at the iterations

```
img_list = brushfire(idx_img)
```

```
Iteration 0 Groups 12 Changes 12
Iteration 1 Groups 8 Changes 8
Iteration 2 Groups 4 Changes 4
Done
```

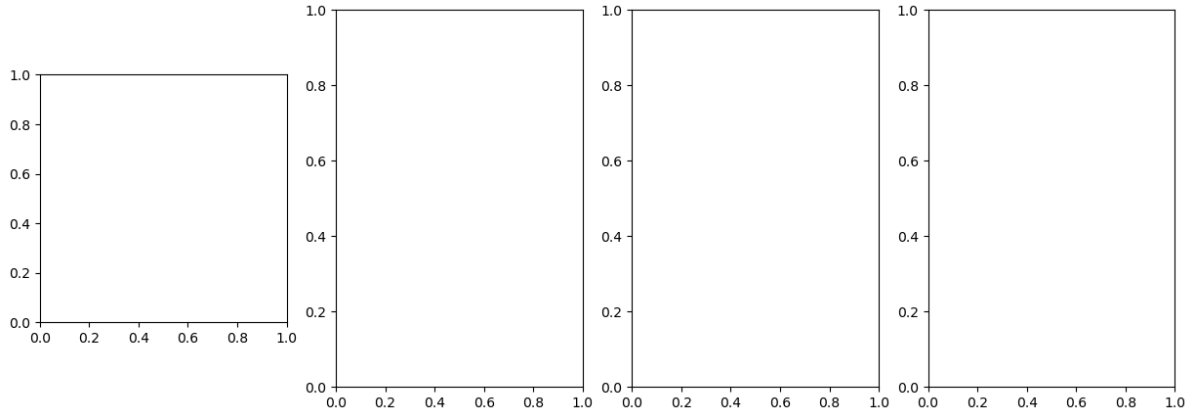
```
fig, m_axs = plt.subplots(1, 4, figsize=(15, 5)); m_axs=m_axs.ravel()
for idx,(c_ax,cur_img) in enumerate(zip(m_axs,img_list)):
    ps.heatmap(cur_img,cmap='nipy_spectral',fontsize=8,precision=0,ax=c_ax);
    c_ax.set_title('Iteration #{}'.format(idx))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[14], line 3
      1 fig, m_axs = plt.subplots(1, 4, figsize=(15, 5)); m_axs=m_axs.ravel()
      2 for idx,(c_ax,cur_img) in enumerate(zip(m_axs,img_list)):
----> 3     ps.heatmap(cur_img,cmap='nipy_spectral',fontsize=8,precision=0,ax=c_
 ↪ax);
      4     c_ax.set_title('Iteration #{}'.format(idx))

File ~/Lectures/Quantitative-Big-Imaging-2025/Lectures/Lecture-06/../common/
 ↪plotsupport.py:20, in heatmap(img, ax, cmap, fontsize, precision, bgbox, vmin,␣
 ↪vmax)
     17 if ax is None :
     18     _,ax=plt.subplots(1)
---> 20 ax.imshow(img,cmap=cmap,origin='upper', vmin=vmin, vmax=vmax)
     21 ax.set(xticks=np.arange(0,img.shape[1]),yticks=np.arange(0,img.shape[0]))
```

(continues on next page)

(continued from previous page)

```
     22 props = dict(boxstyle='round', facecolor='lightgray', alpha=1)

File /opt/anaconda3/envs/qbi2025/lib/python3.9/site-packages/matplotlib/__init__.
↪py:1473, in _preprocess_data.<locals>.inner(ax, data, *args, **kwargs)
   1470 @functools.wraps(func)
   1471 def inner(ax, *args, data=None, **kwargs):
   1472     if data is None:
-> 1473         return func(
   1474             ax,
   1475             *map(sanitize_sequence, args),
   1476             **{k: sanitize_sequence(v) for k, v in kwargs.items()})
   1478     bound = new_sig.bind(ax, *args, **kwargs)
   1479     auto_label = (bound.arguments.get(label_namer)
   1480                   or bound.kwargs.get(label_namer))

File /opt/anaconda3/envs/qbi2025/lib/python3.9/site-packages/matplotlib/axes/_axes.
↪py:5895, in Axes.imshow(self, X, cmap, norm, aspect, interpolation, alpha, vmin,␣
↪vmax, origin, extent, interpolation_stage, filternorm, filterrad, resample, url,␣
↪**kwargs)
   5892 if aspect is not None:
   5893     self.set_aspect(aspect)
-> 5895 im.set_data(X)
   5896 im.set_alpha(alpha)
   5897 if im.get_clip_path() is None:
   5898     # image does not already have clipping set, clip to Axes patch

File /opt/anaconda3/envs/qbi2025/lib/python3.9/site-packages/matplotlib/image.
↪py:729, in _ImageBase.set_data(self, A)
    727 if isinstance(A, PIL.Image.Image):
    728     A = pil_to_array(A)  # Needed e.g. to apply png palette.
--> 729 self._A = self._normalize_image_array(A)
    730 self._imcache = None
    731 self.stale = True

File /opt/anaconda3/envs/qbi2025/lib/python3.9/site-packages/matplotlib/image.
↪py:697, in _ImageBase._normalize_image_array(A)
    695     A = A.squeeze(-1)  # If just (M, N, 1), assume scalar and apply␣
↪colormap.
    696 if not (A.ndim == 2 or A.ndim == 3 and A.shape[-1] in [3, 4]):
--> 697     raise TypeError(f"Invalid shape {A.shape} for image data")
    698 if A.ndim == 3:
    699     # If the input data has values outside the valid range (after
    700     # normalisation), we issue a warning and then clip X to the bounds
    701     # - otherwise casting wraps extreme values, hiding outliers and
    702     # making reliable interpretation impossible.
    703     high = 255 if np.issubdtype(A.dtype, np.integer) else 1

TypeError: Invalid shape (5, 9, 9) for image data
```



Let's animate the iterations

```
%matplotlib inline
fig, c_ax = plt.subplots(1, 1, figsize=(5, 5), dpi=100)

def update_frame(i):
    plt.cla()
    ps.heatmap(img_list[i],cmap='nipy_spectral',fontsize=8,precision=0,ax=c_ax);

    c_ax.set_title('Iteration #{}, Groups {}'.format(i+1,
                                                    len(np.unique(img_list[i][img_
 ↪list[i] > 0].ravel()))))
# write animation frames
anim_code = FuncAnimation(fig, update_frame, frames=len(img_list)-1,
                          interval=1000, repeat_delay=2000).to_html5_video()


plt.close('all')
HTML(anim_code)
```

```
<IPython.core.display.HTML object>
```

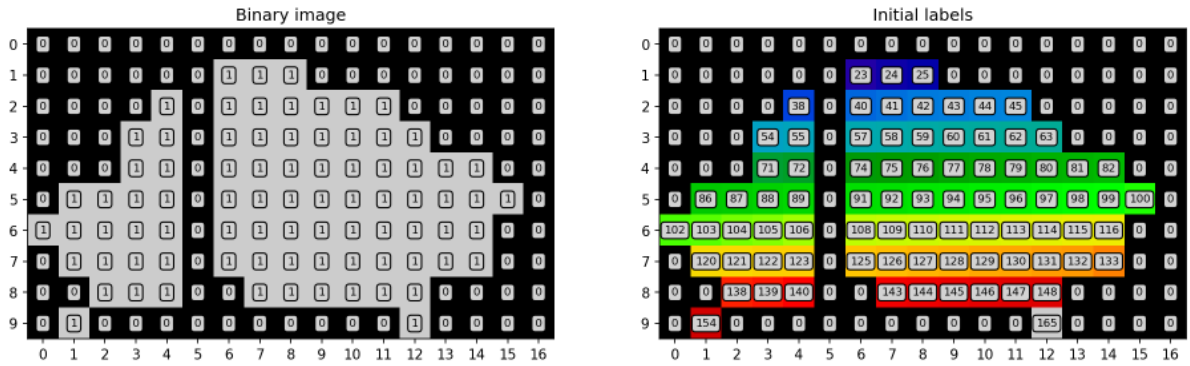
Some comments on the brushfire algorithm

- The image very quickly converges and after 4 iterations the task is complete.
- For larger more complicated images with thousands of components this task can take longer,
- There exist much more efficient algorithms for labeling components which alleviate this issue.
  - Rosenfeld & Pfalz, *Sequential Operations in Digital Picture Processing*, 1966
  - Soille, Morphological Image Processing, page 38, 2004
  - Wikipedia

0.3.3 Bigger Images

How does the same algorithm apply to bigger images?

```
seg_img = (imread('figures/aachen_label.png')[::4, ::4] == 26)[110:130:2, 370:420:3]
seg_img[9, 1] = 1
_, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 7), dpi=150)
ps.heatmap(seg_img,cmap='nipy_spectral',fontsize=8,precision=0,ax=ax1);
ax1.set_title('Binary image')

idx_img = seg_img * np.arange(len(seg_img.ravel())).reshape(seg_img.shape)
ps.heatmap(idx_img,cmap='nipy_spectral',fontsize=8,precision=0,ax=ax2);
ax2.set_title('Initial labels');
```



Run the labelling on the car image

```
img_list3,it3=brushfire(idx_img,Niterations=99)
```

```
Iteration 0 Groups 62 Changes 78
Iteration 1 Groups 45 Changes 75
Iteration 2 Groups 31 Changes 68
Iteration 3 Groups 21 Changes 52
Iteration 4 Groups 13 Changes 37
Iteration 5 Groups 7 Changes 23
Iteration 6 Groups 3 Changes 10
Iteration 7 Groups 2 Changes 3
Done
```

Let's animate the iterations

```
fig, c_ax = plt.subplots(1, 1, figsize=(8, 6))

def update_frame(i):
    plt.cla()
    ps.heatmap(img_list3[i],cmap='nipy_spectral',fontsize=8,precision=0,ax=c_ax);
    c_ax.set_title('Iteration #{}, Groups {}'.format(i+1,
                                                    len(np.unique(img_list3[i][img_
 ↪list3[i] > 0].ravel()))))
```

(continues on next page)

(continued from previous page)

```
# write animation frames
anim_code = FuncAnimation(fig,
                          update_frame,
                          frames=len(img_list3)-1,
                          interval=500,
                          repeat_delay=1000).to_html5_video()
plt.close('all')
HTML(anim_code)
```

```
<IPython.core.display.HTML object>
```

0.3.4 Different Neighborhoods

We can expand beyond the 3x3 neighborhood to a 5x5 for example

```
img_list5,it5=brushfire(idx_img,footprint=np.ones([5,5]),Niterations=99)
```

```
Iteration 0 Groups 48 Changes 79
Iteration 1 Groups 23 Changes 67
Iteration 2 Groups 7 Changes 37
Iteration 3 Groups 2 Changes 16
Iteration 4 Groups 2 Changes 10
Iteration 5 Groups 1 Changes 5
Done
```

Animate the 5x5 labeling iterations

```
fig, c_ax = plt.subplots(1, 1, figsize=(8, 6))

def update_frame(i):
    plt.cla()

    ps.heatmap(img_list5[i],cmap='nipy_spectral',fontsize=8,precision=0,ax=c_ax);
    c_ax.set_title('Iteration #{}, Groups {}'.format(i+1,
                                                     len(np.unique(img_list5[i][img_
 ↪list5[i] > 0].ravel()))))

# write animation frames
anim_code = FuncAnimation(fig,
                          update_frame,
                          frames=len(img_list5)-1,
                          interval=500,
                          repeat_delay=1000).to_html5_video()
plt.close('all')
HTML(anim_code)
```

```
<IPython.core.display.HTML object>
```

0.3.5 Or a smaller kernel

By using a smaller kernel (in this case where $\sqrt{x^2+y^2} <= 1$, we cause the number of iterations to fill to increase and prevent the last pixel from being grouped since it is only connected diagonally

0	1	0
1	1	1
0	1	0

```
img_list1,it1=brushfire(idx_img,footprint=np.array([[0,1,0],[1,1,1],[0,1,0]]),
 ↪Niterations=99)
```

```
Iteration 0 Groups 68 Changes 77
Iteration 1 Groups 54 Changes 74
Iteration 2 Groups 42 Changes 69
Iteration 3 Groups 31 Changes 61
Iteration 4 Groups 21 Changes 52
Iteration 5 Groups 12 Changes 43
Iteration 6 Groups 10 Changes 31
Iteration 7 Groups 9 Changes 24
Iteration 8 Groups 8 Changes 18
Iteration 9 Groups 7 Changes 13
Iteration 10 Groups 6 Changes 9
Iteration 11 Groups 5 Changes 6
Iteration 12 Groups 4 Changes 3
Iteration 13 Groups 3 Changes 1
Done
```

Animate the cross labeling iterations

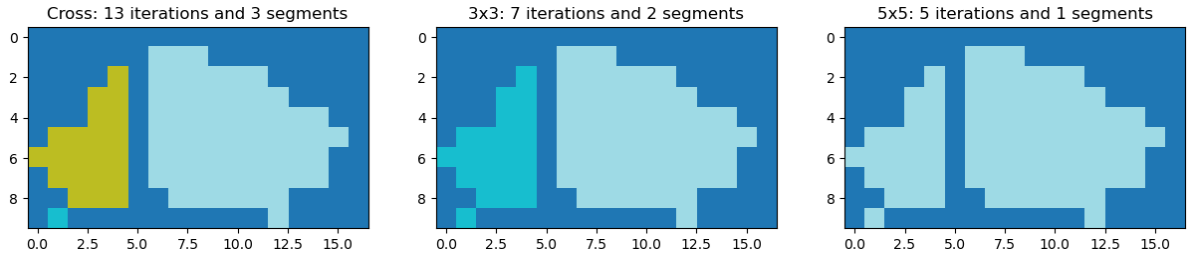
```
fig, c_ax = plt.subplots(1, 1, figsize=(8, 6))


def update_frame(i):
    plt.cla()

    ps.heatmap(img_list1[i],cmap='nipy_spectral',fontsize=8,precision=0,ax=c_ax);
    c_ax.set_title('Iteration #{}, Groups {}'.format(i+1,
                                                    len(np.unique(img_list1[i][img_
 ↪list1[i] > 0].ravel()))))


# write animation frames
anim_code = FuncAnimation(fig,
                          update_frame,
                          frames=len(img_list1)-1,
                          interval=500,
                          repeat_delay=1000).to_html5_video()
plt.close('all')
HTML(anim_code)
```

```
<IPython.core.display.HTML object>
```

0.3.6 Comparing different neighborhoods

Neighborhood size	Iterations	Segments
3x3	9	2
5x5	5	1
cross	14	3

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

cmap='tab20'
ax[0].imshow(img_list1[-1],cmap=cmap)
ax[0].set_title('Cross: {0} iterations and {1} segments'.format(it1[-1]['Iteration'],
 ↪it1[-1]['Groups']))
ax[1].imshow(img_list3[-1],cmap=cmap)
ax[1].set_title('3x3: {0} iterations and {1} segments'.format(it3[-1]['Iteration'],
 ↪it3[-1]['Groups']))
ax[2].imshow(img_list5[-1],cmap=cmap)
ax[2].set_title('5x5: {0} iterations and {1} segments'.format(it5[-1]['Iteration'],
 ↪it5[-1]['Groups']));
```



0.4 Beyond component labeling - what can we measure?

Now all the voxels which are connected have the same label. We can then perform simple metrics like

- counting the number of voxels in each label to estimate volume.
- looking at the change in volume during erosion or dilation to estimate surface area

0.4.1 What we would like to to do

- Count the cells
- Say something about the cells
- Compare the cells in this image to another image

… But where do we start?

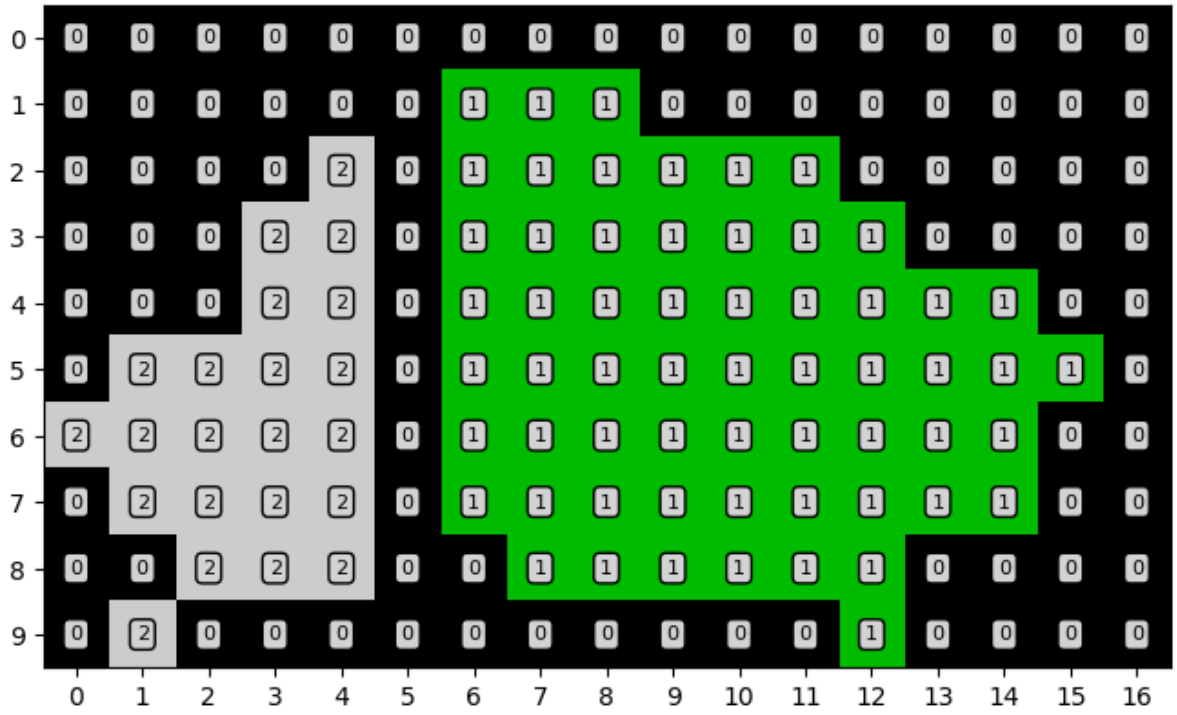
0.4.2 Object position - Center of Volume (COV): With a single object

$$I_{id}(x,y) = \begin{cases} 1, & L(x,y) = id \\ 0, & \text{otherwise} \end{cases}$$

The first step to compute the COV of an item is to label the image to be able to tell which pixels belong to each object. The equation above creates a set of pixels beloning to the item with label *id*.

```
seg_img = imread('figures/aachen_label.png') == 26
seg_img = seg_img[::4, ::4]
seg_img = seg_img[110:130:2, 370:420:3]
seg_img[9, 1] = 1
lab_img = label(seg_img)
```

```
# Visualization
fig, ax = plt.subplots(figsize=[8,6],dpi=100)

ps.heatmap(lab_img,cmap='nipy_spectral',fontsize=8,precision=0,ax=ax);
```



We used our own labeling algorithm in the previous examples. That algorithm is good to demonstrate the principle, but is not so efficient. Therefore, will use the `label` function from SciKit Image from now on.

Define a center

The center is defined by the coordinates of the item along each image axis, i.e., the coordinate pair x,y for a 2D image and x,y,z for a 3D image. Each coordinate component can be computed independently from the other as the axes are orthogonal like:

$$\bar{x} = \frac{1}{N}\sum_{\vec{v}\in I_{id}} \vec{v}\cdot\vec{i}$$

$$\bar{y} = \frac{1}{N}\sum_{\vec{v}\in I_{id}} \vec{v}\cdot\vec{j}$$

$$\bar{z} = \frac{1}{N}\sum_{\vec{v}\in I_{id}} \vec{v}\cdot\vec{k}$$

i.e. the average position of all pixels in each direction.

Center of a labeled item

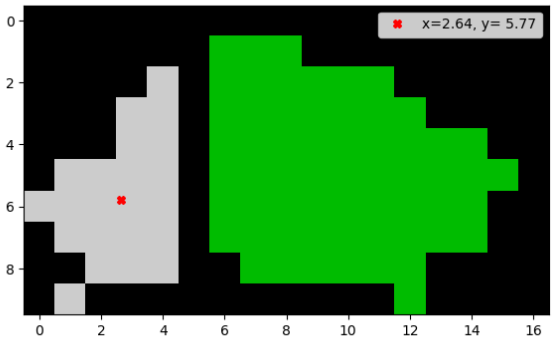
Now, we look into a way to compute the center of an item in the image. The image has been labeled as preparations for the task. There are different ways to solve this task. The one you see here is easier to comprehend and provides a table with each coordinate in I_{id}, but is not the most efficient.

```
x_coord, y_coord = [], []
for x in range(lab_img.shape[0]):
    for y in range(lab_img.shape[1]):
        if lab_img[x, y] == 2:
            x_coord += [x]
            y_coord += [y]

cx = np.mean(x_coord)
cy = np.mean(y_coord)
```

```
fig, ax = plt.subplots(1,2,figsize=[15,6],dpi=100)
items = pd.DataFrame.from_dict({'x': x_coord, 'y': y_coord})
#ps.heatmap(lab_img,cmap='nipy_spectral',fontsize=8,precision=0,ax=ax[1]);
plt.imshow(lab_img,cmap='nipy_spectral');
ax[1].plot(np.mean(y_coord),np.mean(x_coord),'rX',
        label="x={0:0.2f}, y= {1:0.2f}".format(cy , cx))
ax[1].legend()

pd.plotting.table(data=items, ax=ax[0], loc='center')
ax[0].set_title('Coordinates of Item 2'); ax[0].axis('off');
```

Coordinates of Item 2

	x	y
0	2	4
1	3	3
2	3	4
3	4	3
4	4	4
5	5	1
6	5	2
7	5	3
8	5	4
9	6	0
10	6	1
11	6	2
12	6	3
13	6	4
14	7	1
15	7	2
16	7	3
17	7	4
18	8	2
19	8	3
20	8	4
21	9	1



0.4.3 Center of Mass (COM): With a single object

If the gray values are kept (or other meaningful ones are used), this can be seen as a weighted center of volume or center of mass (using I_{gy} to distinguish it from the labels)

Define a center

$$\bar{x} = \frac{1}{\Sigma I_{gy}} \sum_{\vec{v} \in I_{id}} (\vec{v} \cdot \vec{i}) I_{gy}(\vec{v})$$

$$\bar{y} = \frac{1}{\Sigma I_{gy}} \sum_{\vec{v} \in I_{id}} (\vec{v} \cdot \vec{j}) I_{gy}(\vec{v})$$

$$\bar{z} = \frac{1}{\Sigma I_{gy}} \sum_{\vec{v} \in I_{id}} (\vec{v} \cdot \vec{k}) I_{gy}(\vec{v})$$

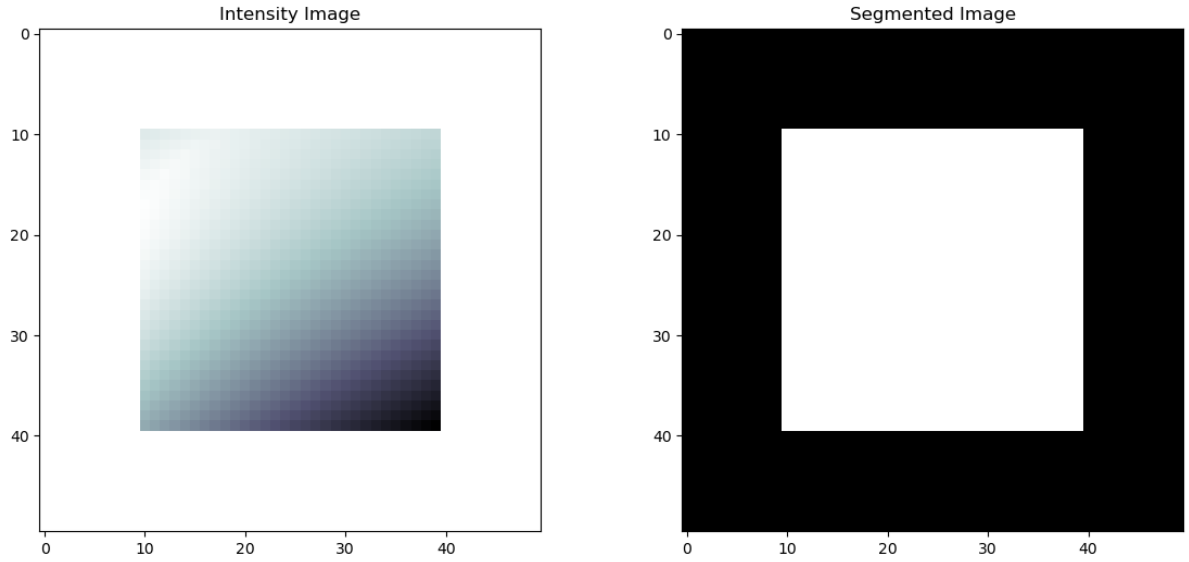
$$\Sigma I_{gy} = \frac{1}{N} \sum_{\vec{v} \in I_{id}} I_{gy}(\vec{v})$$

Demonstrating the center of mass

We need an image for the demonstration

```
xx, yy = np.meshgrid(np.linspace(0, 10, 50),
                     np.linspace(0, 10, 50))
gray_img = 100*(np.abs(xx*yy-7) + np.square(yy-4))+0.25
gray_img *= np.abs(xx-5) < 3
gray_img *= np.abs(yy-5) < 3
gray_img[gray_img > 0] += 5
seg_img = (gray_img > 0).astype(int)
```

```
_, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

ax1.imshow(gray_img,cmap='bone_r')
ax1.set_title('Intensity Image')

ax2.imshow(seg_img,cmap='bone')
ax2.set_title('Segmented Image');
```



Compare COM and COV

Both COV and COM provide a center metric. The difference is that the COV only considers the shape of the object while COM adds the weight of the gray level distribution in the calculation.

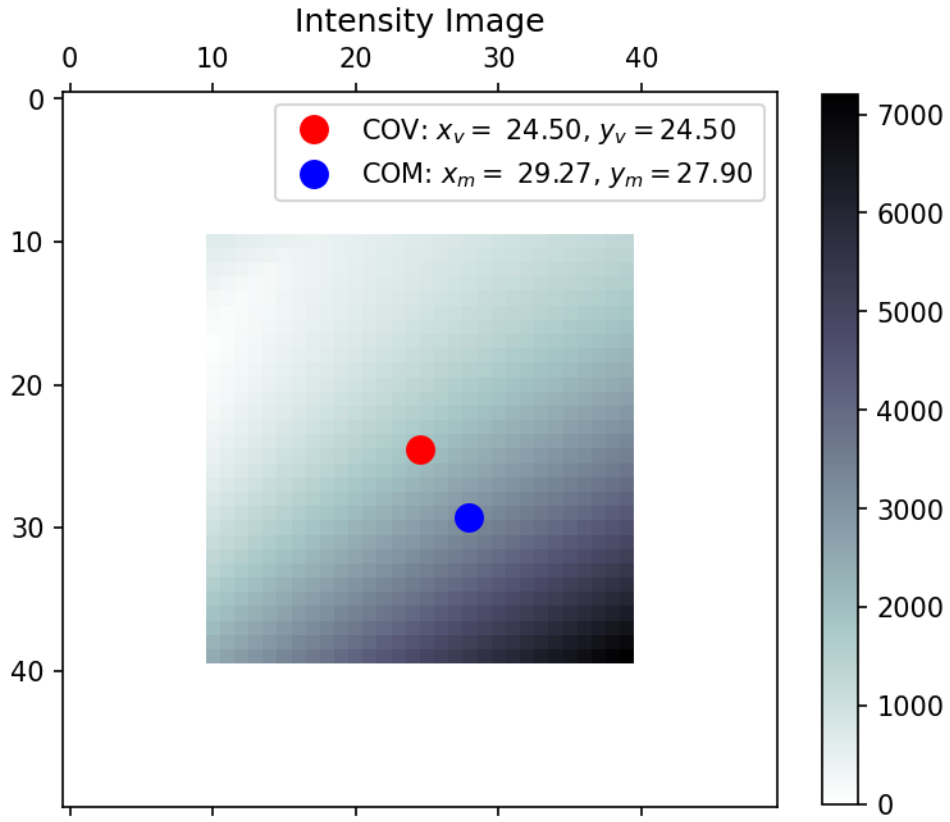
```
# Collect information
x_coord, y_coord, i_val = [], [], []
for x in range(seg_img.shape[0]):
    for y in range(seg_img.shape[1]):
        if seg_img[x, y] == 1:
            x_coord += [x]
            y_coord += [y]
            i_val   += [gray_img[x, y]]

x_coord = np.array(x_coord)
y_coord = np.array(y_coord)
i_val   = np.array(i_val)

cov_x   = np.mean(x_coord)
cov_y   = np.mean(y_coord)
com_x   = np.mean(x_coord*i_val)/i_val.mean()
com_y   = np.mean(y_coord*i_val)/i_val.mean()
```

```
fig, ax1 = plt.subplots(1, 1, figsize=(6, 6), dpi=150)

im=ax1.matshow(gray_img,cmap='bone_r'); fig.colorbar(im,ax=ax1, shrink=0.8)
```

(continues on next page)

(continued from previous page)

```
ax1.set_title('Intensity Image')
ax1.plot([cov_y], [cov_x], 'ro', label='COV: $x_v=$ {0:0.2f}, $y_v=${1:0.2f}'.
 ↪format(cov_x,cov_y), markersize=10)
ax1.plot([com_y], [com_x], 'bo', label='COM: $x_m=$ {0:0.2f}, $y_m=${1:0.2f}'.
 ↪format(com_x,com_y), markersize=10); ax1.legend();
```



0.4.4 Further object metrics

The center tells the position of an object.

We want more! E.g. metrics like:

- Area
- Perimeter length
- Sphericity
- Orientation

… and more

`regionprops` gives us all this!

Regionprops manual page

```
from skimage.measure import regionprops
help(regionprops)
```

```
Help on function regionprops in module skimage.measure._regionprops:

regionprops(label_image, intensity_image=None, cache=True, *, extra_
↪properties=None, spacing=None, offset=None)
    Measure properties of labeled image regions.

    Parameters
    ----------
    label_image : (M, N[, P]) ndarray
        Labeled input image. Labels with value 0 are ignored.

        .. versionchanged:: 0.14.1
            Previously, ``label_image`` was processed by ``numpy.squeeze`` and
            so any number of singleton dimensions was allowed. This resulted in
            inconsistent handling of images with singleton dimensions. To
            recover the old behaviour, use
            ``regionprops(np.squeeze(label_image), ...)``.
    intensity_image : (M, N[, P][, C]) ndarray, optional
        Intensity (i.e., input) image with same size as labeled image, plus
        optionally an extra dimension for multichannel data. Currently,
        this extra channel dimension, if present, must be the last axis.
        Default is None.

        .. versionchanged:: 0.18.0
            The ability to provide an extra dimension for channels was added.
    cache : bool, optional
        Determine whether to cache calculated properties. The computation is
        much faster for cached properties, whereas the memory consumption
        increases.
    extra_properties : Iterable of callables
        Add extra property computation functions that are not included with
        skimage. The name of the property is derived from the function name,
        the dtype is inferred by calling the function on a small sample.
        If the name of an extra property clashes with the name of an existing
        property the extra property will not be visible and a UserWarning is
        issued. A property computation function must take a region mask as its
        first argument. If the property requires an intensity image, it must
        accept the intensity image as the second argument.
    spacing: tuple of float, shape (ndim, )
        The pixel spacing along each axis of the image.
    offset : array-like of int, shape `(label_image.ndim,)`, optional
        Coordinates of the origin ("top-left" corner) of the label image.
        Normally this is ([0, ]0, 0), but it might be different if one wants
        to obtain regionprops of subvolumes within a larger volume.

    Returns
    -------
    properties : list of RegionProperties
        Each item describes one labeled region, and can be accessed using the
        attributes listed below.
```

(continues on next page)

(continued from previous page)

```
    Notes
    -----
    The following properties can be accessed as attributes or keys:

    **area** : float
        Area of the region i.e. number of pixels of the region scaled by pixel-
↪area.
    **area_bbox** : float
        Area of the bounding box i.e. number of pixels of bounding box scaled by␣
↪pixel-area.
    **area_convex** : float
        Area of the convex hull image, which is the smallest convex
        polygon that encloses the region.
    **area_filled** : float
        Area of the region with all the holes filled in.
    **axis_major_length** : float
        The length of the major axis of the ellipse that has the same
        normalized second central moments as the region.
    **axis_minor_length** : float
        The length of the minor axis of the ellipse that has the same
        normalized second central moments as the region.
    **bbox** : tuple
        Bounding box ``(min_row, min_col, max_row, max_col)``.
        Pixels belonging to the bounding box are in the half-open interval
        ``[min_row; max_row)`` and ``[min_col; max_col)``.
    **centroid** : array
        Centroid coordinate tuple ``(row, col)``.
    **centroid_local** : array
        Centroid coordinate tuple ``(row, col)``, relative to region bounding
        box.
    **centroid_weighted** : array
        Centroid coordinate tuple ``(row, col)`` weighted with intensity
        image.
    **centroid_weighted_local** : array
        Centroid coordinate tuple ``(row, col)``, relative to region bounding
        box, weighted with intensity image.
    **coords_scaled** : (N, 2) ndarray
        Coordinate list ``(row, col)``of the region scaled by ``spacing``.
    **coords** : (N, 2) ndarray
        Coordinate list ``(row, col)`` of the region.
    **eccentricity** : float
        Eccentricity of the ellipse that has the same second-moments as the
        region. The eccentricity is the ratio of the focal distance
        (distance between focal points) over the major axis length.
        The value is in the interval [0, 1).
        When it is 0, the ellipse becomes a circle.
    **equivalent_diameter_area** : float
        The diameter of a circle with the same area as the region.
    **euler_number** : int
        Euler characteristic of the set of non-zero pixels.
        Computed as number of connected components subtracted by number of
        holes (input.ndim connectivity). In 3D, number of connected
        components plus number of holes subtracted by number of tunnels.
    **extent** : float
        Ratio of pixels in the region to pixels in the total bounding box.
        Computed as ``area / (rows * cols)``
```

(continues on next page)

(continued from previous page)

```
**feret_diameter_max** : float
    Maximum Feret's diameter computed as the longest distance between
    points around a region's convex hull contour as determined by
    ``find_contours``. [5]_
**image** : (H, J) ndarray
    Sliced binary region image which has the same size as bounding box.
**image_convex** : (H, J) ndarray
    Binary convex hull image which has the same size as bounding box.
**image_filled** : (H, J) ndarray
    Binary region image with filled holes which has the same size as
    bounding box.
**image_intensity** : ndarray
    Image inside region bounding box.
**inertia_tensor** : ndarray
    Inertia tensor of the region for the rotation around its mass.
**inertia_tensor_eigvals** : tuple
    The eigenvalues of the inertia tensor in decreasing order.
**intensity_max** : float
    Value with the greatest intensity in the region.
**intensity_mean** : float
    Value with the mean intensity in the region.
**intensity_min** : float
    Value with the least intensity in the region.
**label** : int
    The label in the labeled input image.
**moments** : (3, 3) ndarray
    Spatial moments up to 3rd order::

        m_ij = sum{ array(row, col) * row^i * col^j }

    where the sum is over the `row`, `col` coordinates of the region.
**moments_central** : (3, 3) ndarray
    Central moments (translation invariant) up to 3rd order::

        mu_ij = sum{ array(row, col) * (row - row_c)^i * (col - col_c)^j }

    where the sum is over the `row`, `col` coordinates of the region,
    and `row_c` and `col_c` are the coordinates of the region's centroid.
**moments_hu** : tuple
    Hu moments (translation, scale and rotation invariant).
**moments_normalized** : (3, 3) ndarray
    Normalized moments (translation and scale invariant) up to 3rd order::

        nu_ij = mu_ij / m_00^[(i+j)/2 + 1]

    where `m_00` is the zeroth spatial moment.
**moments_weighted** : (3, 3) ndarray
    Spatial moments of intensity image up to 3rd order::

        wm_ij = sum{ array(row, col) * row^i * col^j }

    where the sum is over the `row`, `col` coordinates of the region.
**moments_weighted_central** : (3, 3) ndarray
    Central moments (translation invariant) of intensity image up to
    3rd order::
```

(continues on next page)

(continued from previous page)

```
            wmu_ij = sum{ array(row, col) * (row - row_c)^i * (col - col_c)^j }

        where the sum is over the `row`, `col` coordinates of the region,
        and `row_c` and `col_c` are the coordinates of the region's weighted
        centroid.
    **moments_weighted_hu** : tuple
        Hu moments (translation, scale and rotation invariant) of intensity
        image.
    **moments_weighted_normalized** : (3, 3) ndarray
        Normalized moments (translation and scale invariant) of intensity
        image up to 3rd order::

            wnu_ij = wmu_ij / wm_00^[(i+j)/2 + 1]

        where ``wm_00`` is the zeroth spatial moment (intensity-weighted area).
    **num_pixels** : int
        Number of foreground pixels.
    **orientation** : float
        Angle between the 0th axis (rows) and the major
        axis of the ellipse that has the same second moments as the region,
        ranging from `-pi/2` to `pi/2` counter-clockwise.
    **perimeter** : float
        Perimeter of object which approximates the contour as a line
        through the centers of border pixels using a 4-connectivity.
    **perimeter_crofton** : float
        Perimeter of object approximated by the Crofton formula in 4
        directions.
    **slice** : tuple of slices
        A slice to extract the object from the source image.
    **solidity** : float
        Ratio of pixels in the region to pixels of the convex hull image.

    Each region also supports iteration, so that you can do::

      for prop in region:
          print(prop, region[prop])

    See Also
    --------
    label

    References
    ----------
    .. [1] Wilhelm Burger, Mark Burge. Principles of Digital Image Processing:
           Core Algorithms. Springer-Verlag, London, 2009.
    .. [2] B. Jähne. Digital Image Processing. Springer-Verlag,
           Berlin-Heidelberg, 6. edition, 2005.
    .. [3] T. H. Reiss. Recognizing Planar Objects Using Invariant Image
           Features, from Lecture notes in computer science, p. 676. Springer,
           Berlin, 1993.
    .. [4] https://en.wikipedia.org/wiki/Image_moment
    .. [5] W. Pabst, E. Gregorová. Characterization of particles and particle
           systems, pp. 27-28. ICT Prague, 2007.
           https://old.vscht.cz/sil/keramika/Characterization_of_particles/CPPS%20_
↪English%20version_.pdf
```

(continues on next page)

(continued from previous page)

```
    Examples
    --------
    >>> from skimage import data, util
    >>> from skimage.measure import label, regionprops
    >>> img = util.img_as_ubyte(data.coins()) > 110
    >>> label_img = label(img, connectivity=img.ndim)
    >>> props = regionprops(label_img)
    >>> # centroid of first labeled object
    >>> props[0].centroid
    (22.72987986048314, 81.91228523446583)
    >>> # centroid of first labeled object
    >>> props[0]['centroid']
    (22.72987986048314, 81.91228523446583)

    Add custom measurements by passing functions as ``extra_properties``

    >>> from skimage import data, util
    >>> from skimage.measure import label, regionprops
    >>> import numpy as np
    >>> img = util.img_as_ubyte(data.coins()) > 110
    >>> label_img = label(img, connectivity=img.ndim)
    >>> def pixelcount(regionmask):
    ...     return np.sum(regionmask)
    >>> props = regionprops(label_img, extra_properties=(pixelcount,))
    >>> props[0].pixelcount
    7741
    >>> props[1]['pixelcount']
    42
```

Let's try regionprops on our image

```
from skimage.measure import regionprops_table
safe_props = [
    'label', 'area', 'bbox_area', 'centroid', 'convex_area','eccentricity',
 ↪'equivalent_diameter', 'euler_number',
    'extent', 'feret_diameter_max', 'filled_area', 'inertia_tensor', 'inertia_tensor_
 ↪eigvals',
    'major_axis_length', 'minor_axis_length', 'moments_hu', 'orientation', 'perimeter
 ↪', 'perimeter_crofton', 'solidity']

all_regs = regionprops_table(seg_img, intensity_image=gray_img, properties=safe_props)
attr_df=pd.DataFrame.from_dict(all_regs, orient="index")

attr_df
```

```
                                0
label                    1.000000
area                   900.000000
bbox_area              900.000000
centroid-0              24.500000
centroid-1              24.500000
convex_area            900.000000
eccentricity             0.000000
```

(continues on next page)

(continued from previous page)

```
equivalent_diameter        33.851375
euler_number                1.000000
extent                      1.000000
feret_diameter_max         41.725292
filled_area               900.000000
inertia_tensor-0-0         74.916667
inertia_tensor-0-1         -0.000000
inertia_tensor-1-0         -0.000000
inertia_tensor-1-1         74.916667
inertia_tensor_eigvals-0   74.916667
inertia_tensor_eigvals-1   74.916667
major_axis_length          34.621766
minor_axis_length          34.621766
moments_hu-0                0.166481
moments_hu-1                0.000000
moments_hu-2                0.000000
moments_hu-3                0.000000
moments_hu-4                0.000000
moments_hu-5                0.000000
moments_hu-6                0.000000
orientation                -0.785398
perimeter                 116.000000
perimeter_crofton         112.656413
solidity                    1.000000
```

Lots of information

We can tell a lot about each object now, but…

- Too abstract
- Too specific

Ask biologists in the class if they ever asked

- "How long is a cell in the x direction?"
- "how about y?"

When we know which properties we need

We saw before that the `regionprops` function provides a long list of properties. Most of them are often irrelevant for our measurements. We can, therefore, specify which properties we really want to measure. The next example uses a variation `regionprops_table` of the regionprops function which is better to use if we want to put the measurements in a dataframe. This function requires a list of properties which makes it less convenient if we want all properties.

Let's say we want the properties `area`,`perimeter`, and `centroid`

```
from skimage.measure import regionprops_table
attr = regionprops_table(seg_img, intensity_image=gray_img, properties=['area',
 ↪'perimeter', 'centroid'])

attr_df=pd.DataFrame.from_dict(attr, orient="index")
attr_df
```

```
                0
area        900.0
perimeter   116.0
centroid-0   24.5
centroid-1   24.5
```

0.4.5 Extents: With a single object

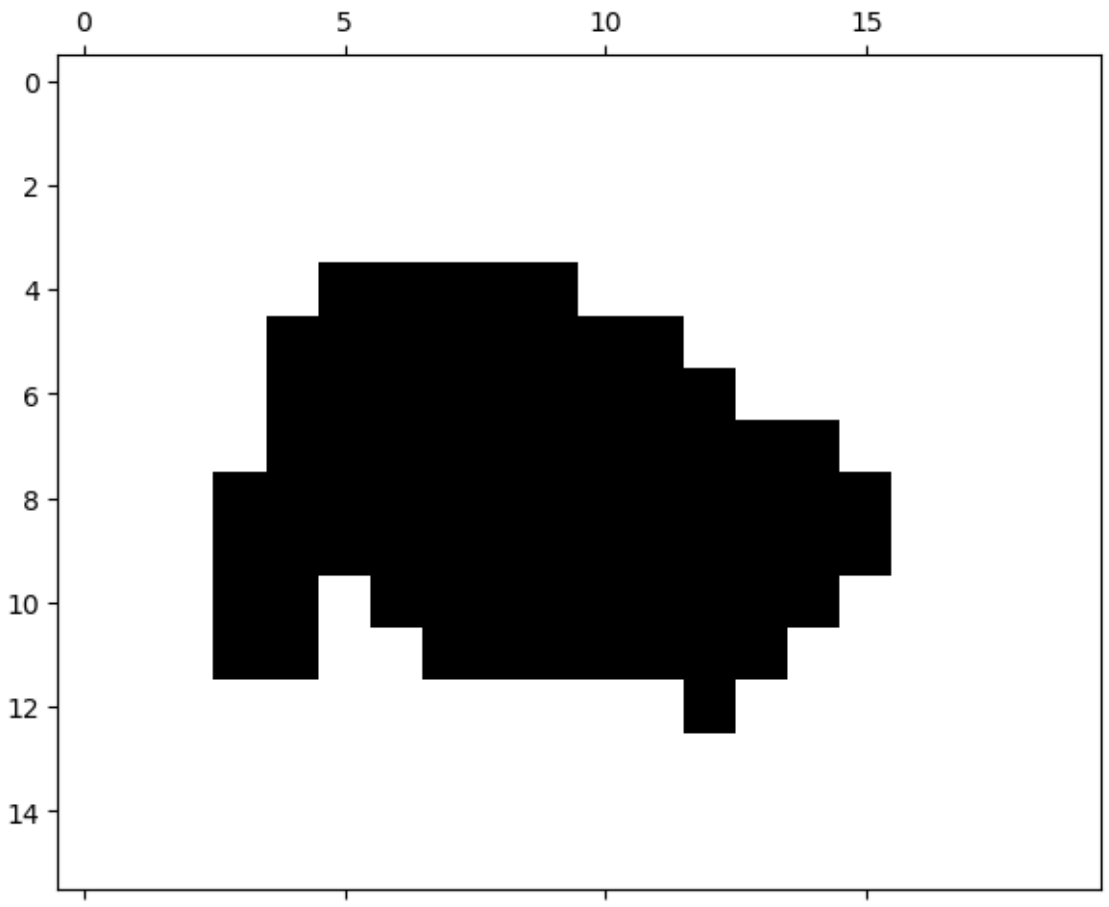
Exents or caliper lenghts are the size of the object in a given direction. Since the coordinates of our image our x and y the extents are calculated in these directions

Define extents as the minimum and maximum values along the projection of the shape in each direction $\text{Ext}_x = \left\{\forall \vec{v} \in I_{id} : max(\vec{v} \cdot \vec{i}) - min(\vec{v} \cdot \vec{i})\right\} \text{Ext}_y = \left\{\forall \vec{v} \in I_{id} : max(\vec{v} \cdot \vec{j}) - min(\vec{v} \cdot \vec{j})\right\} \text{Ext}_z = \left\{\forall \vec{v} \in I_{id} : max(\vec{v} \cdot \vec{k}) - min(\vec{v} \cdot \vec{k})\right\}$

Where is this information useful?

Let's look at a car item

```
seg_img = imread('figures/aachen_label.png') == 26
seg_img = seg_img[::4, ::4]
seg_img = seg_img[110:130:2, 378:420:3] > 0
seg_img = np.pad(seg_img, 3, mode='constant')
_, (ax1) = plt.subplots(1, 1,
                        figsize=(7, 7),
                        dpi=100)
ax1.matshow(seg_img,
            cmap='bone_r');
```



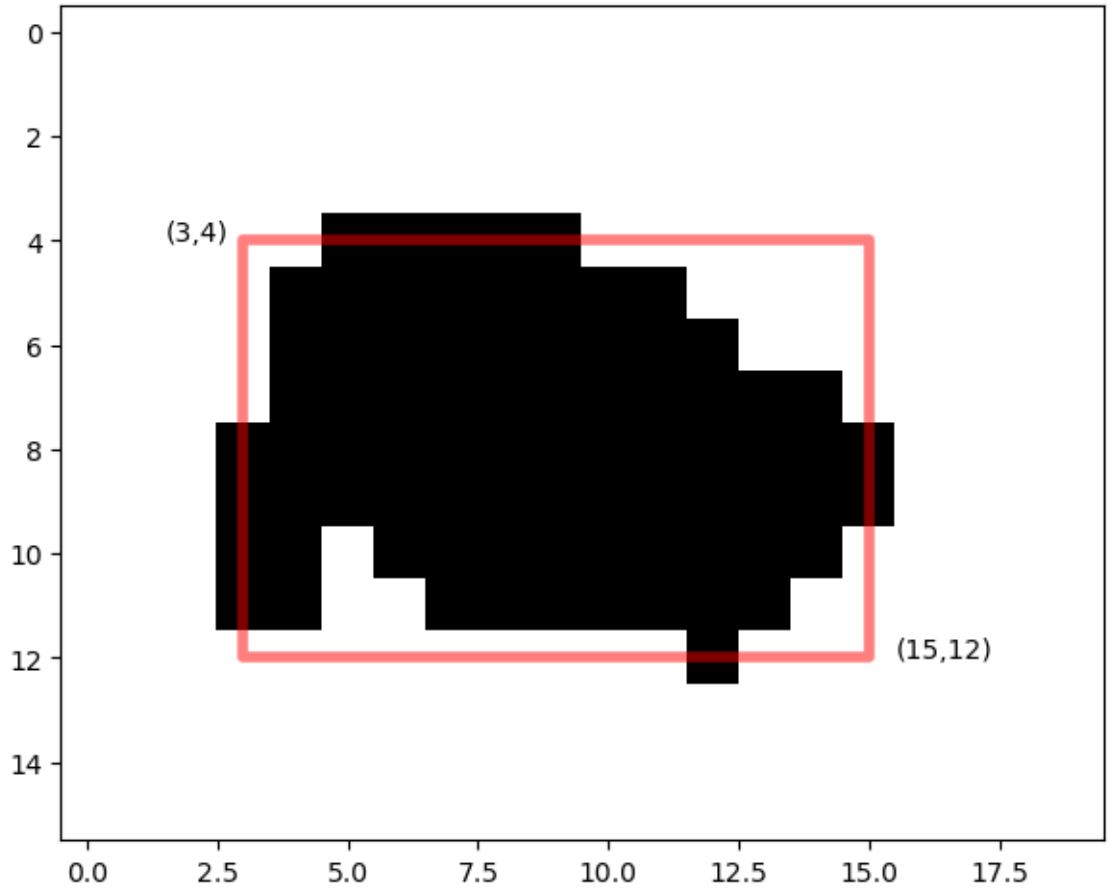
Finding a bounding box

```
# Find all coordinates in an object
x_coord, y_coord = [], []
for x in range(seg_img.shape[0]):
    for y in range(seg_img.shape[1]):
        if seg_img[x, y] == 1:
            x_coord += [x]
            y_coord += [y]

xmin = np.min(x_coord)
xmax = np.max(x_coord)
ymin = np.min(y_coord)
ymax = np.max(y_coord)
print('X -> ', 'Min:', xmin,'Max:', xmax)
print('Y -> ', 'Min:', ymin,'Max:', ymax)
```

```
X ->  Min: 4 Max: 12
Y ->  Min: 3 Max: 15
```

Draw the box

```
_, (ax1) = plt.subplots(1, 1, figsize=(7, 7), dpi=100)

ax1.imshow(seg_img, cmap='bone_r')

xw = (xmax-xmin)
yw = (ymax-ymin)

c_bbox = [Rectangle(xy=(ymin, xmin),
                    width=yw,
                    height=xw
                    )]
c_bb_patch = PatchCollection(c_bbox,
                             facecolor='none',
                             edgecolor='red',
                             linewidth=4,
                             alpha=0.5)
ax1.add_collection(c_bb_patch);
ax1.text(1.5,4,'({0},{1})'.format(ymin,xmin))
ax1.text(15.5,12,'({0},{1})'.format(ymax,xmax));
```

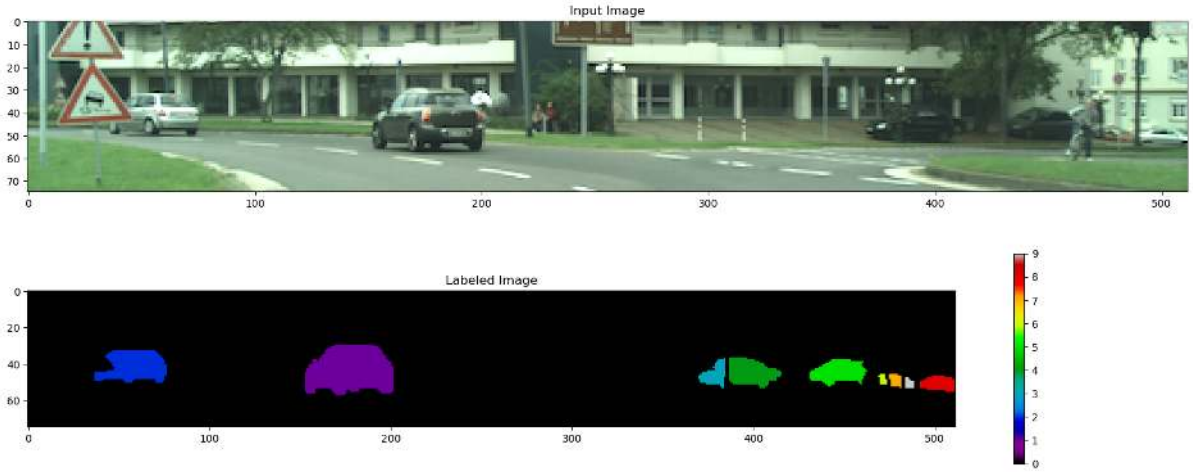


0.5 Using regionprops on real images

So how can we begin to apply the tools we have developed?

We take the original car scene from before.

```
car_img = np.clip(imread('figures/aachen_img.png')
                  [75:150]*2.0, 0, 255).astype(np.uint8)
lab_img = label(imread('figures/aachen_label.png')[::4, ::4] == 26)[75:150]
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(20, 8))
ax1.imshow(car_img)
ax1.set_title('Input Image');

plt.colorbar(ax2.imshow(lab_img, cmap='nipy_spectral',interpolation='None'))
ax2.set_title('Labeled Image');
```



0.5.1 Shape Analysis

We can perform shape analysis and calculate basic shape parameters for each object

```
regs = regionprops_table(lab_img,properties=['label','centroid','bbox'])
attr_df=pd.DataFrame.from_dict(regs, orient="index")
attr_df
```

```
                    0           1           2           3           4  \
label        1.000000    2.000000    3.000000    4.000000    5.000000
centroid-0  43.645280   41.442991   45.871429   44.862573   44.526316
centroid-1 177.822181   58.760748  378.664286  398.669591  446.913313
bbox-0      30.000000   33.000000   37.000000   37.000000   38.000000
bbox-1     153.000000   37.000000  370.000000  387.000000  431.000000
bbox-2      58.000000   51.000000   54.000000   54.000000   53.000000
bbox-3     202.000000   77.000000  385.000000  416.000000  463.000000

                    5           6           7       8
label        6.000000    7.000000    8.000000    9.00
centroid-0  48.636364   49.090909   50.992537   50.92
centroid-1 471.181818  478.227273  501.820896  485.76
```

(continues on next page)

(continued from previous page)

```
bbox-0       46.000000   46.000000   47.000000   48.00
bbox-1      469.000000  475.000000  492.000000  484.00
bbox-2       52.000000   54.000000   56.000000   54.00
bbox-3      474.000000  482.000000  511.000000  489.00
```

Showing the properties in the image

```
fig, ax1 = plt.subplots(1, 1, figsize=(12, 6), dpi=100)
ax1.imshow(car_img)

bbox_list = []
for idx in range(len(regs['centroid-0'])):
    ax1.plot(regs['centroid-1'][idx], regs['centroid-0'][idx], 'o', markersize=5)
    bbox_list += [Rectangle(xy=(regs['bbox-1'][idx],
                                regs['bbox-0'][idx]),
                                width  =regs['bbox-3'][idx]-regs['bbox-1'][idx],
                                height =regs['bbox-2'][idx]-regs['bbox-0'][idx]
                            )]
c_bb_patch = PatchCollection(bbox_list,
                             facecolor='none',
                             edgecolor='red',
                             linewidth=4,
                             alpha=0.5)
ax1.add_collection(c_bb_patch);
```

0.5.2 Statistics

We can then generate a table full of these basic parameters for each object. In this case, we add color as an additional description

```
def ed_img(in_img):
    # shrink an image to a few pixels
    cur_img = in_img.copy()
    while cur_img.max() > 0:
        last_img = cur_img
        cur_img = erosion(cur_img, disk(1))
    return last_img
```

```
from matplotlib import colors as mcolors
from sklearn.neighbors import KNeighborsClassifier

# Get color names and RGB values from matplotlib
color_names = list(mcolors.CSS4_COLORS.keys())
```

(continues on next page)

(continued from previous page)

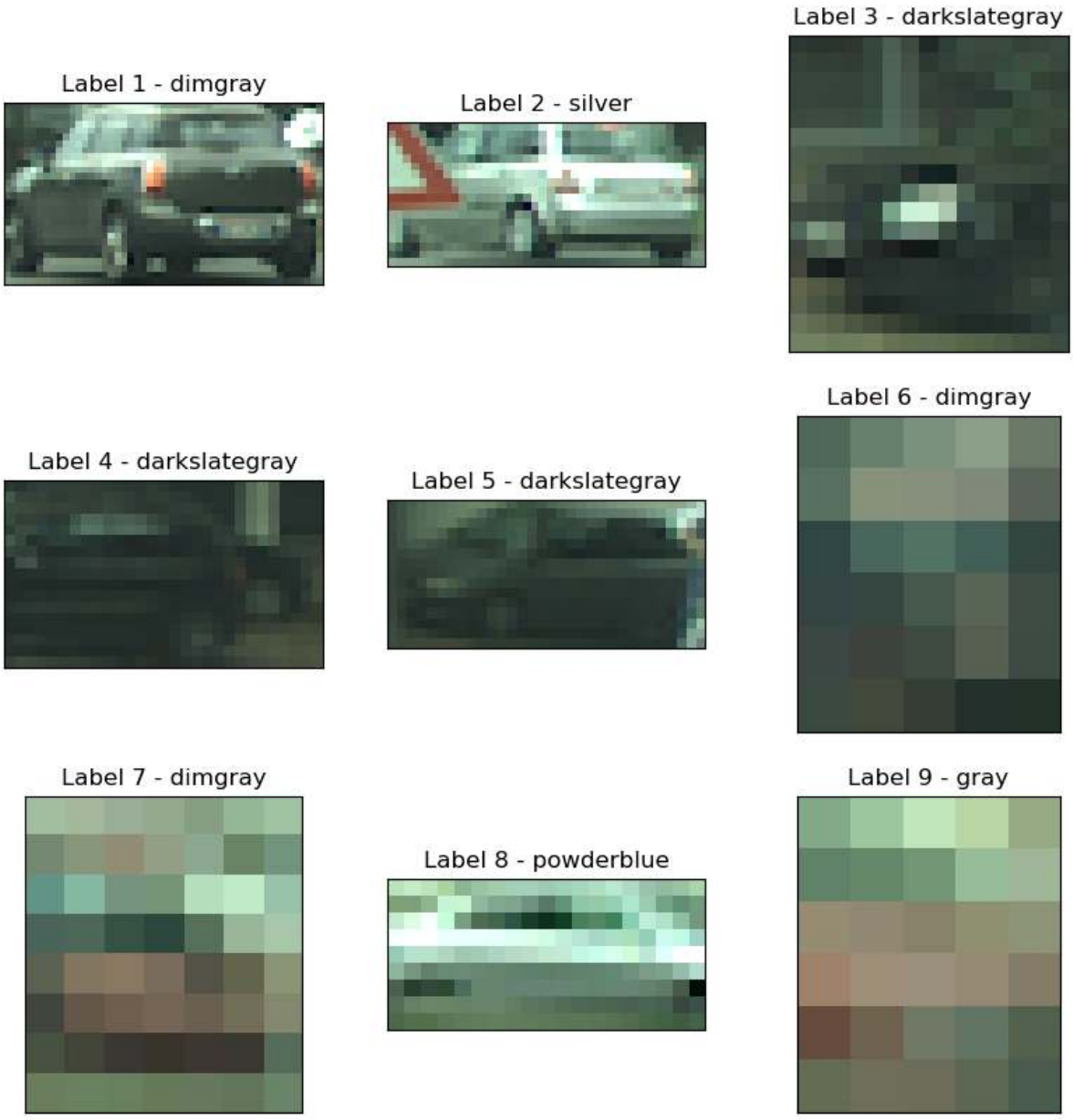
```
color_rgbs = [mcolors.to_rgb(mcolors.CSS4_COLORS[name]) for name in color_names]
color_rgbs = np.array(color_rgbs) * 255  # Convert to 0-255 RGB range

# Fit 1-NN classifier
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(color_rgbs, color_names)

#
reg_df = pd.DataFrame.from_dict(regs)
reg_df['color'] = reg_df['label'].apply(lambda c_reg: knn.predict(np.mean(car_img[ed_
 ↪img(lab_img == c_reg)], 0)[:3].reshape((1, -1)))[0])
reg_df
```

	label	centroid-0	centroid-1	bbox-0	bbox-1	bbox-2	bbox-3	\
0	1	43.645280	177.822181	30	153	58	202	
1	2	41.442991	58.760748	33	37	51	77	
2	3	45.871429	378.664286	37	370	54	385	
3	4	44.862573	398.669591	37	387	54	416	
4	5	44.526316	446.913313	38	431	53	463	
5	6	48.636364	471.181818	46	469	52	474	
6	7	49.090909	478.227273	46	475	54	482	
7	8	50.992537	501.820896	47	492	56	511	
8	9	50.920000	485.760000	48	484	54	489	

	color
0	dimgray
1	silver
2	darkslategray
3	darkslategray
4	darkslategray
5	dimgray
6	dimgray
7	powderblue
8	gray

Does it make sense?

```
fig, m_axs = plt.subplots(np.floor(len(regs['centroid-0'])/3).astype(int),3,␣
 ↪figsize=(10,10))
for idx,c_ax in enumerate(m_axs.ravel()):
    c_ax.imshow(car_img[regs['bbox-0'][idx]:regs['bbox-2'][idx],regs['bbox-1
 ↪'][idx]:regs['bbox-3'][idx]])
    c_ax.set(xticks=[],yticks=[],title='Label {0} - {1}'.format(regs['label'][idx],
 ↪reg_df['color'][idx]))
```



0.6 Object anisotropy

0.6.1 Anisotropy: What is it?

By definition (New Oxford American): **varying in magnitude according to the direction of measurement.**

It allows us to define metrics in respect to one another and thereby characterize shape.

- Is it:
  - tall and skinny,
  - short and fat,

- or perfectly round?

0.6.2 A very vague definition

It can be mathematically characterized in many different very much unequal ways (in all cases 0 represents a sphere)

$$A_{iso1} = \frac{\text{Longest Side}}{\text{Shortest Side}} - 1$$

$$A_{iso2} = \frac{\text{Longest Side} - \text{Shortest Side}}{\text{Longest Side}}

 A_{iso3} = \frac{\text{Longest Side}}{\text{Average Side Length}} - 1

 A_{iso4} = \frac{\text{Longest Side} - \text{Shortest Side}}{\text{Average Side Length}}

 \cdots \rightarrow \text{ad nauseum}$$

```
from collections import defaultdict
from skimage.measure import regionprops
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Let's define some of these metrics

```
xx, yy = np.meshgrid(np.linspace(-5, 5, 100),
                     np.linspace(-5, 5, 100))


def side_len(c_reg): return sorted(
    [c_reg.bbox[3]-c_reg.bbox[1], c_reg.bbox[2]-c_reg.bbox[0]])


aiso_funcs = [lambda x:  side_len(x)[-1]/side_len(x)[0]-1,
              lambda x: (side_len(x)[-1]-side_len(x)[0])/side_len(x)[-1],
              lambda x:  side_len(x)[-1]/np.mean(side_len(x))-1,
              lambda x: (side_len(x)[-1]-side_len(x)[0])/np.mean(side_len(x))]


def ell_func(a, b): return np.sqrt(np.square(xx/a)+np.square(yy/b)) <= 1
```

How does the anisotropy metrics respond?

In this demonstration, we look into how the four different anisotropy metrics respond to different radius ratios of an ellipse. The ellipse is nicely aligned with the x- and y- axes, therefore we can use the bounding box to identify the side lengths as diameters in the two directions. These side lengths will be used to compute the aniotropy with our four metrics.

Much of the code below is for the animated display.

```
ab_list = [(2, 2), (2, 3), (2, 4), (2, 5), (1.5, 5),
           (1, 5), (0.5, 5), (0.1, 5),  (0.05, 5)]
func_pts = defaultdict(list)

fig, m_axs = plt.subplots(2, 3, figsize=(9, 6), dpi=100)
```

(continues on next page)

(continued from previous page)

```
def update_frame(i):
    plt.cla()
    a, b     = ab_list[i]
    c_img    = ell_func(a, b)
    reg_info = regionprops(c_img.astype(int))[0]
    m_axs[0, 0].imshow(c_img, cmap='gist_earth')
    m_axs[0, 0].set_title('Shape #{}'.format(i+1))
    m_axs2=[m_axs[0,1],m_axs[0,2],m_axs[1,1],m_axs[1,2]]

    for j, (c_func, c_ax) in enumerate(zip(aiso_funcs, m_axs2), 1):
        func_pts[j] += [c_func(reg_info)]
        c_ax.plot(func_pts[j], 'r-')
        c_ax.set_title('Anisotropy #{}'.format(j))
        c_ax.set_ylim(-.1, 3)
    m_axs[1,0].axis('off')


# write animation frames
anim_code = FuncAnimation(fig,
                          update_frame,
                          frames=len(ab_list)-1,
                          interval=500,
                          repeat_delay=2000).to_html5_video()
plt.close('all')
HTML(anim_code)
```

```
<IPython.core.display.HTML object>
```

0.7 Statistical tools

0.7.1 Useful Statistical Tools

While many of the topics covered in

- Linear Algebra
- and Statistics courses

might not seem very applicable to real problems at first glance.

at least a few of them come in handy for dealing distributions of pixels

(they will only be briefly covered, for more detailed review look at some of the suggested material)

0.7.2 Principal Component Analysis - PCA

- Similar to K-Means insofar as we start with a series of points in a vector space and want to condense the information.

With PCA

- doesn't search for distinct groups,
- we find a linear combination of components which best explain the variance in the system.

To read Principal component analysis: a review and recent developments

0.7.3 PCA definition

1. Compute the covariance or correlation matrix from a set of images $\$C = cov(X_1, ... X_N)\$$
2. Make an Eigen value or Singular Value Decomposition $\$C = Q\Lambda Q^{-1}\$$
3. Use
    - singular values, Λ, to measure the importance of each eigen value
    - eigen vectors, Q, to transform the data

PCA explains structure by:

- Rotating to directions that show the most variation. Q is a rotation matrix.
- Revealing latent patterns, redundancy, and clusters. Great λ represent data.
- Reducing noise by discarding low-variance directions. Low λ low data content.
- Providing a new coordinate system aligned with your data's geometry

0.7.4 PCA on spectroscopy

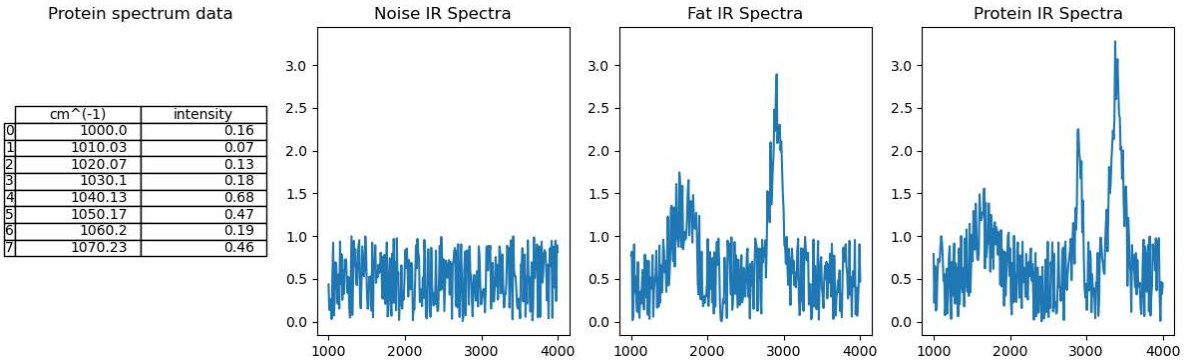
As an example we will use a very simple (simulated) example from spectroscopy:

```
cm_dm = np.linspace(1000, 4000, 300)

# helper functions to build our test data
def peak(cent, wid, h): return h/(wid*np.sqrt(2*np.pi)) * \
    np.exp(-np.square((cm_dm-cent)/wid))

def peaks(plist): return np.sum(np.stack(
    [peak(cent, wid, h) for cent, wid, h in plist], 0), 0)+np.random.uniform(0, 1,␣
 ↪size=cm_dm.shape)

# Define material spectra
fat_curve     = [(2900, 100, 500), (1680, 200, 400)]
protein_curve = [(2900, 50, 200), (3400, 100, 600), (1680, 200, 300)]
noise_curve   = [(3000, 50, 1)]

# Plotting
fig, ax = plt.subplots(1, 4, figsize=(15, 4))
ax[2].plot(cm_dm, peaks(fat_curve));     ax[2].set_title('Fat IR Spectra')
ax[3].plot(cm_dm, peaks(protein_curve)); ax[3].set_title('Protein IR Spectra')
ax[1].plot(cm_dm, peaks(noise_curve));   ax[1].set_title('Noise IR Spectra')
ax[1].set_ylim(ax[3].get_ylim())
```

(continues on next page)

(continued from previous page)

```
ax[2].set_ylim(ax[3].get_ylim())
data=pd.DataFrame({'cm^(-1)': cm_dm, 'intensity': peaks(protein_curve)}).head(8)
pd.plotting.table(data=data.round(decimals=2), ax=ax[0], loc='center'); ax[0].axis(
↪'off'); ax[0].set_title('Protein spectrum data');
```



	cm^(-1)	intensity
0	1000.0	0.16
1	1010.03	0.07
2	1020.07	0.13
3	1030.1	0.18
4	1040.13	0.68
5	1050.17	0.47
6	1060.2	0.19
7	1070.23	0.46

0.7.5 Test Dataset of a number of curves

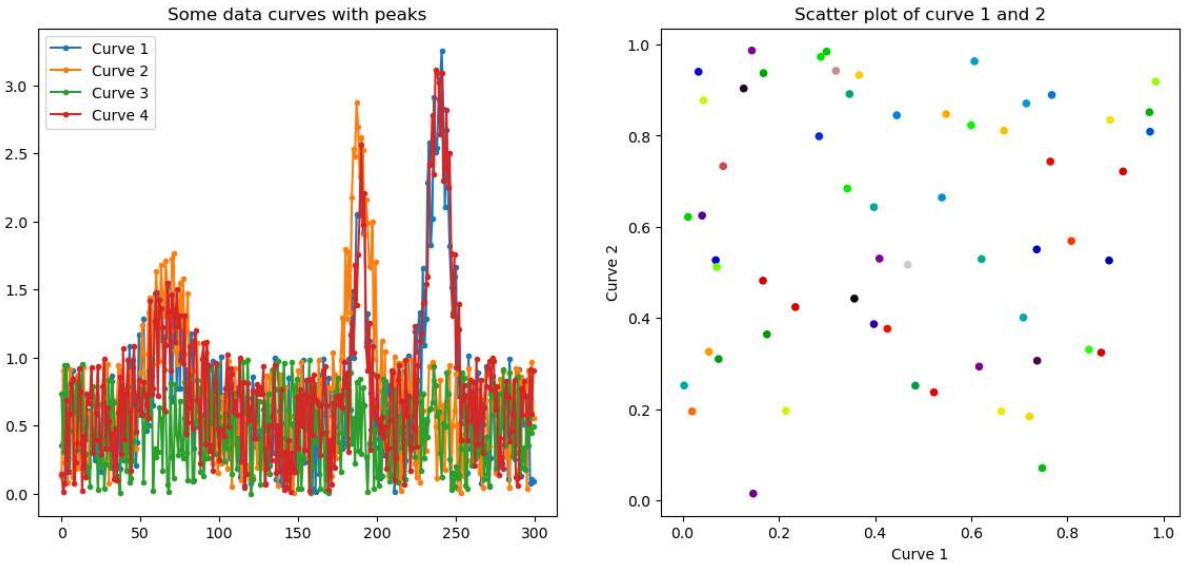
We want to sort cells or samples into groups of being

- more fat like
- or more protein like.

We generate 20 noisy spectra for each type

How can we analyze this data without specifically looking for peaks or building models?

```
test_data = np.stack([peaks(c_curve) for _ in range(20)
                      for c_curve in [protein_curve, fat_curve, noise_curve]], 0)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

ax1.plot(test_data[:4].T, '.-')
ax1.legend(['Curve 1', 'Curve 2', 'Curve 3', 'Curve 4'])
ax1.set_title('Some data curves with peaks')
ax2.scatter(test_data[:, 0], test_data[:,1], c=range(test_data.shape[0]),
            s=20, cmap='nipy_spectral')
ax2.set_title('Scatter plot of curve 1 and 2'); ax2.set_xlabel('Curve 1'); ax2.set_
↪ylabel('Curve 2');
```



We see some peaks, but it is hard to tell one type from the other.

Fit the data with PCA

We expect high correlation within three groups among the 60 spectra. Let's make a PCA analysis with some more components to see the effect of the noise in the data. We select five components for our PCA. PCA is a very common way for preanalysis and data reduction that there is already a function in SciKit Learn.

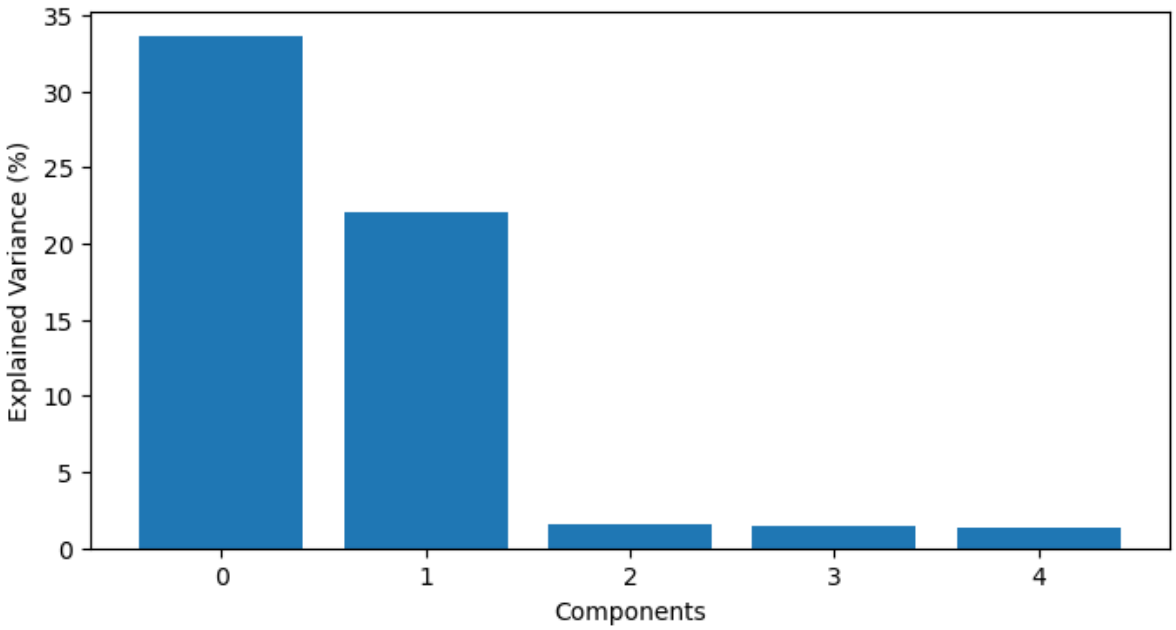
```
from sklearn.decomposition import PCA
pca_tool = PCA(5)
pca_tool.fit(test_data)
```

```
PCA(n_components=5)
```

How important is each component?

Let's first look at the eigenvalues of the PCA. Here, we can already tell that most of the signal energy resides in the first two components, i.e. there is great redundancy in the data. This is expected given the nature of the data with two spectra containing relevant protein information while the last only contains noise.

```
fig, ax1 = plt.subplots(1, 1, figsize=(8, 4), dpi=100)
ax1.bar(x=range(pca_tool.explained_variance_ratio_.shape[0]),
        height=100*pca_tool.explained_variance_ratio_)
ax1.set_xlabel('Components')
ax1.set_ylabel('Explained Variance (%)');
```



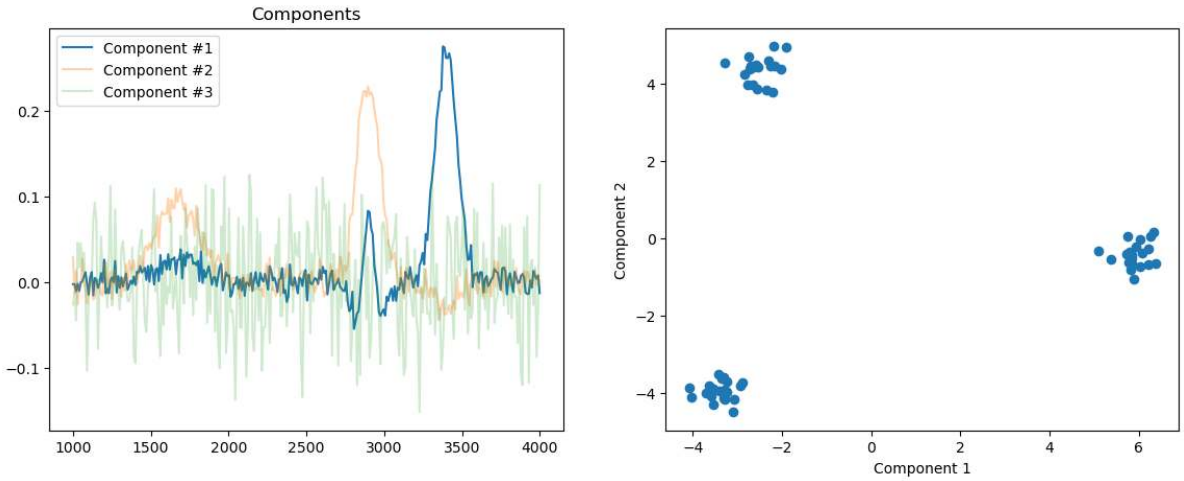
Plot principal components

The next step is to look at the eigen vectors of the PCA which tells how to transform the signals to obtain the greatest separation between the classes.

The eigen vectors of the first three eigen values confirm that the first two components provide relevant information about the peaks while the third essentially only show noise.

```
score_matrix = pca_tool.transform(test_data)
```

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5),dpi=100)
ax1.plot(cm_dm, pca_tool.components_[0, :], label='Component #1')
ax1.plot(cm_dm, pca_tool.components_[
         1, :], label='Component #2', alpha=pca_tool.explained_variance_ratio_[0])
ax1.plot(cm_dm, pca_tool.components_[
         2, :], label='Component #3', alpha=pca_tool.explained_variance_ratio_[1])
ax1.legend(), ax1.set_title('Components')
ax2.scatter(score_matrix[:, 0],
            score_matrix[:, 1])
ax2.set_xlabel('Component 1')
ax2.set_ylabel('Component 2');
```



Now, using components 1 and 2 to transform the specta we get a nice separation between the signals when we look at the scatter plot. This can now be used to classify the signals into their expected three categories using for example k-means clustering.

PCA - Common Pitfalls

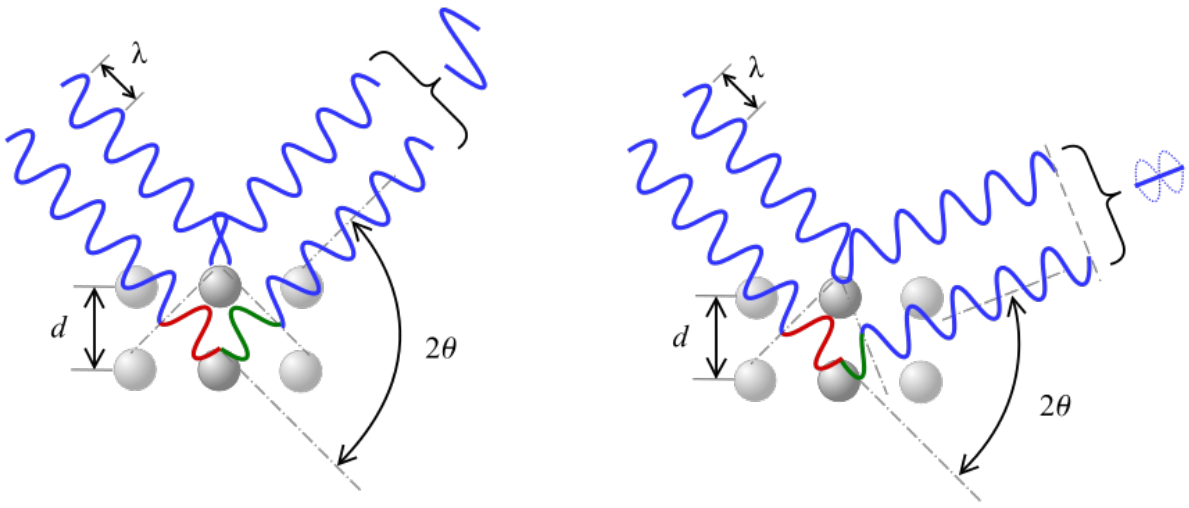
Mistake	Why it's a problem
Not standardizing the data	PCA is sensitive to scale, always normalize first!
Over-interpreting low-variance PCs	They may contain noise
Misreading sign of loadings	Sign is arbitrary; focus on relative magnitude

0.8 PCA in materials science

Explore crystal textures in metals.

0.8.1 Bragg edge imaging - the imaging technique

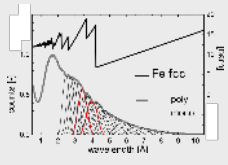
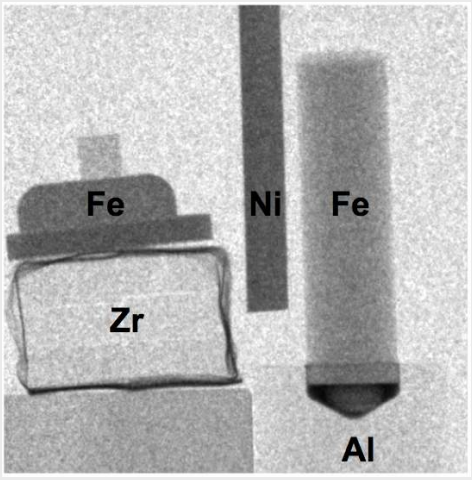
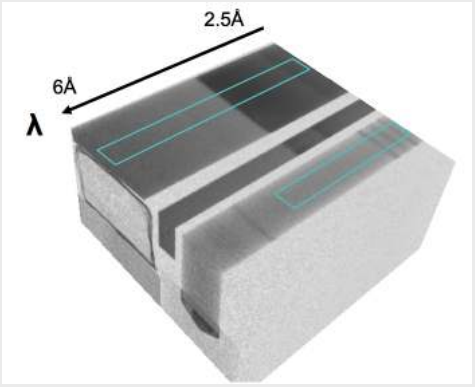
- Wavelength resolved neutron imaging provides spectral images
- Crystaline properties can be explored in the spectra
- Bragg edges appear when the Bragg criterion is fulfilled $2d \sin\theta = n\lambda$



Neutrons in the thermal and cold spectrum have wavelengths in the same scale as the lattice distances in many metals. Neutrons can be used to probe the crystal structure using diffraction thanks to this relation.

The diffraction can cause constructive and destructive interference depending on the match between wavelength, observation angle and lattice distances. The imaging method probes the sample in transmission at different wavelengths and each material will show a characteristic spectral transmission pattern per pixel.

0.8.2 Bragg edge imaging example

Samples	Spectrum	Wavelength scan
		

Images courtesy of S. Peetermans

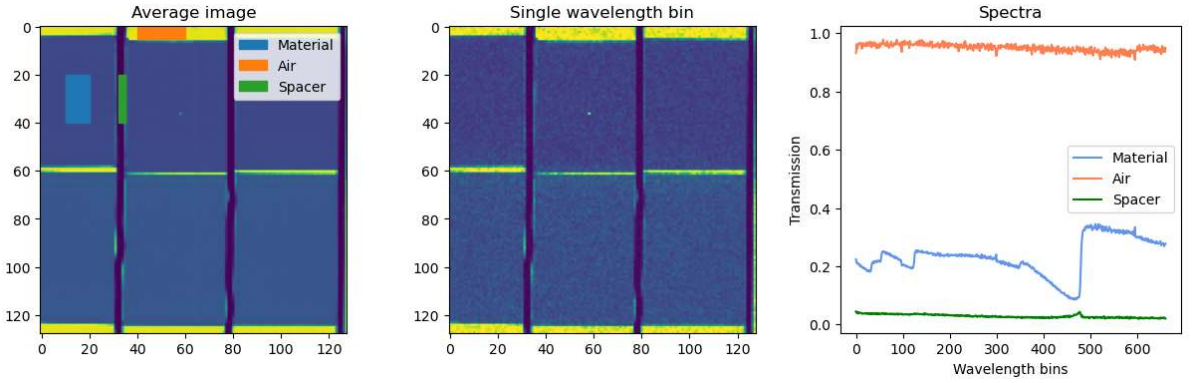
The iron power spectrum is shown together with the neutron source spectrum. The sharp edges in the spectrum are caused at the wavelengths where the Bragg condition is fulfilled. A collection of different materials was tested in this example. The Fe pieces are of particular interest as one sample is poly crystaline while the other is a single crystal.

0.8.3 Bragg edge imaging and PCA

The data

The images in this example come from a neutron wavelength scan of six steel samples produced by additive manufacturing (3D printed). The spectra are supposed to show how the crystal structure change depending on printing method and further treatment.

```
data=np.load('data/tofdata.npy')
fig,ax=plt.subplots(1,3,figsize=(15,4),dpi=100)
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']

ax[0].imshow(data.mean(axis=2), cmap='viridis'), ax[0].set_title('Average image');
ax[0].add_patch(Rectangle(xy=(10, 20), width =10, height = 20, facecolor=colors[0],␣
 ↪edgecolor=colors[0], label = "Material"))
ax[0].add_patch(Rectangle(xy=(40, 0),  width =20, height = 5,  facecolor=colors[1],␣
 ↪edgecolor=colors[1], label="Air"))
ax[0].add_patch(Rectangle(xy=(32, 20), width =3,  height = 20, facecolor=colors[2],␣
 ↪edgecolor=colors[2],label="Spacer"))
ax[0].legend()

ax[1].imshow(data[:,:,100], cmap='viridis'), ax[1].set_title('Single wavelength bin');
ax[2].plot(data[20:40,10:20].mean(axis=0).mean(axis=0), color='Cornflowerblue', label␣
 ↪= 'Material');
ax[2].plot(data[0:5,40:60].mean(axis=0).mean(axis=0),   color='coral',          label␣
 ↪= 'Air');
ax[2].plot(data[20:40,32:35].mean(axis=0).mean(axis=0), color='green',          label␣
 ↪= 'Spacer');
ax[2].set_title('Spectra'); ax[2].set_xlabel('Wavelength bins'); ax[2].set_ylabel(
 ↪'Transmission'); ax[2].legend();
```



You can see in the plots that there is only diffraction response in the steel area. The other two areas are not affected by the change in wavelength only by the attenuation coefficient.

Prepare the data for analysis

1. Rearrange the images into 1D arrays $M \times N \times T \rightarrow M \cdot N \times T$

The first preparation step is to reshape the images from 2D to 1D and thus giving a 1D array per spectral bin. This is needed to be able to compute the covariance matrix.

```
fdata=data.reshape(data.shape[0]*data.shape[1],data.shape[2])
```

2. Compute mean and standard deviation

```
m  = np.reshape(fdata.mean(axis=0),(1,fdata.shape[1]))
mm = np.ones((fdata.shape[0],1))*m
s  = np.reshape(fdata.std(axis=0),(1,fdata.shape[1]))
ss = np.ones((fdata.shape[0],1))*m
```

3. Normalize data

```
mfdata=(fdata-mm)/ss
```

Run the PCA

1. Initialize PCA with 5 components

```
pca_tool = PCA(5)
```

2. Fit data with PCA

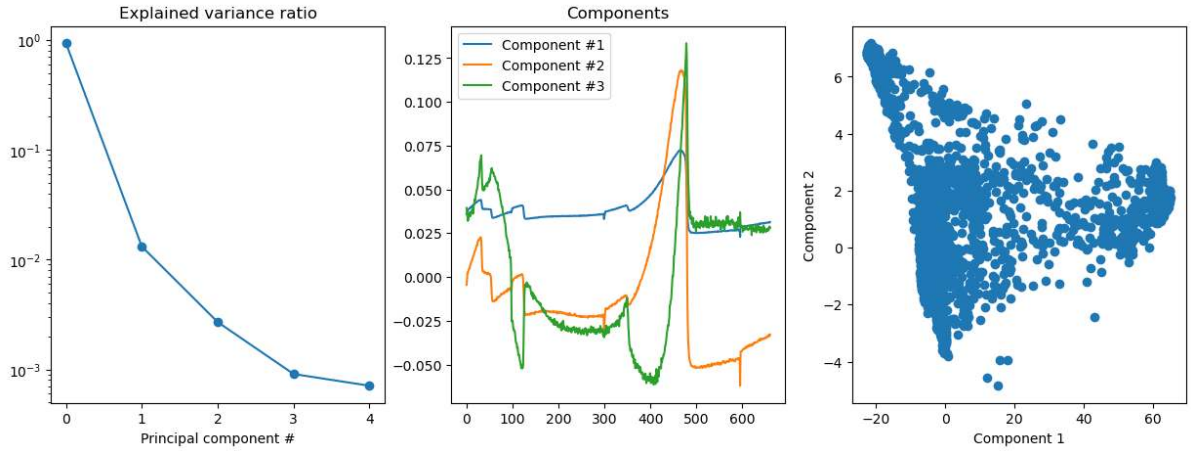
```
pca_tool.fit(mfdata)
```

```
PCA(n_components=5)
```

Inspect the PCA fit

```
score_matrix = pca_tool.transform(mfdata)
```

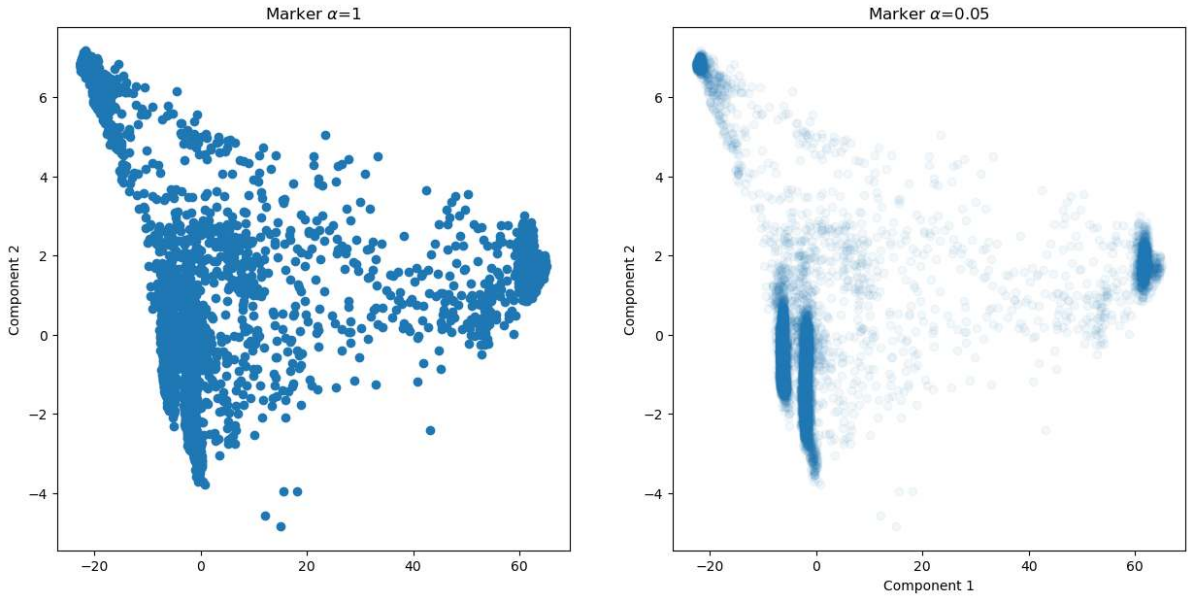
```
fig, ax = plt.subplots(1, 3, figsize=(15, 5),dpi=100)
ax[0].semilogy(pca_tool.explained_variance_ratio_,'o-');
ax[0].set_title('Explained variance ratio'); ax[0].set_xlabel('Principal component #')
ax[1].plot(pca_tool.components_[0, :], label='Component #1')
ax[1].plot(pca_tool.components_[1, :], label='Component #2')
ax[1].plot(pca_tool.components_[2, :], label='Component #3')
ax[1].legend(); ax[1].set_title('Components')
ax[2].scatter(score_matrix[:, 0], score_matrix[:, 1]); ax[2].set_xlabel('Component 1
↪'); ax[2].set_ylabel('Component 2');
```



Improve the scatter plot

The scatter plot of the data is very dense and it is not easy to spot the peaks of the clusters. This can be addressed by changing the opacity of the plotted dots. Below you see an example where the opacity value α was set to 5%. The clusters now appear much clearer than before.

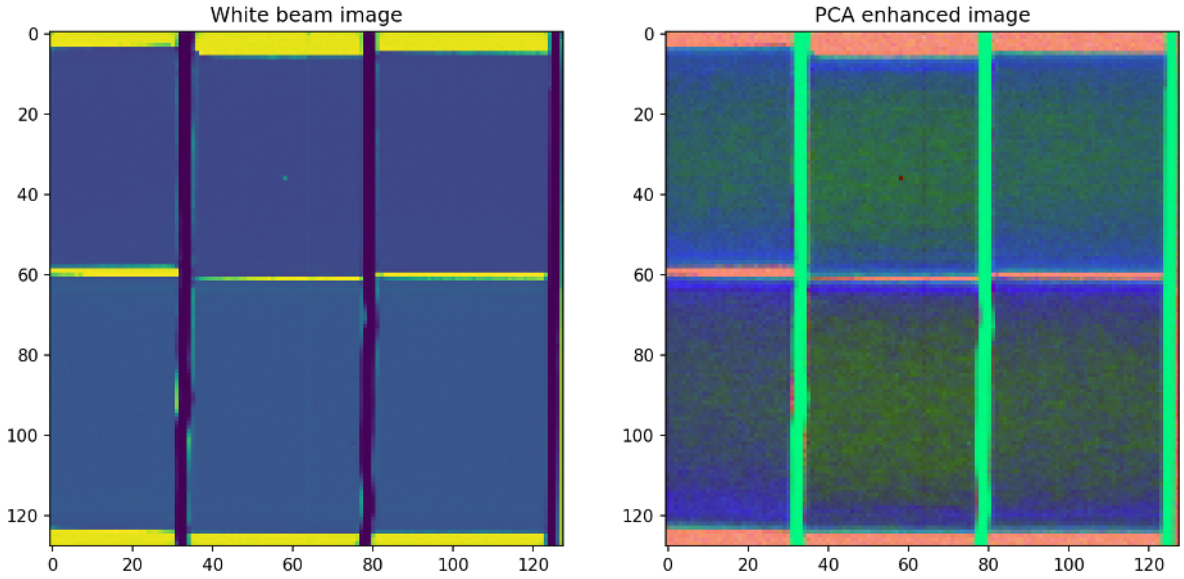
```
fig, ax = plt.subplots(1, 2, figsize=(15, 7))
ax[0].scatter(score_matrix[:, 0], score_matrix[:, 1]);
ax[1].set_xlabel('Component 1'); ax[0].set_ylabel('Component 2');
ax[0].set_title(r'Marker $\alpha$=1');

ax[1].scatter(score_matrix[:, 0], score_matrix[:, 1], alpha=0.05);
ax[1].set_xlabel('Component 1'); ax[1].set_ylabel('Component 2');
ax[1].set_title(r'Marker $\alpha$=0.05');
```



Visualize the PCAs with color coding

Inspired by PCA in materials science

```
# Reshape the first three principal components
cdata = score_matrix[:,:3].reshape([data.shape[0],data.shape[1],3])

# Normalize the chanels
for i in range(3) :
    cdata[:,:,i]=(cdata[:,:,i]-cdata[:,:,i].min())/(cdata[:,:,i].max()-cdata[:,:,i].
 ↪min())

fig, ax = plt.subplots(1,2,figsize=(12,6), dpi=150)
ax[0].imshow(data.mean(axis=2)); ax[0].set_title('White beam image')
ax[1].imshow(cdata); ax[1].set_title('PCA enhanced image');
```

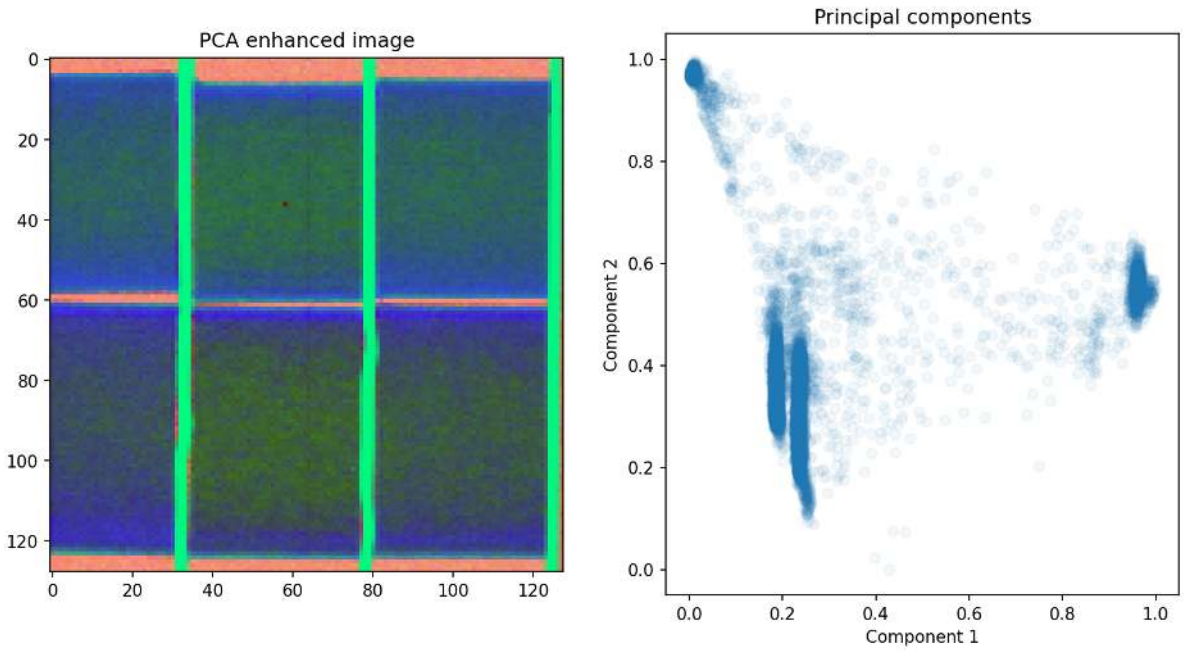


The PCA enhanced image does not represent a physical reality, but it can be used as a qualitative tool to guide the material analysis.

PCA and segmentation

The next step is to use the enhanced image for segmentation:

```
fig, ax = plt.subplots(1,2,figsize=(12,6), dpi=150)
ax[0].imshow(cdata); ax[0].set_title('PCA enhanced image');
ax[1].scatter(score_matrix[:, 0], score_matrix[:, 1], alpha=0.05); ax[1].set_xlabel(
 ↪'Component 1'); ax[1].set_ylabel('Component 2');
ax[1].set_title(r'Principal components');
```



A project task…

1. Combine clustering and PCA to identify regions in the samples

0.8.4 SciKit-learn Face Analyis

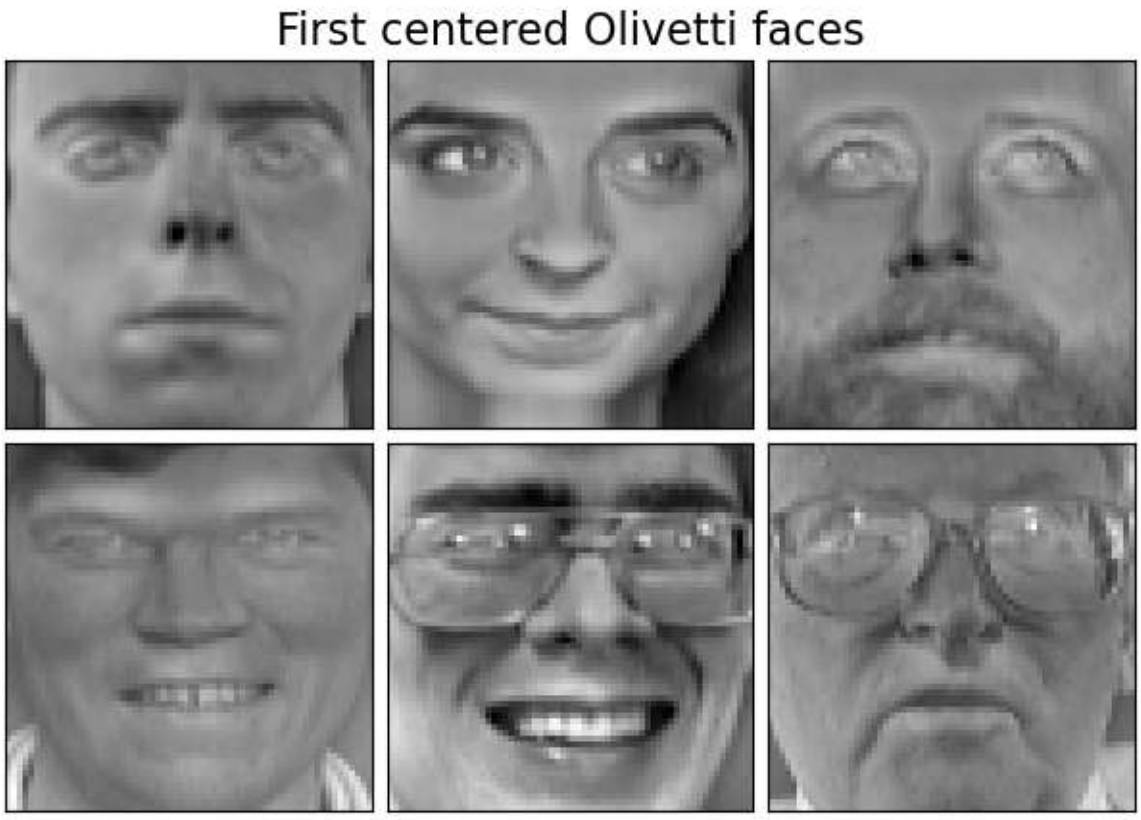
This is another example of using PCA in image analysis. It is not covered during the lecture and provided fo the interested reader. Here we show a more imaging related example from the scikit-learn documentation where we do basic face analysis with scikit-learn.

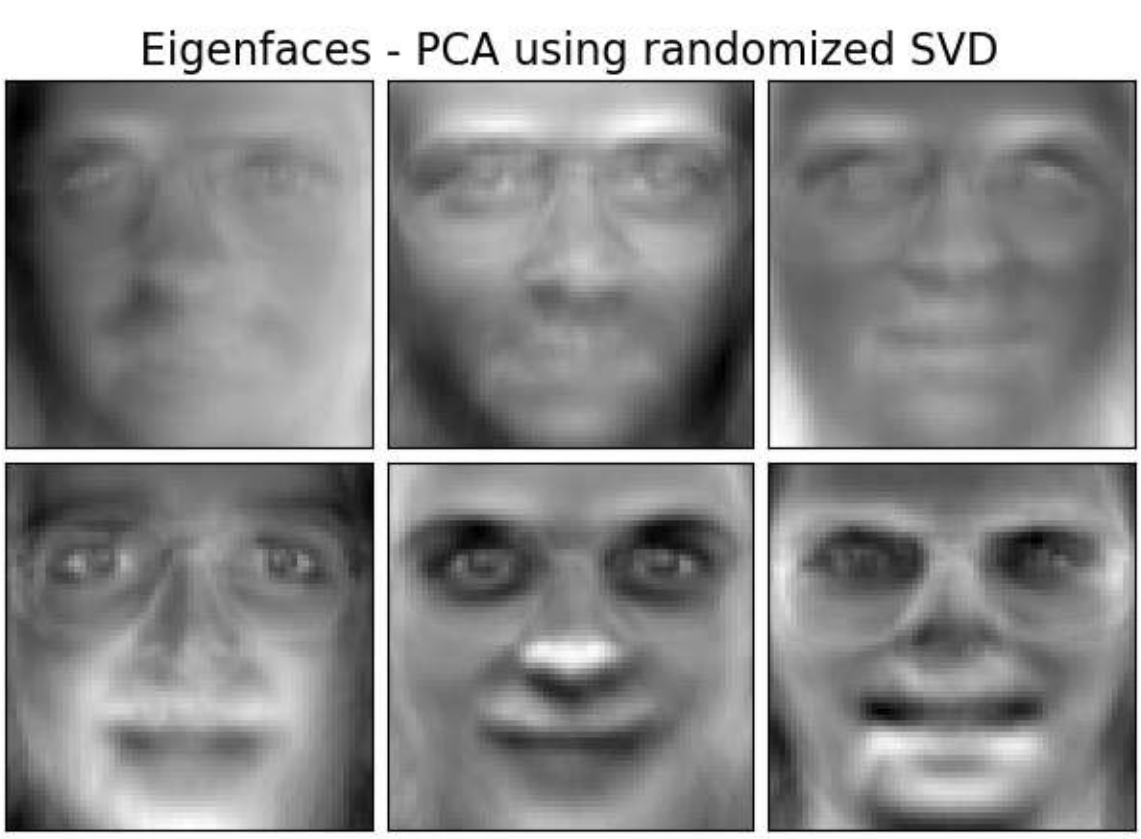
```
from sklearn.datasets import fetch_olivetti_faces
from sklearn import decomposition
# Load faces data
try:
    dataset = fetch_olivetti_faces(
        shuffle=True, random_state=2018, data_home='.')
    faces = dataset.data
except Exception as e:
    print('Face data not available', e)
    faces = np.random.uniform(0, 1, (400, 4096))

n_samples, n_features = faces.shape
n_row, n_col = 2, 3
n_components = n_row * n_col
image_shape = (64, 64)

# global centering
faces_centered = faces - faces.mean(axis=0)

# local centering
faces_centered -= faces_centered.mean(axis=1).reshape(n_samples, -1)

print("Dataset consists of %d faces" % n_samples)
```

```
Dataset consists of 400 faces
```

```
def plot_gallery(title, images, n_col=n_col, n_row=n_row):
    plt.figure(figsize=(2. * n_col, 2.26 * n_row))
    plt.suptitle(title, size=16)
    for i, comp in enumerate(images):
        plt.subplot(n_row, n_col, i + 1)
        vmax = max(comp.max(), -comp.min())
        plt.imshow(comp.reshape(image_shape), cmap=plt.cm.gray,
                   interpolation='nearest',
                   vmin=-vmax, vmax=vmax)
        plt.xticks(())
        plt.yticks(())
    plt.subplots_adjust(0.01, 0.05, 0.99, 0.93, 0.04, 0.)


# ############################################################################
# List of the different estimators, whether to center and transpose the
# problem, and whether the transformer uses the clustering API.
estimators = [
    ('Eigenfaces - PCA using randomized SVD',
     decomposition.PCA(n_components=n_components, svd_solver='randomized',
                       whiten=True),
     True)]
# ############################################################################
# Plot a sample of the input data

plot_gallery("First centered Olivetti faces", faces_centered[:n_components])

# ############################################################################
# Do the estimation and plot it

for name, estimator, center in estimators:
    print("Extracting the top %d %s..." % (n_components, name))
    data = faces
    if center:
        data = faces_centered
    estimator.fit(data)

    if hasattr(estimator, 'cluster_centers_'):
        components_ = estimator.cluster_centers_
    else:
        components_ = estimator.components_
    plot_gallery(name,
                 components_[:n_components])

plt.show()
```

```
Extracting the top 6 Eigenfaces - PCA using randomized SVD...
```


First centered Olivetti faces


Eigenfaces - PCA using randomized SVD

0.9 Applied PCA: Shape Tensor

0.9.1 How do these statistical analyses help us?

Going back to a single cell, we have the a distribution of x and y values.

- are not however completely independent
- greatest variance does not normally lie in either x nor y alone.

A principal component analysis of the voxel positions, will calculate two new principal components (the components themselves are the relationships between the input variables and the scores are the final values.)

- An optimal rotation of the coordinate system

We start off by calculating the covariance matrix from the list of x, y, and z points that make up our object of interest.

$$COV(I_{id}) = \frac{1}{N}\sum_{\forall \vec{v} \in I_{id}} \begin{bmatrix} \vec{v}_x\vec{v}_x & \vec{v}_x\vec{v}_y & \vec{v}_x\vec{v}_z \\ \vec{v}_y\vec{v}_x & \vec{v}_y\vec{v}_y & \vec{v}_y\vec{v}_z \\ \vec{v}_z\vec{v}_x & \vec{v}_z\vec{v}_y & \vec{v}_z\vec{v}_z \end{bmatrix}$$

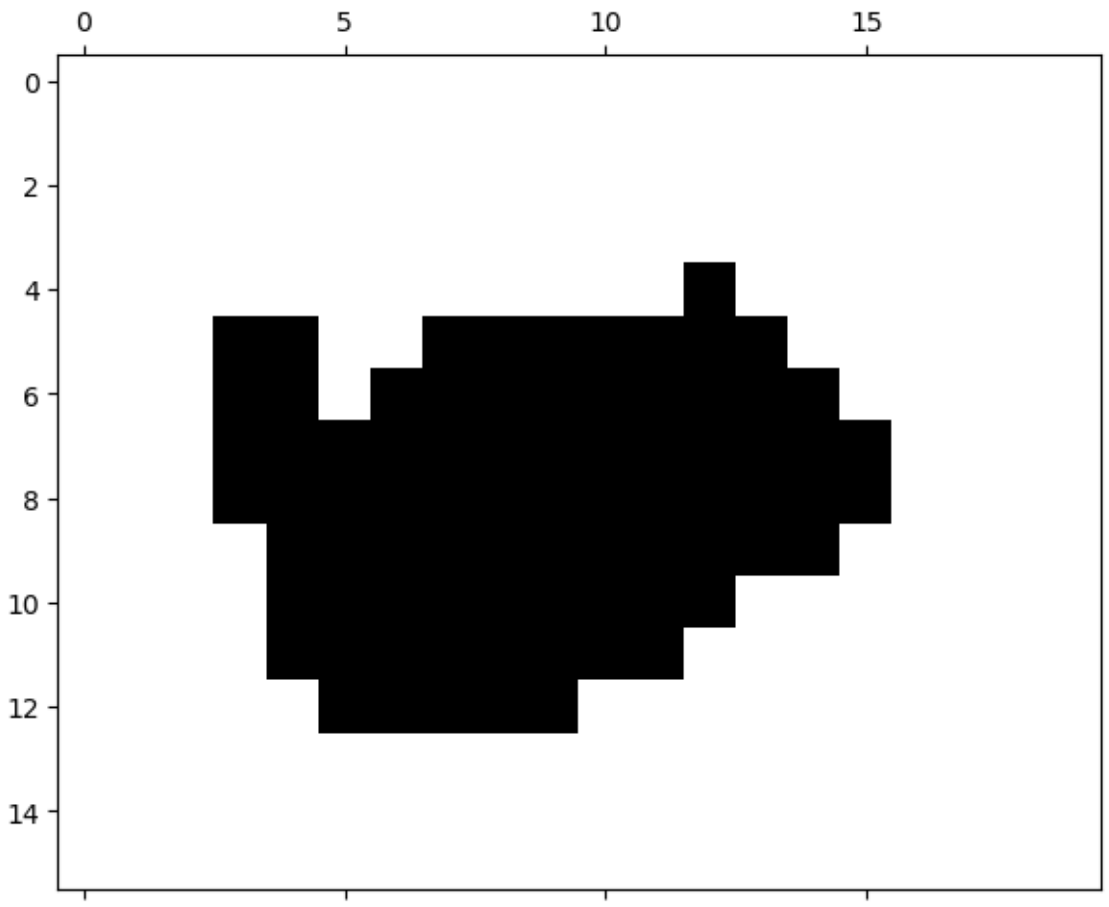
We then take the eigentransform of this array to obtain the eigenvectors (principal components, $\vec{\Lambda}_{1\cdots3}$) and eigenvalues (scores, $\lambda_{1\cdots3}$)

$$COV(I_{id}) \longrightarrow \underbrace{\begin{bmatrix} \vec{\Lambda}_{1x} & \vec{\Lambda}_{1y} & \vec{\Lambda}_{1z} \\ \vec{\Lambda}_{2x} & \vec{\Lambda}_{2y} & \vec{\Lambda}_{2z} \\ \vec{\Lambda}_{3x} & \vec{\Lambda}_{3y} & \vec{\Lambda}_{3z} \end{bmatrix}}_{\text{Eigenvectors}} * \underbrace{\begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix}}_{\text{Eigenvalues}} * \underbrace{\begin{bmatrix} \vec{\Lambda}_{1x} & \vec{\Lambda}_{1y} & \vec{\Lambda}_{1z} \\ \vec{\Lambda}_{2x} & \vec{\Lambda}_{2y} & \vec{\Lambda}_{2z} \\ \vec{\Lambda}_{3x} & \vec{\Lambda}_{3y} & \vec{\Lambda}_{3z} \end{bmatrix}^T}_{\text{Eigenvectors}}$$

The principal components tell us about the orientation of the object and the scores tell us about the corresponding magnitude (or length) in that direction.

0.9.2 Testing orientation analysis

```
seg_img = imread('figures/aachen_label.png') == 26
seg_img = seg_img[::4, ::4]
seg_img = seg_img[130:110:-2, 378:420:3] > 0
seg_img = np.pad(seg_img, 3, mode='constant')
seg_img[0, 0] = 0
_, (ax1) = plt.subplots(1, 1, figsize=(7, 7), dpi=100)
ax1.matshow(seg_img, cmap='bone_r');
```



Eigenvectors of the positions

```
x_coord, y_coord = np.where(seg_img > 0) # Get object coordinates
xy_pts = np.stack([x_coord, y_coord], 1) # Build a N x 2 matrix

shape_pca = PCA()
shape_pca.fit(xy_pts)
```

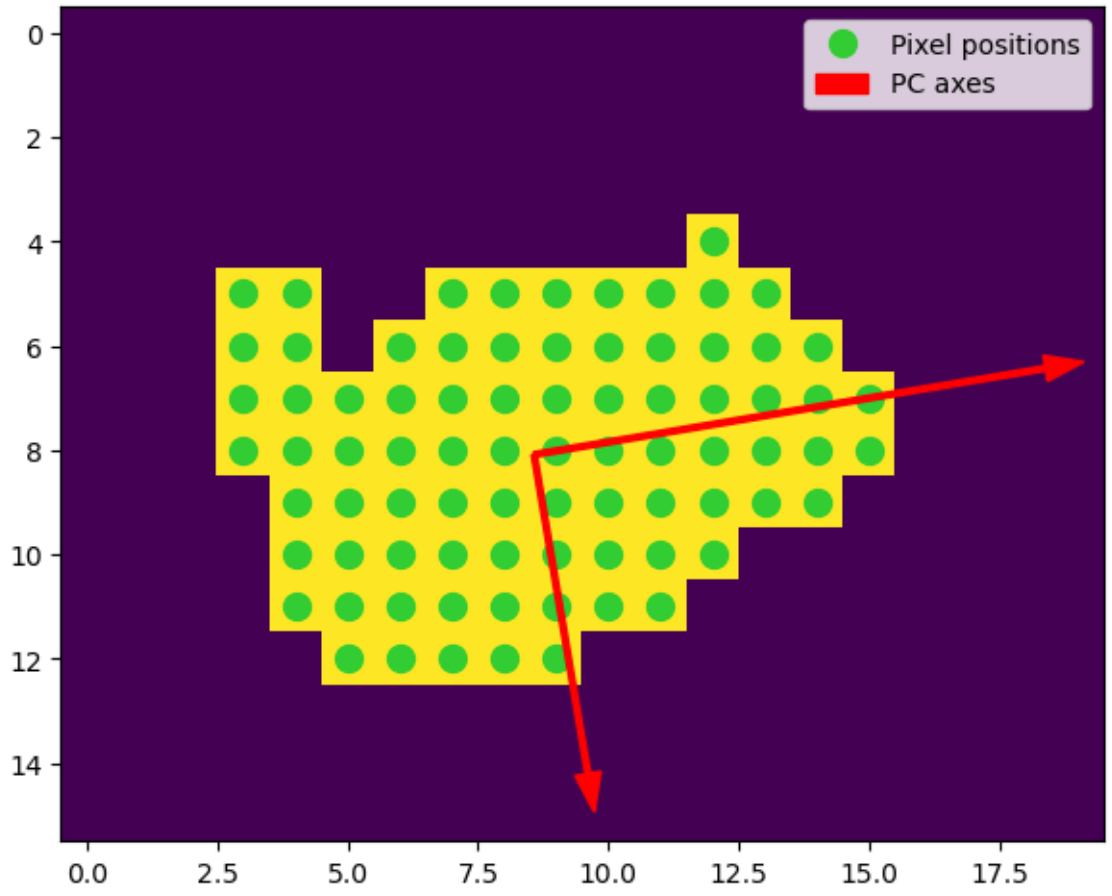
```
PCA()
```

```
_, (ax1) = plt.subplots(1, 1,
                        figsize=(7, 7),
                        dpi=100)

ax1.imshow(seg_img>0)
ax1.plot(xy_pts[:, 1], xy_pts[:, 0], 'o', color='limegreen', markersize=10,label=
 ↪'Pixel positions',zorder=1);

# draw the eigen vectors
mean = shape_pca.mean_

for idx,(length, vector) in enumerate(zip(shape_pca.explained_variance_, shape_pca.
```

(continues on next page)

(continued from previous page)

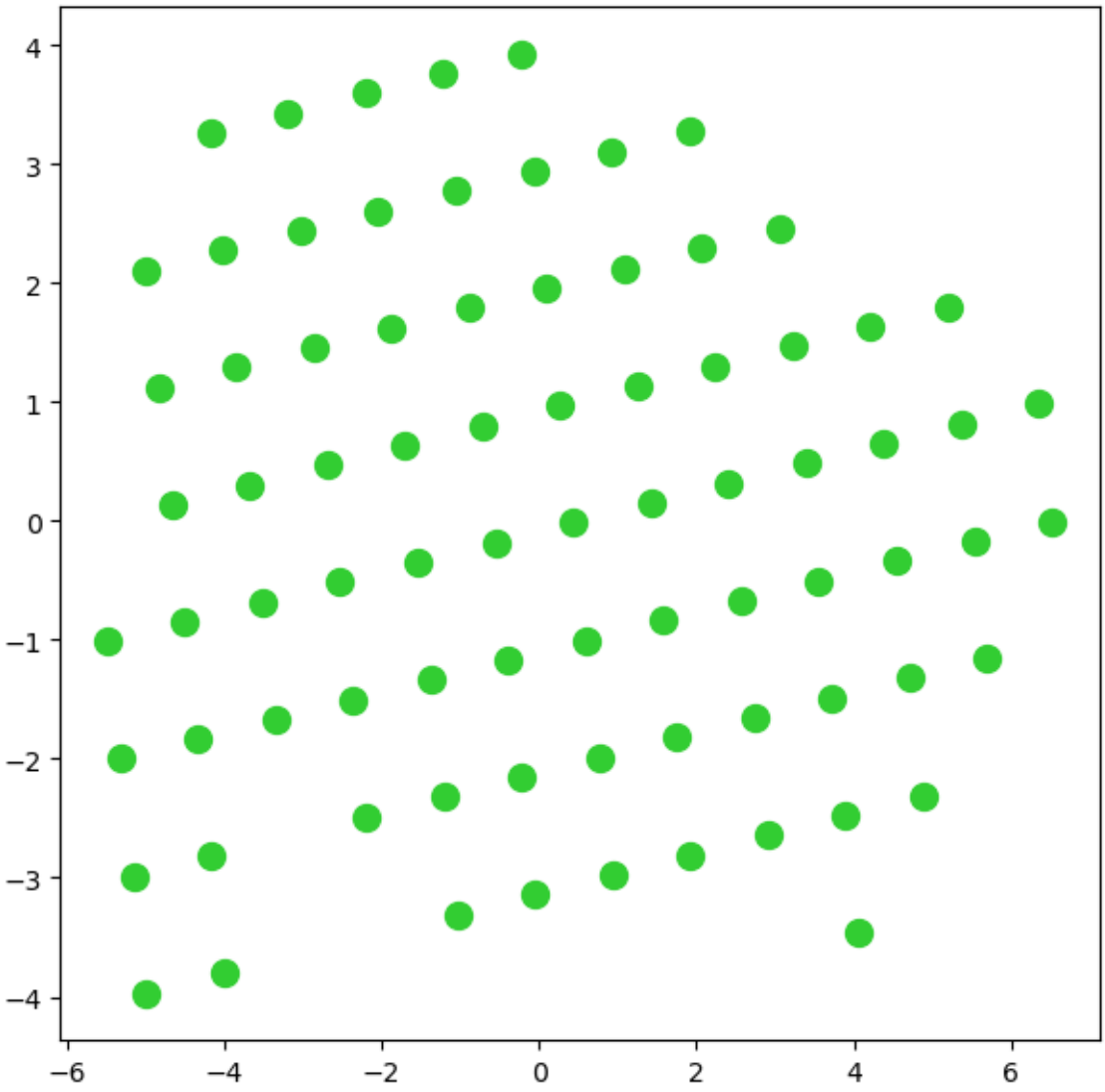
```
↪components_)):
    arrow = vector * np.sqrt(length) * 3  # scale for visibility
    arr=plt.arrow(mean[1], mean[0], arrow[1], arrow[0],
              color='red', width=0.1, head_width=0.5,zorder=2)

arr.set_label('PC axes')

plt.legend();
```



Rotate object using eigenvectors

```
from sklearn.decomposition import PCA
x_coord, y_coord = np.where(seg_img > 0)      # Get object coordinates
xy_pts      = np.stack([x_coord, y_coord], 1) # Build a N x 2 matrix
shape_pca   = PCA()
shape_pca.fit(xy_pts)
pca_xy_vals = shape_pca.transform(xy_pts)
```

```
_, (ax1) = plt.subplots(1, 1,
                        figsize=(7, 7),
                        dpi=100)
ax1.plot(pca_xy_vals[:, 0], pca_xy_vals[:, 1], 'o', color='limegreen', markersize=10);
```



0.9.3 Principal Component Analysis: Take home message

- We calculate the statistical distribution individually for x, y, and z and the 'correlations' between them.
- From these values we can estimate the orientation in the direction of largest variance
- We can also estimate magnitude
- These functions are implemented as `princomp` or `pca` in various languages and scale well to very large datasets.

0.9.4 Principal Component Analysis: Elliptical Model

While the eigenvalues and eigenvectors are in their own right useful

- Not obvious how to visually represent these tensor objects
- Ellipsoidal (Ellipse in 2D) representation alleviates this issue

Ellipsoidal Representation

1. Center of Volume is calculated normally
2. Eigenvectors represent the unit vectors for the semiaxes of the ellipsoid
3. $\sqrt{\text{Eigenvalues}}$ is proportional to the length of the semiaxis ($l = \sqrt{5\lambda_i}$), derivation similar to moment of inertia tensor for ellipsoids.

0.10 Next Time on QBI

So, while bounding box and ellipse-based models are useful for many object and cells, they do a very poor job with other samples

---

0.10.1 Why

- We assume an entity consists of connected pixels (wrong)
- We assume the objects are well modeled by an ellipse (also wrong)

0.10.2 What to do?

- Is it 3 connected objects which should all be analzed seperately?
- If we could **divide it**, we could then analyze each spart as an ellipse
- Is it one network of objects and we want to know about the constrictions?
- Is it a cell or organelle with docking sites for cell?
- Neither extents nor anisotropy are very meaningful, we need a **more specific metric** which can characterize

0.11 Summary

0.11.1 Component labeling

- What is it
- What is the impact of difference neighborhoods

0.11.2 Object description

- Center of objects
- Bounding boxes - extents

0.11.3 Principal component analysis

- Definition
- PCA in spectroscopy
- PCA to analyze shapes

0.12 Advanced Shape and Texture

0.12.1 Literature / Useful References

Books

- Pierre Soille, Morphological image analysis
- Jean Claude, Morphometry with R through ETHZ
- John C. Russ, "The Image Processing Handbook",(Boca Raton, CRC Press)

Papers / Sites

Thickness

- Hildebrand, T., & Ruegsegger, P. (1997). A new method for the model-independent assessment of thickness in three-dimensional images. Journal of Microscopy, 185(1), 67–75.

Curvature

- Mean curvature
- "Computation of Surface Curvature from Range Images Using Geometrically Intrinsic Weights"*, T. Kurita and P. Boulanger, 1992.
- http://radiomics.io

0.12.2 Let's load some modules for the notebook

```
import os
from tqdm.notebook import tqdm

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import sys
sys.path.append('../common')
import plotsupport as ps


from scipy import ndimage
from matplotlib.animation import FuncAnimation
from IPython.display import HTML

from scipy.ndimage import distance_transform_edt
import skimage.io as io

import skimage.transform
from skimage.feature.texture import graycomatrix, graycoprops
from skimage.util            import montage as montage2d
from skimage.morphology      import binary_opening, binary_closing, disk
from skimage.io              import imread
from skimage.filters         import gaussian
from skimage                 import measure
from skimage.color           import label2rgb

from mpl_toolkits.mplot3d.art3d import Poly3DCollection

import seaborn as sns
import warnings
warnings.filterwarnings("ignore", "use_inf_as_na")

import ipyvolume as p3

%matplotlib inline
```

0.12.3 Outline

- Motivation (Why and How?)
- What are Distance Maps?
- What are thickness maps?

- Characteristic Shapes
- Texture Analysis

0.13 Motivation (Why and How?)

0.13.1 Objects

- How can we measure sizes in complicated objects?
- How do we measure sizes relavant for diffusion or other local processes?
- How do we investigate surfaces in more detail and their shape?
- How can we compare shape of complex objects when they grow?
  - Are there characteristic shape metrics?

0.13.2 Patterns

- How to we quantify patterns inside images?
- How can compare between different patterns?
- How can we quantify as radiologists say *looking evil*?

0.14 Distance Maps: What are they?

A distance map is:

- A map (or image) of distances.
- Each point in the map is the distance that point is from a given feature of interest
  - surface of an object
  - ROI,
  - center of object, etc

0.14.1 Different distance map metrics

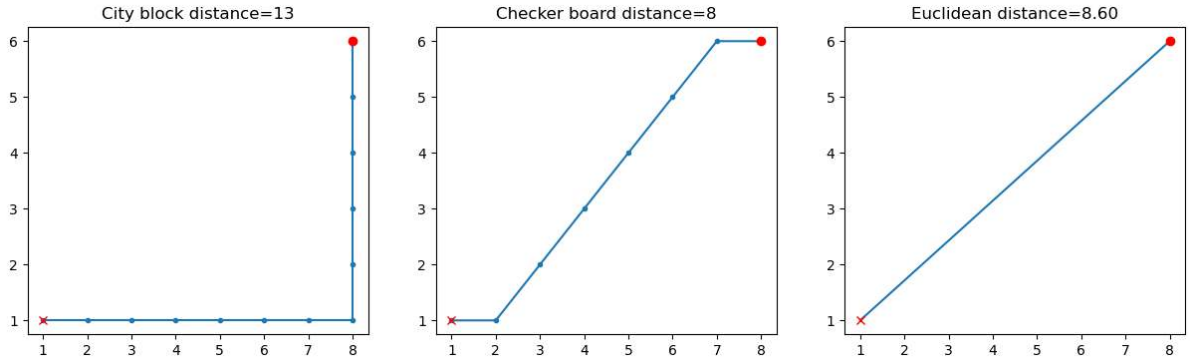
- Euclidean (and approximations)
- City block (4-connected)
- Checkerboard (8-connected)

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

city = np.array([[1,1],[1,2],[1,3],[1,4],[1,5],[1,6],[1,7],[1,8],[2,8],[3,8],[4,8],[5,
 ↪8],[6,8]])
checker = np.array([[1,1],[1,2],[2,3],[3,4],[4,5],[5,6],[6,7],[6,8]])

ax[0].plot(city[:,1],city[:,0],'.-')
ax[0].plot(city[0,1],city[0,0],'x',color='red')
ax[0].plot(city[-1,1],city[-1,0],'o',color='red')
ax[0].set_title(f'City block distance={city.shape[0]}')
```

(continues on next page)

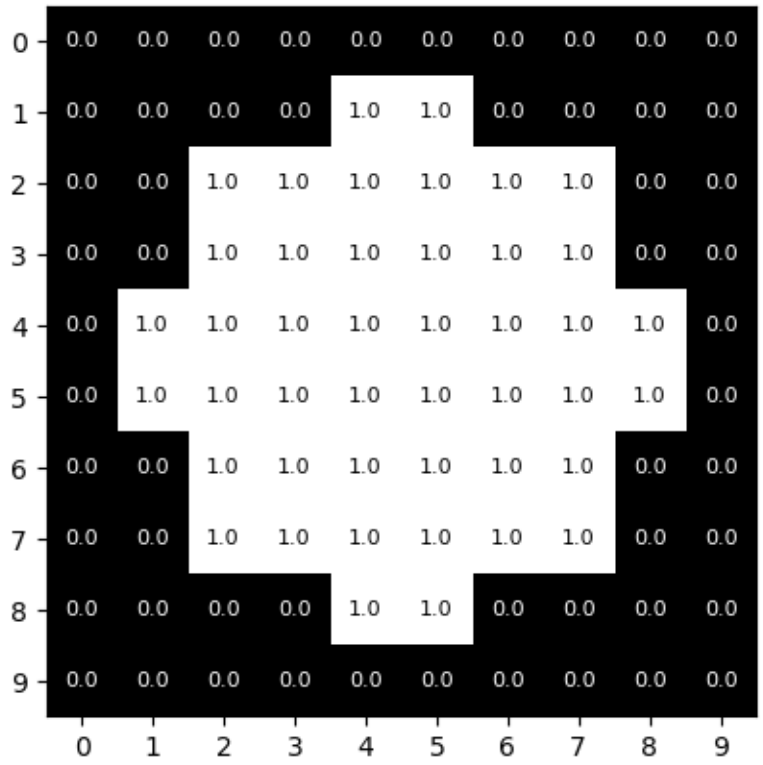
(continued from previous page)

```
ax[1].plot(checker[:,1],checker[:,0],'.-')
ax[1].plot(checker[0,1],checker[0,0],'x',color='red')
ax[1].plot(checker[-1,1],checker[-1,0],'o',color='red')
ax[1].set_title(f'Checker board distance={checker.shape[0]}')
ax[2].plot([city[0,1],city[-1,1]],[city[0,0],city[-1,0]])
ax[2].plot(city[0,1],city[0,0],'x',color='red')
ax[2].plot(city[-1,1],city[-1,0],'o',color='red')
ax[2].set_title('Euclidean distance={0:0.2f}'.format(np.sqrt((city[-1,0]-city[0,
 ↪0])**2+(city[-1,1]-city[0,1])**2)));
```



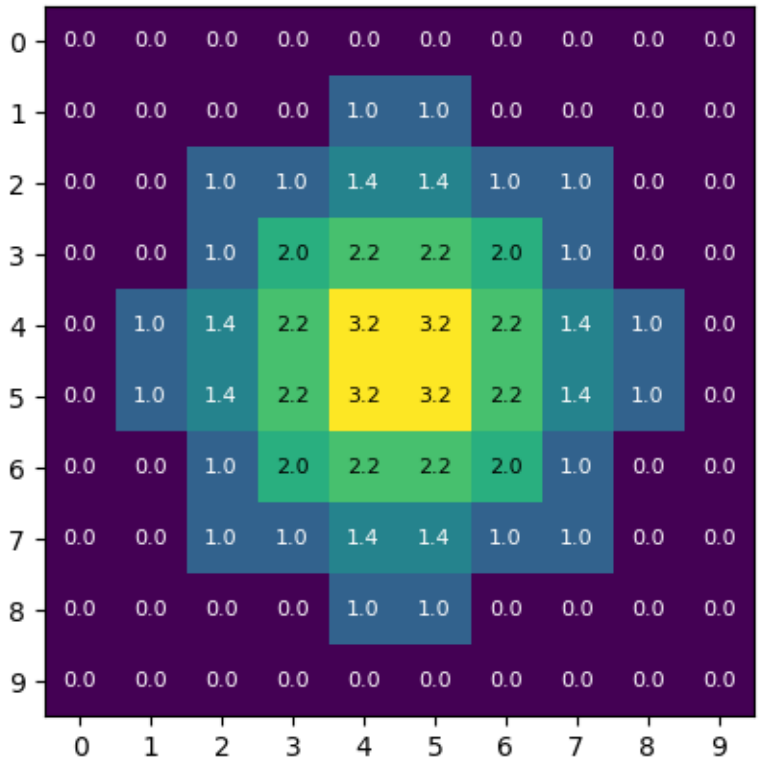
0.14.2 What does a distance map look like?

```
def generate_dot_image(size=100, rad_a=1, rad_b=None):
    xx, yy = np.meshgrid(np.linspace(-1, 1, size), np.linspace(-1, 1, size))
    if rad_b is None:
        rad_b = rad_a
    return np.sqrt(np.square(xx/rad_a)+np.square(yy/rad_b)) <= 1.0


img_bw = generate_dot_image(10, 0.8)


ps.heatmap(img_bw,cmap='gray',fontsize=8,bgbox=False,precision=1);plt.gca().set_
 ↪aspect(aspect='equal')
```



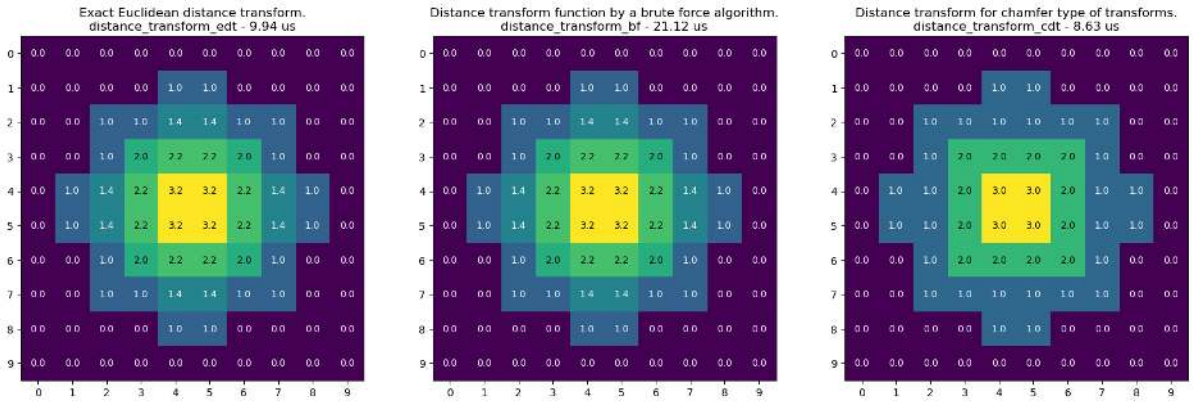
The Euclidean distance map

```
dmap = ndimage.distance_transform_edt(img_bw)

ps.heatmap(dmap,cmap='viridis',fontsize=8,bgbox=False,precision=1)
```



The distance map depends on the neighborhood

```
from timeit import timeit
dmap_list = [ndimage.distance_transform_edt,
             ndimage.distance_transform_bf, ndimage.distance_transform_cdt]
fig, m_axs = plt.subplots(1, len(dmap_list), figsize=(20, 6))
for dmap_func, c_ax in zip(dmap_list, m_axs):
    ms_time = timeit(lambda: dmap_func(img_bw), number=10000)/10000*1e6
    dmap = dmap_func(img_bw)

    ps.heatmap(dmap,cmap='viridis',fontsize=9,bgbox=False,precision=1,ax=c_ax)
    c_ax.set_title('{}\n{} - {} us'.format(dmap_func.__doc__.split('\n')[1].strip(),
                                           dmap_func.__name__,
                                           '%2.2f' % ms_time))
```



Distance maps in python

```
help(ndimage.distance_transform_edt)
```

```
Help on function distance_transform_edt in module scipy.ndimage._morphology:

distance_transform_edt(input, sampling=None, return_distances=True, return_
↪indices=False, distances=None, indices=None)
    Exact Euclidean distance transform.

    This function calculates the distance transform of the `input`, by
    replacing each foreground (non-zero) element, with its
    shortest distance to the background (any zero-valued element).

    In addition to the distance transform, the feature transform can
    be calculated. In this case the index of the closest background
    element to each foreground element is returned in a separate array.

    Parameters
    ----------
    input : array_like
        Input data to transform. Can be any type but will be converted
        into binary: 1 wherever input equates to True, 0 elsewhere.
    sampling : float, or sequence of float, optional
        Spacing of elements along each dimension. If a sequence, must be of
        length equal to the input rank; if a single number, this is used for
        all axes. If not specified, a grid spacing of unity is implied.
    return_distances : bool, optional
        Whether to calculate the distance transform.
        Default is True.
    return_indices : bool, optional
        Whether to calculate the feature transform.
        Default is False.
    distances : float64 ndarray, optional
        An output array to store the calculated distance transform, instead of
        returning it.
        `return_distances` must be True.
        It must be the same shape as `input`.
    indices : int32 ndarray, optional
        An output array to store the calculated feature transform, instead of
        returning it.
        `return_indicies` must be True.
        Its shape must be `(input.ndim,) + input.shape`.

    Returns
    -------
    distances : float64 ndarray, optional
        The calculated distance transform. Returned only when
        `return_distances` is True and `distances` is not supplied.
        It will have the same shape as the input array.
    indices : int32 ndarray, optional
        The calculated feature transform. It has an input-shaped array for each
        dimension of the input. See example below.
        Returned only when `return_indices` is True and `indices` is not
        supplied.
```

(continues on next page)

(continued from previous page)

```
    Notes
    -----
    The Euclidean distance transform gives values of the Euclidean
    distance::

                    n
      y_i = sqrt(sum (x[i]-b[i])**2)
                    i

    where b[i] is the background point (value 0) with the smallest
    Euclidean distance to input points x[i], and n is the
    number of dimensions.

    Examples
    --------
    >>> from scipy import ndimage
    >>> import numpy as np
    >>> a = np.array(([0,1,1,1,1],
    ...               [0,0,1,1,1],
    ...               [0,1,1,1,1],
    ...               [0,1,1,1,0],
    ...               [0,1,1,0,0]))
    >>> ndimage.distance_transform_edt(a)
    array([[ 0.    ,  1.    ,  1.4142,  2.2361,  3.    ],
           [ 0.    ,  0.    ,  1.    ,  2.    ,  2.    ],
           [ 0.    ,  1.    ,  1.4142,  1.4142,  1.    ],
           [ 0.    ,  1.    ,  1.4142,  1.    ,  0.    ],
           [ 0.    ,  1.    ,  1.    ,  0.    ,  0.    ]])

    With a sampling of 2 units along x, 1 along y:

    >>> ndimage.distance_transform_edt(a, sampling=[2,1])
    array([[ 0.    ,  1.    ,  2.    ,  2.8284,  3.6056],
           [ 0.    ,  0.    ,  1.    ,  2.    ,  3.    ],
           [ 0.    ,  1.    ,  2.    ,  2.2361,  2.    ],
           [ 0.    ,  1.    ,  2.    ,  1.    ,  0.    ],
           [ 0.    ,  1.    ,  1.    ,  0.    ,  0.    ]])

    Asking for indices as well:

    >>> edt, inds = ndimage.distance_transform_edt(a, return_indices=True)
    >>> inds
    array([[[0, 0, 1, 1, 3],
            [1, 1, 1, 1, 3],
            [2, 2, 1, 3, 3],
            [3, 3, 4, 4, 3],
            [4, 4, 4, 4, 4]],
           [[0, 0, 1, 1, 4],
            [0, 1, 1, 1, 4],
            [0, 0, 1, 4, 4],
            [0, 0, 3, 3, 4],
            [0, 0, 3, 3, 4]]])

    With arrays provided for inplace outputs:

    >>> indices = np.zeros(((np.ndim(a),) + a.shape), dtype=np.int32)
```

(continues on next page)

(continued from previous page)

```
    >>> ndimage.distance_transform_edt(a, return_indices=True, indices=indices)
    array([[ 0.    ,  1.    ,  1.4142,  2.2361,  3.    ],
           [ 0.    ,  0.    ,  1.    ,  2.    ,  2.    ],
           [ 0.    ,  1.    ,  1.4142,  1.4142,  1.    ],
           [ 0.    ,  1.    ,  1.4142,  1.    ,  0.    ],
           [ 0.    ,  1.    ,  1.    ,  0.    ,  0.    ]])
    >>> indices
    array([[[0, 0, 1, 1, 3],
            [1, 1, 1, 1, 3],
            [2, 2, 1, 3, 3],
            [3, 3, 4, 4, 3],
            [4, 4, 4, 4, 4]],
           [[0, 0, 1, 1, 4],
            [0, 1, 1, 1, 4],
            [0, 0, 1, 4, 4],
            [0, 0, 3, 3, 4],
            [0, 0, 3, 3, 4]]])
```

0.14.3 Why does speed matter?

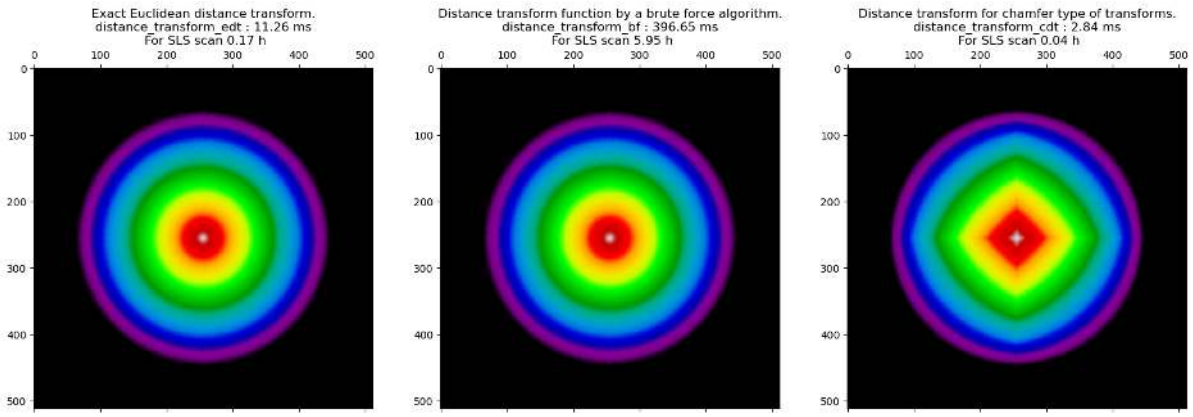
As for the question why speed matters,

- for small images clearly the more efficient approaches don't make much of a difference,
- how about data from a Synchrotron measurement 2160x2560x2560 (SLS)?
- even worse when we have series of large volumes…

Now, time becomes much more important

Extrapolated execution time

```
img_bw = generate_dot_image(512, 0.75)
sls_fact = 2160*2560*2560/np.prod(img_bw.shape)
dmap_list = [ndimage.distance_transform_edt,
             ndimage.distance_transform_bf, ndimage.distance_transform_cdt]
fig, m_axs = plt.subplots(1, len(dmap_list), figsize=(20, 6))
for dmap_func, c_ax in zip(dmap_list, m_axs):
    ms_time = timeit(lambda: dmap_func(img_bw), number=1)*1e3
    dmap = dmap_func(img_bw)
    c_ax.matshow(dmap, cmap='nipy_spectral')
    c_ax.set_title('{0}\n{1} : {2:0.2f} ms\n For SLS scan {3:0.2f} h'.format(dmap_
 ↪func.__doc__.split('\n')[1].strip(),
                                                                    dmap_func.__name__,
                                                                    ms_time,
                                                                    (ms_time*sls_fact/3600/
 ↪1e3)))
```



In this example you can see that the execution time for a distance transform can vary quite a lot depending on the algorithm. The brute force algorithm owuld in this case only serve as prototype or baseline for more advanced and faster algorithms.

The chamfer distance is an approximative Euclidean distance transform which has sacrificed the distance accuracy to gain in execution time. It is up to the scientist using the distance transform to decide if it is justified to use this transform depending on the application. This naturally depends on the application of the distance transform. Will it be used a guide for a following algorithm or will it be used to quantify dimensions in the images? You can maybe also see that the accuracy of the chamfer distance depends on the distance from the edge of the object. The errors are less for short distances which is an interesting observation when the items in the image are small.

0.14.4 Distance map - Definition

If we start with an image as a collection of points divided into two categories

- $Im(x, y) = \{\text{Foreground, Background}\}$
- We can define a distance map operator $(dist)$ that transforms the image into a distance map

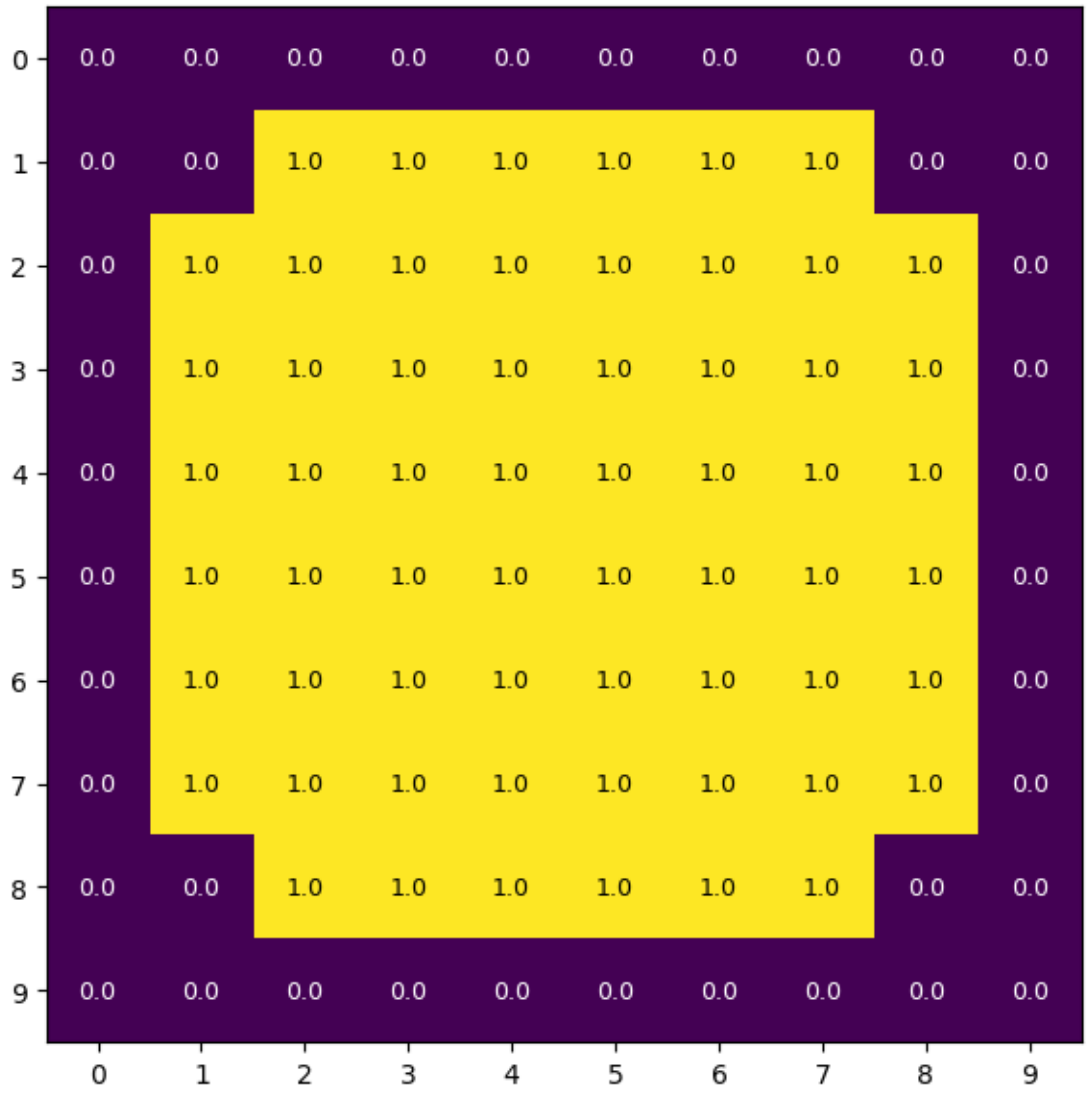
$$dist(\vec{x}) = \min(||\vec{x} - \vec{y}|| \forall \vec{y} \in \text{Background})$$

We will use Euclidean distance $||\vec{x} - \vec{y}||$ for this class but there are other metrics which make sense when dealing with other types of data like Manhattan/City-block or weighted metrics.

```
def simple_distance_iteration(last_img):
    cur_img = last_img.copy()
    for x in range(last_img.shape[0]):
        for y in range(last_img.shape[1]):
            if last_img[x, y] >= 0:
                i_xy = last_img[x, y]
                for xp in [-1, 0, 1]: # Checking in the 3xw3 neighborhood
                    if (x+xp < last_img.shape[0]) and (x+xp >= 0):
                        for yp in [-1, 0, 1]:
                            if (y+yp < last_img.shape[1]) and (y+yp >= 0):

                                p_dist = np.sqrt(np.square(xp)+np.square(yp)) #␣
 ↪Distance metric

                                if cur_img[x+xp, y+yp] > (last_img[x, y]+p_dist):
                                    cur_img[x+xp, y+yp] = last_img[x, y]+p_dist
    return cur_img
```

A test image for distance transform

```
fig, ax=plt.subplots(1,1, figsize=(7,7))
img_bw = generate_dot_image(10, 1.0)
plt.imshow(img_bw)

ps.heatmap(img_bw,cmap='viridis',fontsize=9,bgbox=False,precision=1,ax=ax)
ax.set_aspect(aspect='equal')
```



Animating the iterations

```
fig, c_ax = plt.subplots(1, 1, figsize=(5, 5), dpi=100)
img_base = (img_bw*255).astype(np.float32)
img_list = [img_base]
for i in range(5):
    img_base = simple_distance_iteration(img_base)
    img_list += [img_base]


def update_frame(i):
    plt.cla()
    ps.heatmap(img_list[i],cmap='nipy_spectral',fontsize=9,bgbox=False,precision=1,
 ↪ax=c_ax,vmin=0, vmax=5)
    c_ax.set_title('Iteration #{}'.format(i+1))
# write animation frames
anim_code = FuncAnimation(fig, update_frame, frames=5,interval=1000,repeat_
 ↪delay=2000).to_html5_video()
plt.close('all')
HTML(anim_code)
```

```
<IPython.core.display.HTML object>
```

0.14.5 Distance Maps: Types

Using this rule a distance map can be made for the euclidean metric

Similarly the Manhattan or city block distance metric can be used where the distance is defined as $\sum_i |\vec{x} - \vec{y}|_i$

```
def simple_distance_iteration(last_img):
    cur_img = last_img.copy()
    for x in range(last_img.shape[0]):
        for y in range(last_img.shape[1]):
            if last_img[x, y] >= 0:             # Is img(x,y) foreground?
                i_xy = last_img[x, y]
                for xp in [-1, 0, 1]:           # Scan neighborhood
                    if (x+xp < last_img.shape[0]) and (x+xp >= 0):
                        for yp in [-1, 0, 1]:
                            if (y+yp < last_img.shape[1]) and (y+yp >= 0):

                                p_dist = np.abs(xp)+np.abs(yp)      # Distance metric

                                if cur_img[x+xp, y+yp] > (last_img[x, y]+p_dist):
                                    cur_img[x+xp, y+yp] = last_img[x, y]+p_dist
    return cur_img
```

Animating City-block distance

```
fig, c_ax = plt.subplots(1, 1, figsize=(5, 5), dpi=120)
img_base = (img_bw*255).astype(np.float32)
img_list = [img_base]
for i in range(5):
    img_base = simple_distance_iteration(img_base)
    img_list += [img_base]


def update_frame(i):
    plt.cla()
    ps.heatmap(img_list[i],cmap='nipy_spectral',fontsize=9,bgbox=False,precision=1,
 ↪ax=c_ax,vmin=0, vmax=5)
    c_ax.set_title('Iteration #{}'.format(i+1))
    c_ax.set_aspect(1)


# write animation frames
anim_code = FuncAnimation(fig, update_frame, frames=5,
                          interval=1000, repeat_delay=2000).to_html5_video()
plt.close('all')
HTML(anim_code)
```

```
<IPython.core.display.HTML object>
```

0.14.6 Distance Maps: Precaution

The distance map is one of the crictical points where the resolution of the imaging system is important.

- We measure distances computationally in pixels or voxels
- but for them to have a meaning physically they must be converted
- Isotropic imaging (1 μ m x 1μ m x 1 μ m) is **fine**

Anisotropic

Options to handle anisotropic voxels:

- as part of filtering, resample and convert to an isotropic scale.
- custom distance map algorithms
  - use the side-lengths of the voxels to calculate distance rather than assuming 1x1x1

0.14.7 What does the Distance Maps show?

We can create 2 distance maps

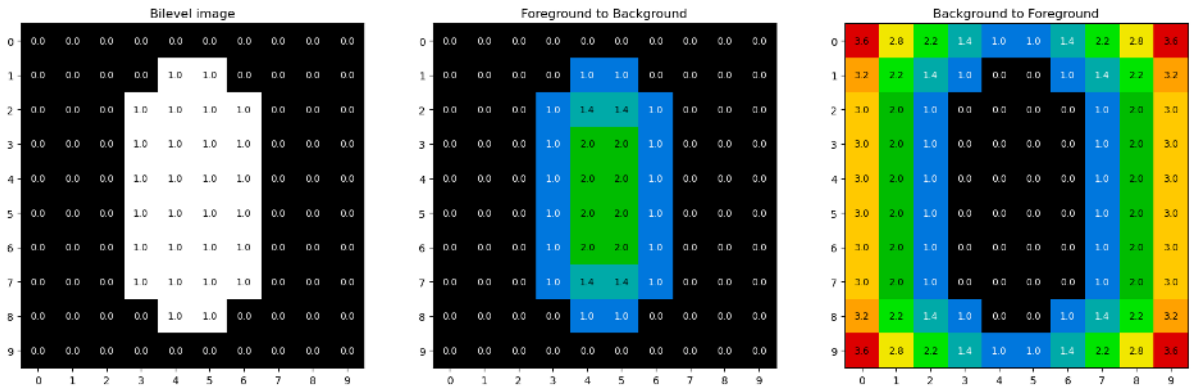
Foreground → Background

Information about the objects size and interior

Background → Foreground

Information about the distance / space between objects

```
def generate_dot_image(size=100, rad_a=1, rad_b=None):
    xx, yy = np.meshgrid(np.linspace(-1, 1, size), np.linspace(-1, 1, size))
    if rad_b is None:
        rad_b = rad_a
    return np.sqrt(np.square(xx/rad_a)+np.square(yy/rad_b)) <= 1.0


img_bw = generate_dot_image(10, 0.5, 1.0)
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 6))

ps.heatmap(img_bw,cmap='gray',fontsize=9,bgbox=False,precision=1,ax=ax1)
ax1.set_title('Bilevel image')

ps.heatmap(ndimage.distance_transform_edt(img_bw),cmap='nipy_spectral',fontsize=9,
 ↪bgbox=False,precision=1,ax=ax2,vmin=0, vmax=4)
ax2.set_title('Foreground to Background')

ps.heatmap(ndimage.distance_transform_edt(1-img_bw),cmap='nipy_spectral',fontsize=9,
 ↪bgbox=False,precision=1,ax=ax3,vmin=0, vmax=4)
ax3.set_title('Background to Foreground');
```



0.14.8 Distance Map

One of the most useful components of the distance map is that it is *relatively* insensitive to small changes in connectivity.

- Component Labeling would find radically different results for these two images
- One has 4 small circles
- One has 1 big blob

Some code for the demo

```
def update_frame(i):
    [tax.cla() for tax in m_axs.flatten()]
    ((c_ax, c_hist, c_dmean), (c_lab, c_labhist, c_lmean)) = m_axs
    c_ax.imshow(img_list[i],vmin=0,vmax=5, cmap='nipy_spectral')

    c_ax.set_title('DMap #{}'.format(i+1))
    c_hist.hist(img_list[i].ravel(),
                np.linspace(0, 3, 6))
    c_hist.set_title('Mean: %2.2f\nStd: %2.2f' % (np.mean(img_list[i][img_list[i] >␣
↪0]),
                                                  np.std(img_list[i][img_list[i] >␣
↪0])))

    c_dmean.plot(range(i+1), [np.mean(img_list[k][img_list[k] > 0])
                              for k in range(i+1)], 'r+-')
    c_dmean.set_ylim(0, 2)
    c_dmean.set_title('Avergage Distance')

    lab_img = ndimage.label(img_list[i] > 0)[0]
    c_lab.imshow(lab_img,vmin=0,vmax=2, cmap='nipy_spectral')

    c_lab.set_title('Component Labeled')

    def avg_area(c_img):
        l_img = ndimage.label(c_img > 0)[0]
        return np.mean([np.sum(l_img == k) for k in range(1, l_img.max()+1)])

    n, _, _ = c_labhist.hist(lab_img[lab_img > 0].ravel(), [
                             0, 1, 2, 3], rwidth=0.8)
    c_labhist.set_title('# Components: %d\nAvg Area: %d' %
                        (lab_img.max(), avg_area(img_list[i])))

    c_lmean.plot(range(i+1),
                 [avg_area(img_list[k]) for k in range(i+1)], 'r+-')
    c_lmean.set_ylim(0, 150)
    c_lmean.set_title('Average Area')
    plt.tight_layout()

fig, m_axs = plt.subplots(2, 3, figsize=(9, 5))

img_list = []
for i in np.linspace(0.6, 1.3, 7):
    img_bw = np.concatenate([generate_dot_image(12, 0.7, i),
                             generate_dot_image(12, 0.6, i)], 0)
    img_base = ndimage.distance_transform_edt(img_bw)
```

(continues on next page)

(continued from previous page)

```
    img_list += [img_base]

# write animation frames
anim_code = FuncAnimation(fig,
                          update_frame,
                          frames=len(img_list)-1,
                          interval=1000,
                          repeat_delay=2000).to_html5_video()
plt.close('all')
```

Animate the distances

In this example we grow two objects and observe how the maximum distance changes compared to the area of labeled object. Initially, the two metrics behave similarly. It becomes interesting when the two objects grow into each other. Then the labeling only finds a single object and the area is double compared to before the merge. The distance, however, stay within the same magnitude thoughout the entire growing sequence.

```
HTML(anim_code)
```

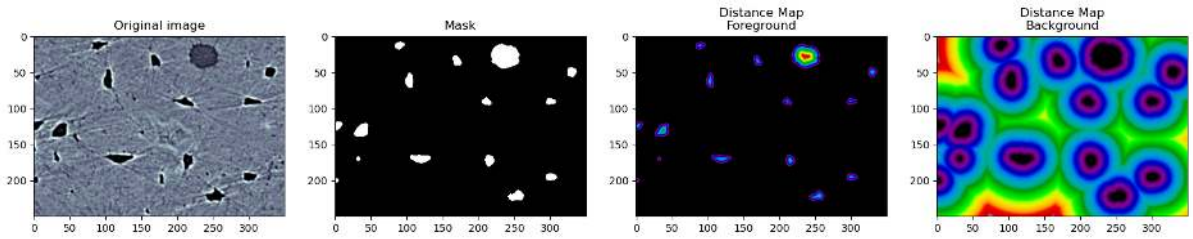
```
<IPython.core.display.HTML object>
```

0.15 Distance Map of Real Images

We now have a basic idea of how the distance map works we can now try to apply it to real images

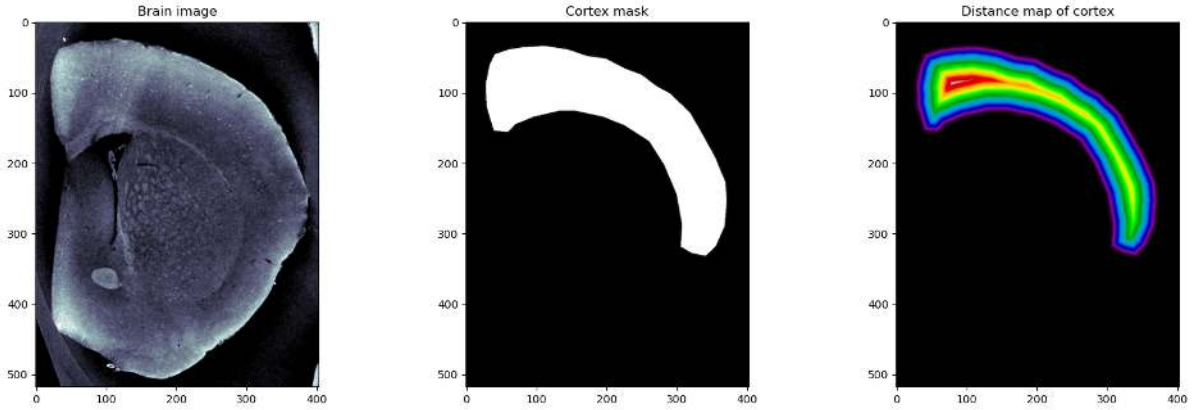
0.15.1 The bone slice

```
bw_img = imread("../Lecture-05/figures/bonegfiltslice.png")

thresh_img = binary_closing(binary_opening(bw_img < 90, disk(3)), disk(5))
fg_dmap = distance_transform_edt(thresh_img)
bg_dmap = distance_transform_edt(1-thresh_img)

fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(20, 6), dpi=100)
ax1.imshow(bw_img, cmap='bone'),           ax1.set_title('Original image')
ax2.imshow(thresh_img, cmap='bone'),       ax2.set_title('Mask')
ax3.imshow(fg_dmap, cmap='nipy_spectral'); ax3.set_title('Distance Map\nForeground');
ax4.imshow(bg_dmap, cmap='nipy_spectral'); ax4.set_title('Distance Map\nBackground');
```



0.15.2 The cortex

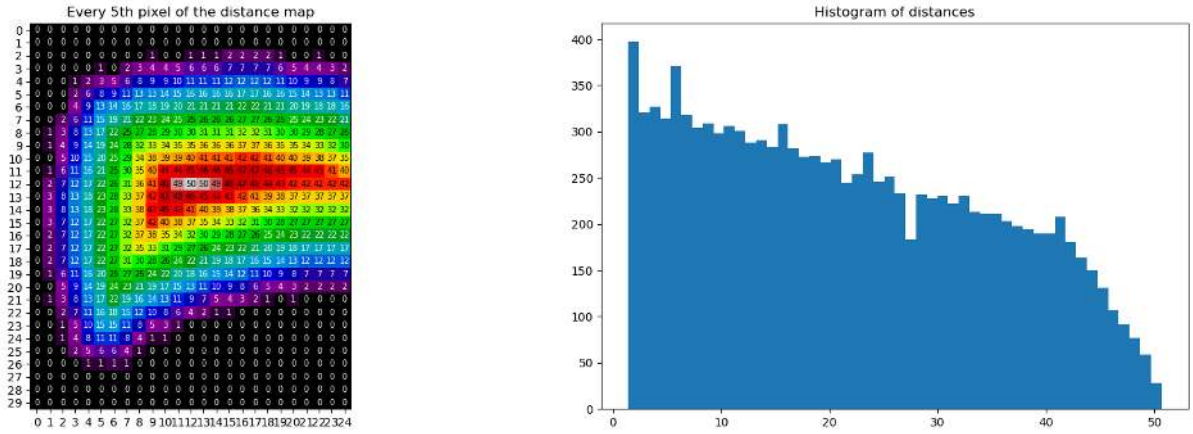
```
cortex_img  = imread("figures/example_poster.tif")[::2, ::2]/2048
cortex_mask = imread("figures/example_poster_mask.tif")[::1, ::1, 0]/255.0
cortex_dmap = distance_transform_edt(cortex_mask,distances=True)
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 6), dpi=100)

ax1.imshow(cortex_img, cmap='bone'), ax1.set_title('Brain image')
ax2.imshow(cortex_mask, cmap='bone'), ax2.set_title('Cortex mask')
ax3.imshow(cortex_dmap, cmap='nipy_spectral'); ax3.set_title('Distance map of cortex
↪');
```



0.15.3 How can we utilize this information?

- So we can see in the distance map some information about the size of the object,
- but the raw values and taking a histogram do not seem to provide this very clearly

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 6))
cortex_dmap_roi = cortex_dmap[25:175, 25:150]
ps.heatmap(cortex_dmap_roi[::5,::5], ax=ax1, precision=0,bgbox=False, fontsize=7,cmap=
↪'nipy_spectral')
ax1.set_title('Every 5th pixel of the distance map')
ax2.hist(cortex_dmap_roi[cortex_dmap_roi > 1].ravel(), 50); ax2.set_title('Histogram␣
↪of distances');
```



```
from skimage.morphology import binary_opening, binary_closing, disk
from scipy.ndimage import distance_transform_edt
import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imread
%matplotlib inline
```

0.15.4 Compute average as function of distance

The distance map can be used measure

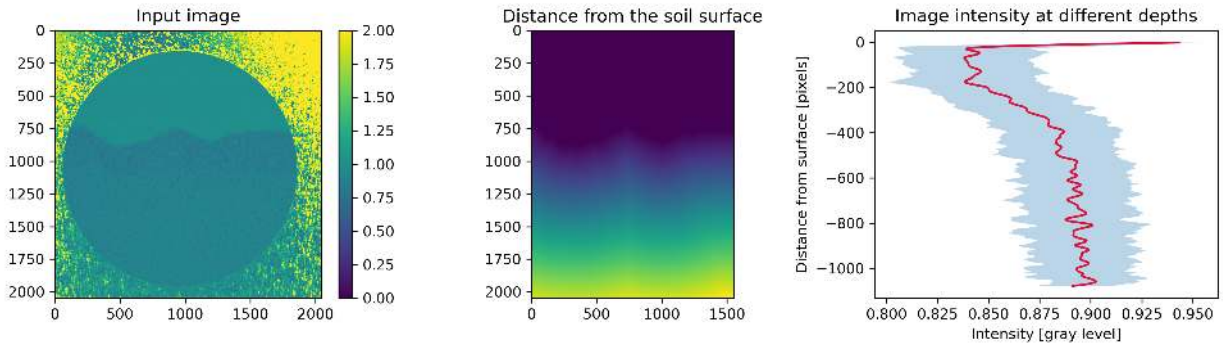
- gray levels
- porosity

As function of distance:

1. Find all pixels p at given distance interval $[d_a, d_b)$ using the distance map D.
2. Compute the average

$$E_{d\in[d_a,d_b)}[f] = E[f(p)]$$

0.15.5 Demo isoline average



0.16 The thickness map

Thickness is a metric for assessing the size and structure of objects in a very generic manner. For every point $\vec{x}$ in the image you find the largest sphere which:

- contains that point
- is entirely contained within the object

The thickness map is quantifying the regional sizes compared to the distance map which describes distances from pixel to pixel. That is, in the thickness map all pixels belonging to a sphere that can be inscribed in the object will have the value of the sphere radius. There will be many overlapping spheres and it is always the largest radius that defines the pixel value.

In the comparison below, you can see that the distance map has a greater representation of short distances while the thickness map has large areas with great distances.

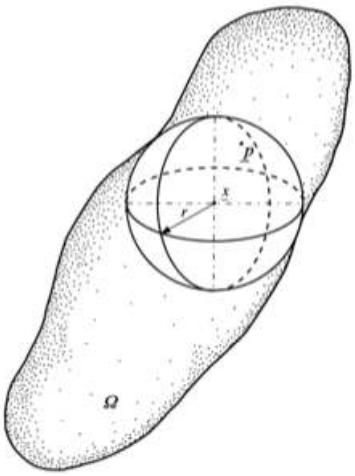


Fig. 1: The largest sphere inscribed in the object.

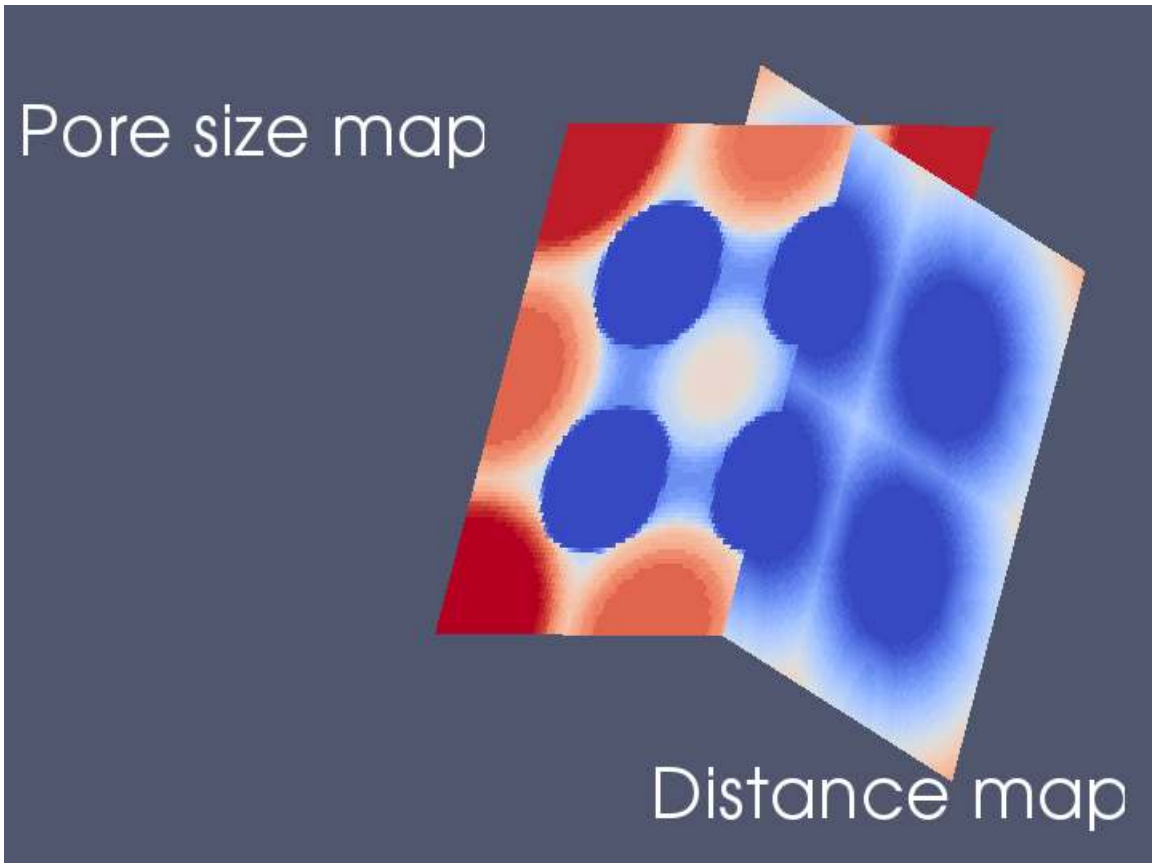


Fig. 2: Comparing the thickness map to the distance map.

The thickness map has many names. So, when you hear terms like pore radius map or pore size map, then it is all the same thing. One little difference would be that the thickness map indicates that you are measuring solid objects while the pore size map focuses on the voids between the objects.

Taken from Hildebrand 1997

- The image shows a typical object
- The sphere centered at point p with a radius r is the largest that fits

0.16.1 Applications

- Ideal for spherical and cylindrical objects since it shows their radius
- Also relevant for flow and diffusion since it can highlight bottlenecks in structure (then called pore radius map) Lehmann 2006

0.16.2 A thickness map implementation

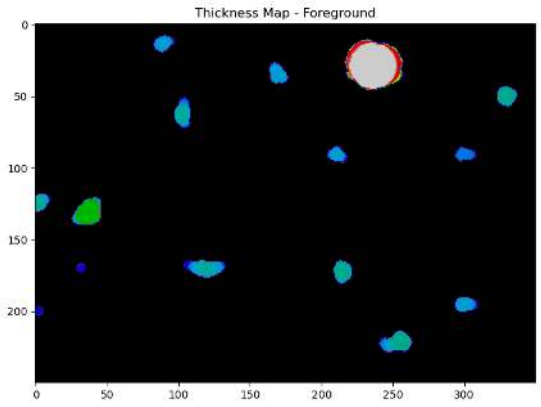
```
def simple_thickness_func(distance_map):
    dm = distance_map.ravel()
    th_map = distance_map.ravel().copy()
    xx, yy = [v.ravel() for v in np.meshgrid(range(distance_map.shape[1]),
                                             range(distance_map.shape[0]))]

    for idx in np.argsort(-dm):
        dval = dm[idx]
        if dval > 0:
            p_mask = (np.square(xx-xx[idx]) +
                      np.square(yy-yy[idx])) <= np.square(dval)
            p_mask &= th_map < dval
            th_map[p_mask] = dval
    th_map[dm == 0] = 0  # make sure zeros stay zero (rounding errors)
    return th_map.reshape(distance_map.shape)
```

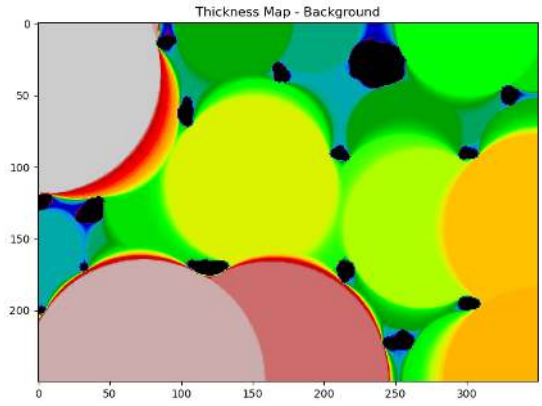
This is not an efficient implementation as it computes the distance from the current pixel to all pixels in the image. This is repeated for all foreground pixels. It would be smarter to only probe the pixels with the radius of the current pore. This would however lead to a more comlicated algorithm that would require higher level of detail to explain.

0.16.3 Thickness map on foregrund and background

```
fg_th_map = simple_thickness_func(fg_dmap)
bg_th_map = simple_thickness_func(bg_dmap)
```

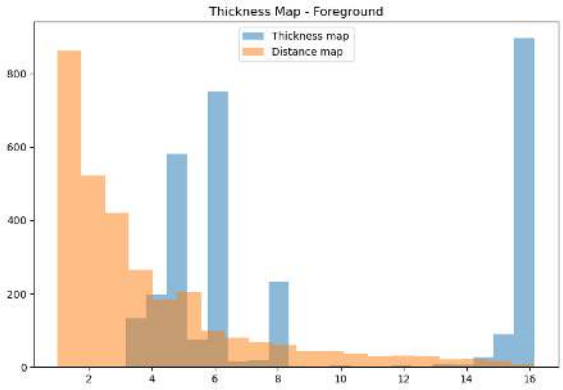
```
fig, (ax3, ax4) = plt.subplots(1, 2, figsize=(20, 6), dpi=100)

ax3.imshow(fg_th_map, cmap='nipy_spectral'); ax3.set_title('Thickness Map - Foreground
 ↪');
ax4.imshow(bg_th_map, cmap='nipy_spectral'); ax4.set_title('Thickness Map - Background
 ↪');
```

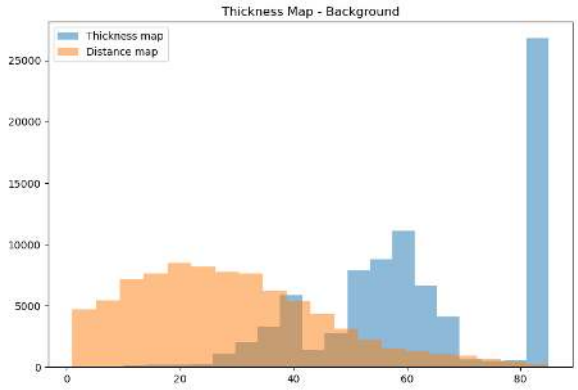




0.16.4 Thickness distributions

```
fig, (ax3, ax4) = plt.subplots(1, 2, figsize=(20, 6), dpi=100)

ax3.hist(fg_th_map[fg_th_map > 0].ravel(), 20,alpha=0.5, label='Thickness map'); ax3.
 ↪set_title('Thickness Map - Foreground');
ax3.hist(fg_dmap[fg_dmap > 0].ravel(),bins=20,alpha=0.5, label='Distance map');
ax3.legend()
ax4.hist(bg_th_map[bg_th_map > 0].ravel(), 20,alpha=0.5, label='Thickness map'); ax4.
 ↪set_title('Thickness Map - Background');
ax4.hist(bg_dmap[bg_dmap > 0].ravel(),bins=20,alpha=0.5, label='Distance map');
ax4.legend();
```





We see in these examples that the thickness map better represents the size distribution in the image. The distance transform puts too much emphasis on shorter to mid-range distances, which gives a misleading impression of the size distribution.

0.17 Distance Maps in 3D

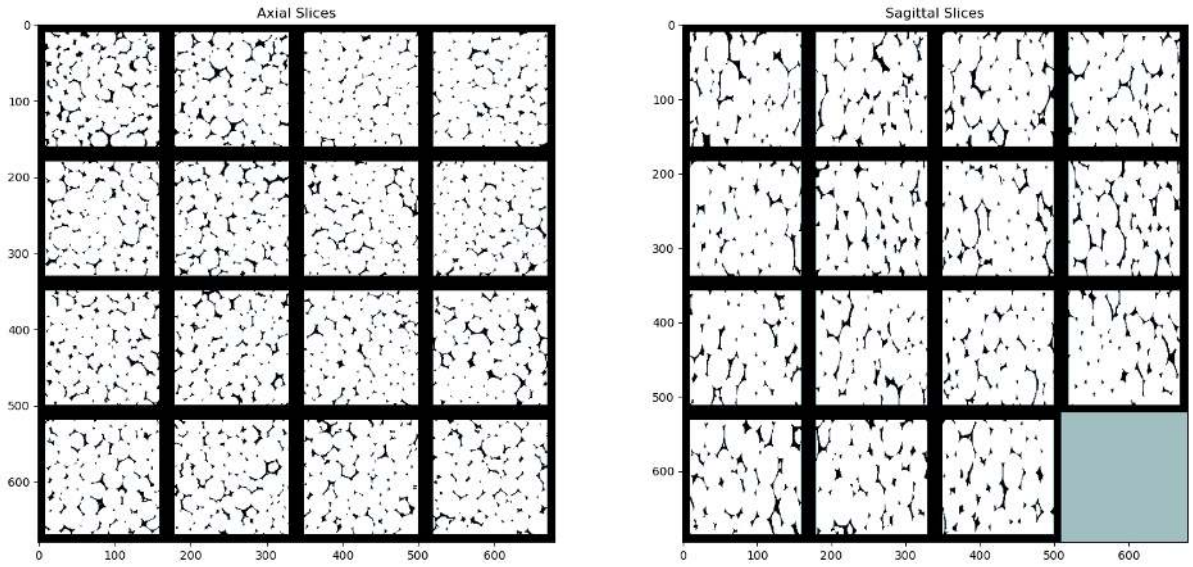
- Distance maps work in more than two dimensions and can be used in n-d problems.
- Beyond 3D requires serious thought about what the meaning of this distance is.

0.17.1 Let's load a 3D image

We need some data to demonstrate the distance transform of 3D images. This an image of oblatoid grains which are aligned with the major axis in the vertical direction.

```
def montage_pad(x): return montage2d(
    np.pad(x, [(0, 0), (10, 10), (10, 10)], mode='constant', constant_values=0))

bw_img = 255 - imread("data/plateau_border.tif")[:, 100:400, 100:400][:, ::2, ::2]
print('Loaded Image:', bw_img.shape, bw_img.max())
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(18, 8))
ax1.imshow(montage_pad(bw_img[::10]), cmap='bone');ax1.set_title('Axial Slices')
ax2.imshow(montage_pad(bw_img.swapaxes(0, 1)[::10]), cmap='bone'); ax2.set_title(
↪'Sagittal Slices');
```

```
Loaded Image: (154, 150, 150) 255
```



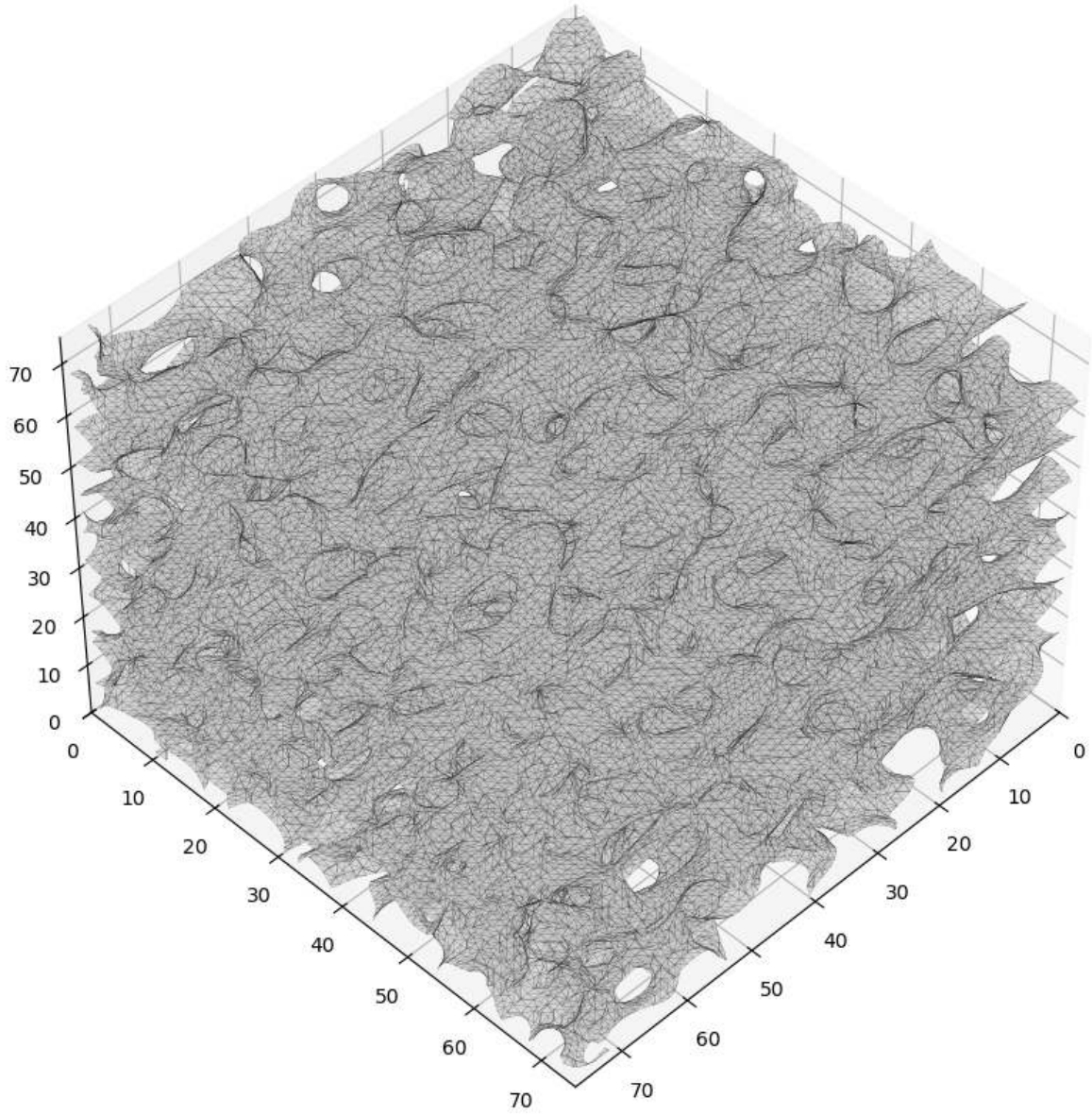
0.17.2 Inspecting the data in 3D using a surface mesh

```
def show_3d_mesh(image, thresholds, edgecolor='none', alpha=0.5):
    p = image[::-1].swapaxes(1, 2)
    cmap = plt.colormaps.get_cmap('nipy_spectral_r')
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111, projection='3d')
    for i, c_threshold in list(enumerate(thresholds)):
        verts, faces, _, _ = measure.marching_cubes(p, c_threshold)
```

(continues on next page)

(continued from previous page)

```
        mesh = Poly3DCollection(verts[faces], alpha=alpha, edgecolor=edgecolor,␣
↪linewidth=0.1)
        mesh.set_facecolor(cmap(i / len(thresholds))[:3])
        ax.add_collection3d(mesh)
    ax.set_xlim(0, p.shape[0]); ax.set_ylim(0, p.shape[1]); ax.set_zlim(0, p.shape[2])

    ax.view_init(45, 45)
    return fig


smooth_pt_img = gaussian(bw_img[::2, ::2, ::2]*1.0/bw_img.max(), 1.5)
show_3d_mesh(smooth_pt_img, [smooth_pt_img.mean()],
             alpha=0.75, edgecolor='black');
```



Rendering the mesh as a surface

```
help(p3.plot_isosurface)
```

```
Help on function plot_isosurface in module ipyvolume.pylab:

plot_isosurface(data, level=None, color='red', wireframe=True, surface=True,␣
↪controls=True, extent=None, description=None)
    Plot a surface at constant value (like a 2d contour).

    :param data: 3d numpy array
    :param float level: value where the surface should lie
    :param color: color of the surface, although it can be an array, the length is␣
↪difficult to predict beforehand,
                  if per vertex color are needed, it is better to set them on the␣
↪returned mesh afterwards.
    :param bool wireframe: draw lines between the vertices
    :param bool surface: draw faces/triangles between the vertices
    :param bool controls: add controls to change the isosurface
    :param extent: list of [[xmin, xmax], [ymin, ymax], [zmin, zmax]] values that␣
↪define the bounding box of the mesh,
                   otherwise the viewport is used
    :param description: Used in the legend and in popup to identify the object
    :return: :any:`Mesh`
```

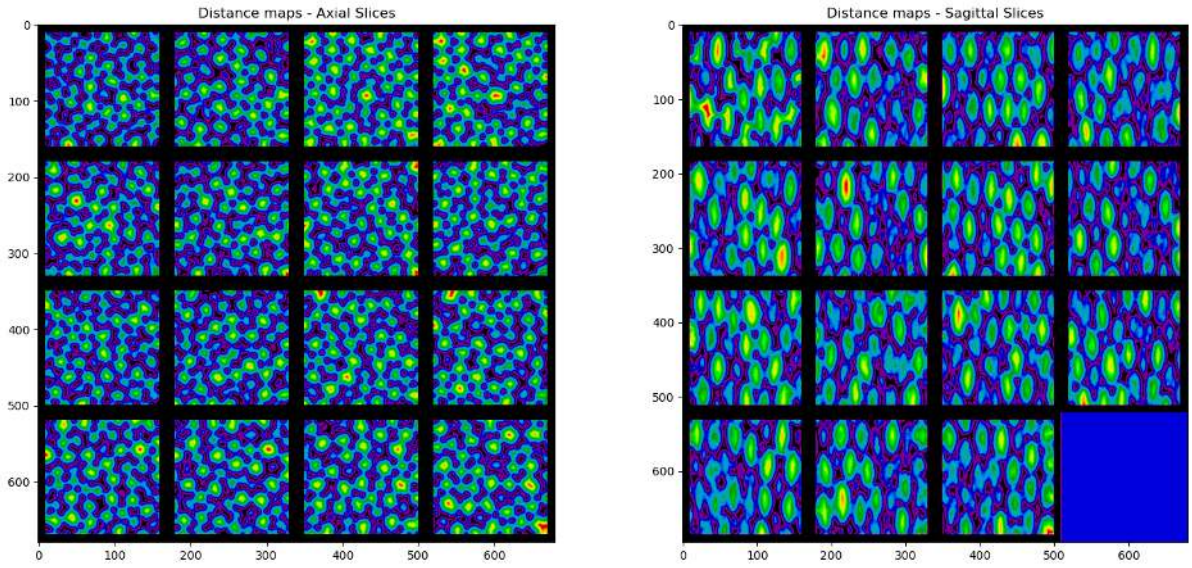
```
fig = p3.figure()
# create a custom LUT
temp_tf = plt.cm.nipy_spectral(np.linspace(0, 1, 256))
# make transparency more aggressive
temp_tf[:, 3] = np.linspace(-.3, 0.5, 256).clip(0, 1)
tf = p3.transferfunction.TransferFunction(rgba=temp_tf)
p3.plot_isosurface((smooth_pt_img/smooth_pt_img.max()).astype(np.float32), level=0.85)

p3.show()
```

```
Container(children=[HBox(children=(FloatSlider(value=0.85, max=0.9999726414680481,␣
↪min=0.6454506587982177, ste…
```

0.17.3 Look at the distance transform

```
bubble_dist = distance_transform_edt(bw_img)
```

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(18, 8))
ax1.imshow(montage_pad(bubble_dist[::10]), cmap='nipy_spectral')
ax1.set_title('Distance maps - Axial Slices')
ax2.imshow(montage_pad(bubble_dist.swapaxes(0, 1)[::10]), cmap='nipy_spectral')
ax2.set_title('Distance maps - Sagittal Slices');
```



0.18 Thickness in 3D Images

While the examples and demonstrations so far have been shown in 2D, the same exact technique can be applied to 3D data as well. For example for this liquid foam structure

- The thickness can be calculated of the background (air) voxels in the same manner.
- With care, this can be used as a proxy for bubble size distribution in systems where all the bubbles are connected to it is difficult to identify single ones.

0.18.1 An algorithm to create the 3D thickness map

```
from skimage.feature import peak_local_max
bubble_candidates = peak_local_max(bubble_dist, min_distance=12)
print('Found', len(bubble_candidates), 'bubbles')

thickness_map = np.zeros(bubble_dist.shape, dtype=np.float32)
xx, yy, zz = np.meshgrid(np.arange(bubble_dist.shape[1]),
                         np.arange(bubble_dist.shape[0]),
                         np.arange(bubble_dist.shape[2])
                         )
# sort candidates by size
sorted_candidates = sorted(
    bubble_candidates, key=lambda xyz: bubble_dist[tuple(xyz)])
for label_idx, (x, y, z) in enumerate(sorted_candidates):
    cur_bubble_radius = bubble_dist[x, y, z]
    cur_bubble = ((xx-float(y))**2 +
                  (yy-float(x))**2 +
                  (zz-float(z))**2) <= cur_bubble_radius**2
    thickness_map[cur_bubble] = cur_bubble_radius
```

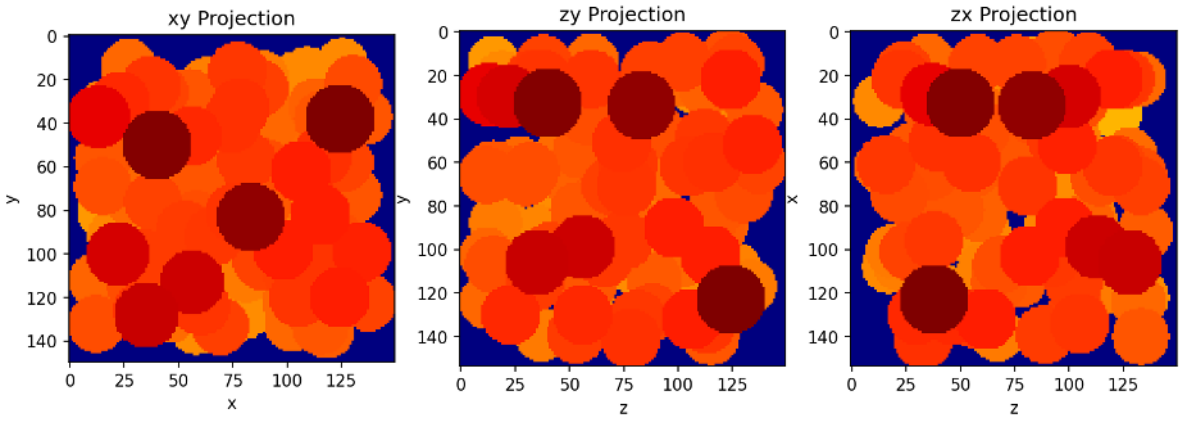
```
Found 98 bubbles
```

This algorithm is simplified to speed up the 3D case. It locates the distance peaks of each pore and draws a sphere for each. This approach works under the assumption that the pores are perfect spheres.

0.18.2 Visualizing the 3D thickness map - max projection

Visualizing 3D information on a 2D screen is a challenge. Here, we use the max projection from in the three principal directions.

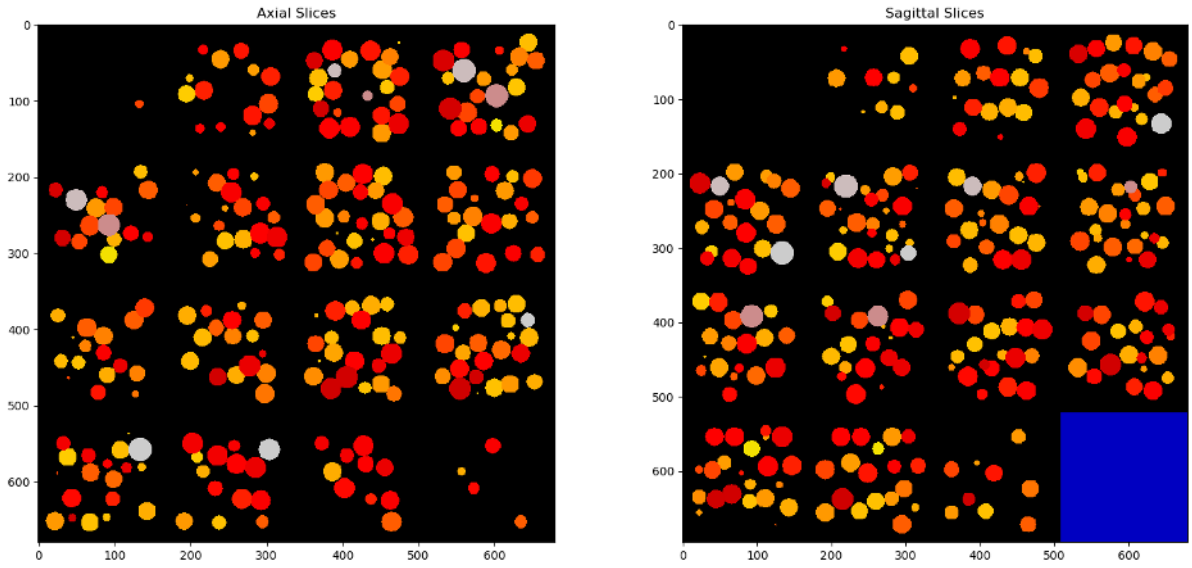
```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 4), dpi=150)
for i, (cax, clabel) in enumerate(zip([ax1, ax2, ax3], ['xy', 'zy', 'zx'])):
    cax.imshow(np.max(thickness_map, i).squeeze(),
               interpolation='none', cmap='jet')
    cax.set(title='{} Projection'.format(clabel), xlabel=clabel[0], ylabel=clabel[1]);
```



0.18.3 Visualizing the 3D thickness map - mosaic of slices

The max projection looses some information. Showing a selection of slices as a mosaic is sometimes better.

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(18, 8),dpi=100)
ax1.imshow(montage_pad(thickness_map[::10]), cmap='nipy_spectral',interpolation="None
 ↪")
ax1.set_title('Axial Slices')
ax2.imshow(montage_pad(thickness_map.swapaxes(
    0, 1)[::10]), cmap='nipy_spectral',interpolation="None")
ax2.set_title('Sagittal Slices');
```



0.19 Interactive 3D Views

Here we can show the thickness map in an interactive 3D manner using the ipyvolume tools.

```
fig = p3.figure()
# create a custom LUT
temp_tf = plt.cm.nipy_spectral(np.linspace(0, 1, 256))
# make transparency more aggressive
temp_tf[:, 3] = np.linspace(-.3, 0.5, 256).clip(0, 1)
tf = p3.transferfunction.TransferFunction(rgba=temp_tf)

p3.volshow((thickness_map[::2, ::2, ::2]/thickness_map.max()).astype(np.float32),
           lighting=True,
           max_opacity=0.5,
           tf=tf,
           controls=True)

p3.show()
p3.save('bubbles.html')
```

```
/opt/anaconda3/envs/qbi2025/lib/python3.9/site-packages/ipyvolume/serialize.
↪py:102: RuntimeWarning: invalid value encountered in cast
  subdata[..., i] = ((gradient[i][zindex] / 2.0 + 0.5) * 255).astype(np.uint8)
```

```
Container(children=[HBox(children=(FloatLogSlider(value=1.0, description='opacity',
↪ max=2.0, min=-2.0), FloatL…
```

0.20 Interfaces / Surfaces

Many physical and chemical processes occur at surfaces and interfaces and consequently these structures are important in material science and biology. For this lecture surface and interface will be used interchangebly and refers to the boundary between two different materials (calcified bone and soft tissue, magma and water, liquid and gas) Through segmentation we have identified the unique phases in the sample under investigation.

- Segmentation identifying volumes (3D) or areas (2D)
- Interfaces are one dimension lower corresponding to surface area (3D) or perimeter (2D)
- Interfaces are important for
- connectivity of cell networks, particularly neurons
- material science processes like coarsening or rheological behavior
- chemical processes (surface-bound diffusion, catalyst action)

0.21 Surface Area / Perimeter

We see that the dilation and erosion affects are strongly related to the surface area of an object:

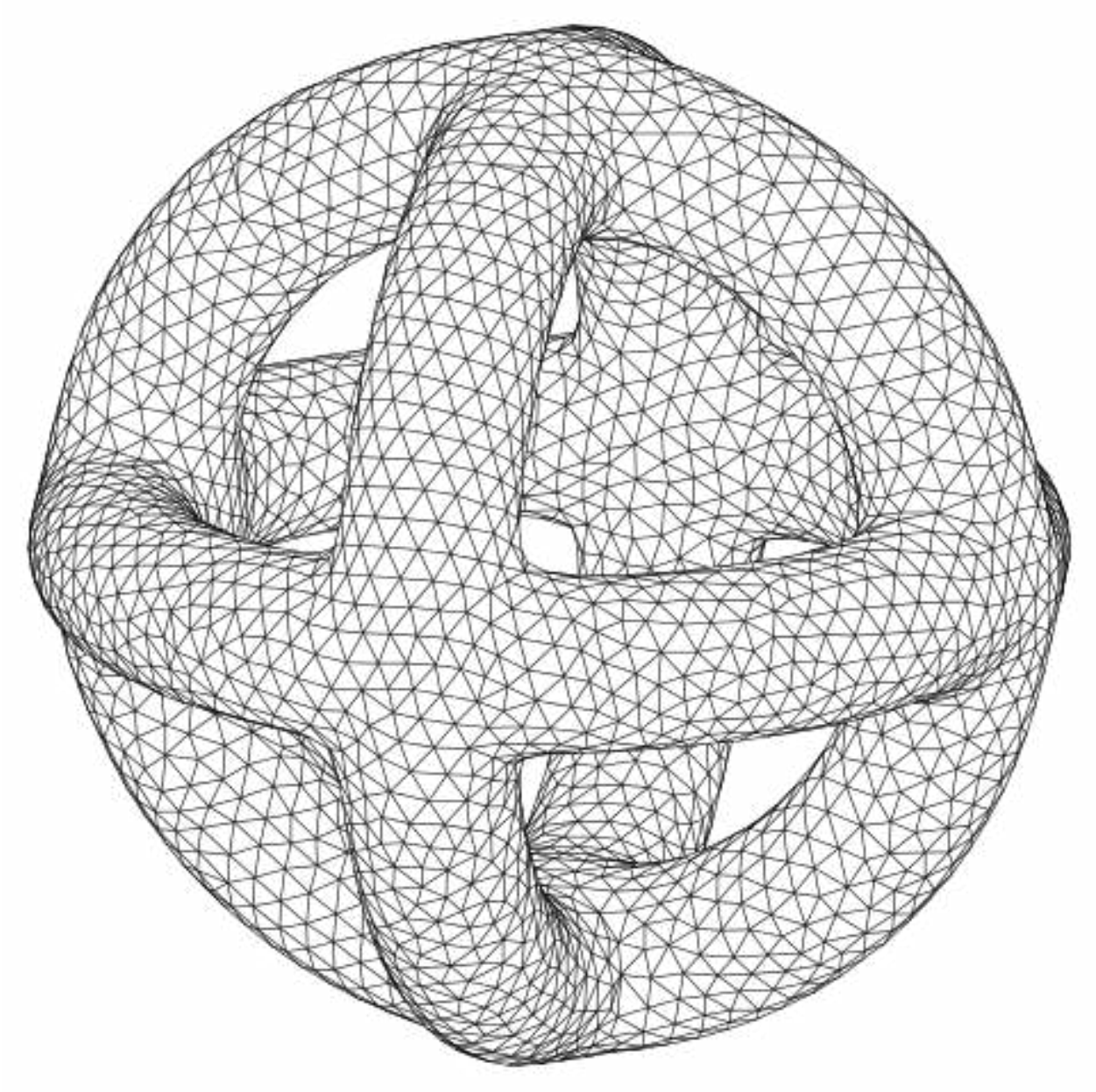
- the more surface area
- the larger the effect of a single dilation or erosion step.

```
d=np.linspace(-20,20,41)
x,y = np.meshgrid(d,d)
R=15
disk = (x**2+y**2)<R**2
```

0.22 Meshing

The process of turning a (connected) set of pixels into a list of vertices and edges

- For these vertices and edges we can define forces.
- Most crucially this comes when looking at physical processes like deformation.
- Meshes are also useful for visualization.



Example

Looking at stress-strain relationships in mechanics using Hooke's Model

$$\vec{F} = k(\vec{x}_0 - \vec{x})$$

the force needed to stretch one of these edges is proportional to how far it is stretched.

0.22.1 Meshing of images

Since we uses voxels to image and identify the volume we can use the voxels themselves as an approimation for the surface of the structure.

- Each 'exposed' face of a voxel belongs to the surface

From this we can create a mesh by

- adding each exposed voxel face to a list of surface squares.
- adding connectivity information for the different squares (shared edges and vertices)

A wide variety of methods of which we will only graze the surface (http://en.wikipedia.org/wiki/Image-based_meshing)

0.22.2 Marching Cubes

A well-known algorithm to determine a surface

- Maps 2x2 or 2x2x2 pixel combinations to surface elements
- Uses a look-up table to.
- Can be used with thresholding

Why

Voxels are very poor approximations for the surface and are very rough (they are either normal to the x, y, or z axis and nothing between).

Because of their inherently orthogonal surface normals, any analysis which utilizes the surface normal to calculate another value (growth, curvature, etc) is going to be very inaccurate at best and very wrong at worst.

How (https://en.wikipedia.org/wiki/Marching_cubes)

The image is processed one voxel at a time and the neighborhood (not quite the same is the morphological definition) is checked at every voxel. From this configuration of values, faces are added to the mesh to incorporate the most simple surface which would explain the values.

1. The 2x2x2 neighborhood is checked at every voxel.
2. Different faces are assigned depending on the neighborhood configuration.

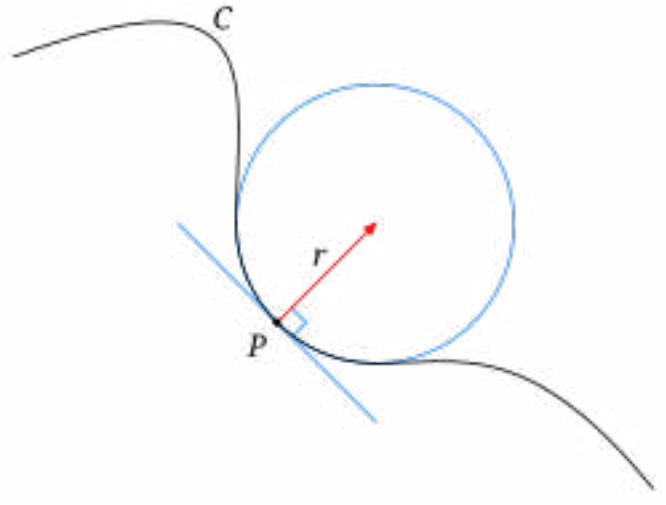
This algortihm is nicely explained in this video

Marching tetrahedra is for some applications a better suited approach

0.23 Curvature

Curvature is a metric related to the surface or interface between phases or objects.

- It is most easily understood in its 1D sense or being the radius of the circle that matchs the local shape of a curve



- Mathematically it is defined as

$$\kappa = \frac{1}{R}$$

- Thus a low curvature means the value means a very large radius
- and high curvature means a very small radius

0.23.1 Curvature: Surface Normal

In order to meaningfully talk about curvatures of surfaces, we first need to define a consistent frame of reference for examining the surface of an object.

We thus define a surface normal vector as a vector oriented orthogonally to the surface away from the interior of the object $\rightarrow \vec{N}$

0.23.2 Curvature in 3D

With the notion of surface normal defined ($\vec{N}$), we can define many curvatures at point $\vec{x}$ on the surface.

- This is because there are infinitely many planes which contain both point $\vec{x}$ and $\vec{N}$
- More generally we can define an angle θ about which a single plane containing both can be freely rotated
- We can then define two principal curvatures by taking the maximum and minimum of this curve

$$\kappa_1 = \max(\kappa(\theta))$$

$$\kappa_2 = \min(\kappa(\theta))$$

Mean Curvature

The mean of the two principal curvatures $H = \frac{1}{2}(\kappa_1 + \kappa_2)$

Gaussian Curvature

The product of the two principal curvatures $K = \kappa_1\kappa_2$

- positive for spheres (or spherical inclusions)
- curvatures agree in sign
- negative for saddles (hyperboloid surfaces)
- curvatures disgree in sign
- 0 for planes

0.23.3 Curvature: 3D Examples

Examining a complex structure with no meaningful ellipsoidal or watershed model.

The images themselves show the type of substructures and shapes which are present in the sample.

- the comparative amount of surface at, above, and below 0
- from spherical particles into annealed mesh of balls

0.23.4 Characteristic Shape

Characteristic shape can be calculated by

- measuring principal curvatures and normalizing them by scaling to the structure size.
- A distribution of these curvatures then provides shape information on a structure indepedent of the size.

For example a structure transitioning from a collection of perfectly spherical particles to a annealed solid will go

- **from** having many round spherical faces with positive gaussian curvature
- **to** many saddles and more complicated structures with 0 or negative curvature.

0.23.5 Curvature: Take Home Message

It provides another metric for characterizing complex shapes

- Particularly useful for examining interfaces
- Folds,
- saddles,
- and many other types of points that are not characterized well with ellipsoids or thickness maps
- Provides a scale-free metric for assessing structures
- Can provide visual indications of structural changes

0.24 Other Techniques

There are hundreds of other techniques which can be applied to these complicated structures, but they go beyond the scope of this course.

Many of them are model-based which means they work well but only for particular types of samples or images.

Of the more general techniques several which are easily testable inside of FIJI are

- Directional Analysis = Looking at the orientation of different components using Fourier analysis (*Analyze* → *Directionality*)
- Tubeness / Surfaceness (*Plugins* → *Analyze* →) characterize binary images and the shape at each point similar to curvature but with a different underlying model

0.24.1 References to other techniques

- Fractal Dimensionality = A metric for assessing the structure as you scale up and down by examining various spatial relationships
- Ma, D., Stoica, A. D., & Wang, X.-L. (2009). Power-law scaling and fractal nature of medium-range order in metallic glasses. Nature Materials, 8(1), 30–4. doi:10.1038/nmat2340
- Two (or more) point correlation functions = Used in theoretical material science and physics to describe random materials and can be used to characterize distances, orientations, and organization in complex samples
- Jiao, Y., Stillinger, F., & Torquato, S. (2007). Modeling heterogeneous materials via two-point correlation functions: Basic principles. Physical Review E, 76(3). doi:10.1103/PhysRevE.76.031110
- Andrey, P., Kiêu, K., Kress, C., Lehmann, G., Tirichine, L., Liu, Z., … Debey, P. (2010). Statistical analysis of 3D images detects regular spatial distributions of centromeres and chromocenters in animal and plant nuclei. PLoS Computational Biology, 6(7), e1000853. doi:10.1371/journal.pcbi.1000853
- Haghpanahi, M., & Miramini, S. (2008). Extraction of morphological parameters of tissue engineering scaffolds using two-point correlation function, 463–466. Retrieved from http://portal.acm.org/citation.cfm?id=1713360.1713456

0.25 Texture Analysis

The world is full of patterns (aka Textures)

Example taken from

Some metrics to characterize textures Haralick texture features and DOI

0.25.1 Analyzing textures

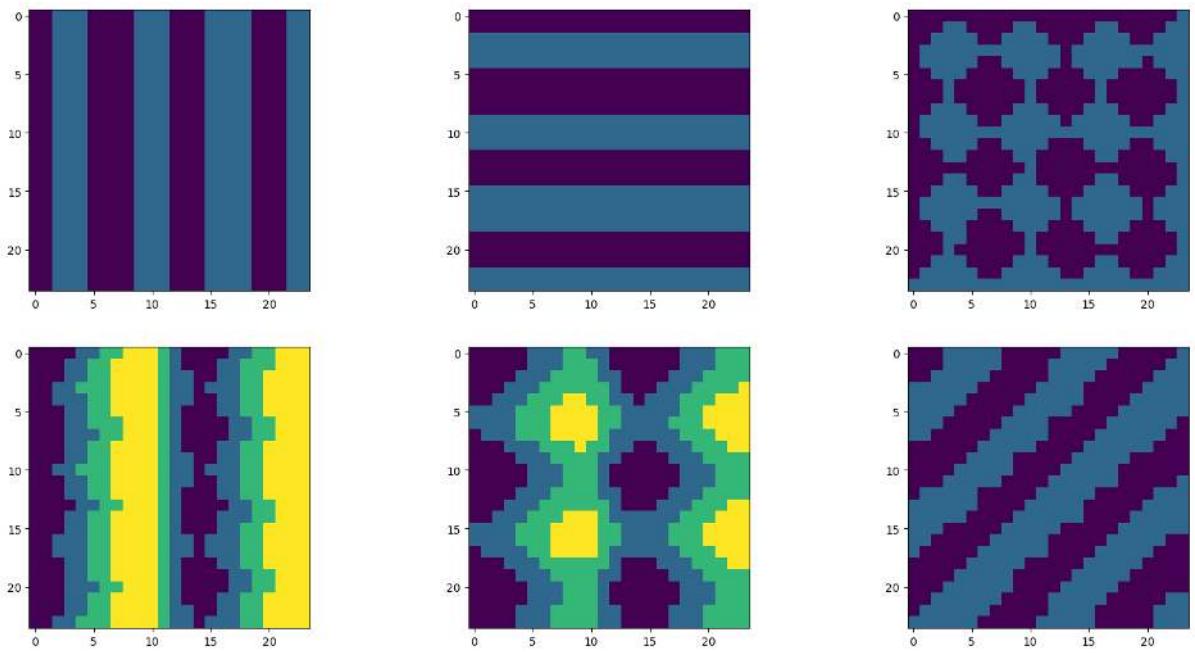
- Textures are spatial patterns
- The analysis must be done in regions or ranges
- The resolution suffers

Method Type	Technique	Captures
Statistical	GLCM, LBP	Spatial relationships, patterns
Frequency-based	FFT, Gabor, Wavelets	Periodicity, orientation, scales
Model-based	AR Models, Fractals	Structural or roughness patterns
Morphological	Granulometry, Profiles	Shape size distribution
Edge-based	Edge density, Hough	Repetitive structures, line textures

0.25.2 Sample Textures

Let's create some sample textures

```
x, y = np.meshgrid(range(8), range(8))

def blur_img(c_img): return (ndimage.zoom(c_img.astype('float'),
                                          3,
                                          order=3,
                                          prefilter=False)*4).astype(int).clip(1, 4)-1

text_imgs = [blur_img(c_img)
             for c_img in [x % 2,
                           y % 2,
                           (x % 2+y % 2)/2.0,
                           (x % 4+y % 2)/2.5,
                           (x % 4+y % 3)/3.5,
                           ((x+y) % 3)/2.0]]

fig, m_axs = plt.subplots(2, 3, figsize=(20, 10))

for c_ax, c_img in zip(m_axs.flatten(), text_imgs):
    c_ax.imshow(c_img,cmap='viridis', vmin=0, vmax=3)
```



The Gray-Level Co-Occurrence Matrix (GLCM)

Measures how often pairs of pixel intensities occur at a given distance and angle.

Extract texture features like:

- Contrast
- Correlation
- Energy
- Homogeneity
- ASM (Angular Second Moment)

Defining the Co-occurrence matrix

$$C_{\Delta x,\Delta y}(i,j) = \sum_{x=1}^{n}\sum_{y=1}^{m}\begin{cases}1, & \text{if } I(x,y)=i \text{ and } I(x+\Delta x, y+\Delta y)=j\\ 0, & \text{otherwise}\end{cases}$$

As text →

- for every possible offset $\vec{r}$ = (Δx, Δy) given the current intensity (i) how likely is it that value at a point offset by that amount is j.
- This is similar to the two point correlation function but is a 4D representation instead of just a 2D
- In order to use this we need to discretize the image into bins

```
def montage_nd(in_img):
    if len(in_img.shape) > 3:
        return montage2d(np.stack([montage_nd(x_slice) for x_slice in in_img], 0))
    elif len(in_img.shape) == 3:
        return montage2d(in_img)
    else:
        warn('Input less than 3d image, returning original', RuntimeWarning)
        return in_img


dist_list  = np.linspace(1, 6, 15)
angle_list = np.linspace(0, 2*np.pi, 15)

def calc_coomatrix(in_img):
    return graycomatrix(image=in_img,
                        distances=dist_list,
                        angles=angle_list,
                        levels=4)

def coo_tensor_to_df(x): return pd.DataFrame(
    np.stack([x.ravel()]+[c_vec.ravel() for c_vec in np.meshgrid(range(x.shape[0]),
                                                                 range(x.shape[1]),
                                                                 dist_list,
                                                                 angle_list,
                                                                 indexing='xy')], -1),
    columns=['E', 'i', 'j', 'd', 'theta'])


coo_tensor_to_df(calc_coomatrix(text_imgs[0])).sample(5)
```

```
          E    i    j         d     theta
796     0.0  3.0  0.0  3.857143  0.448799
1628    0.0  3.0  1.0  2.071429  3.590392
2485    0.0  3.0  2.0  1.000000  4.487990
1461    0.0  2.0  1.0  3.500000  2.692794
1096  110.0  0.0  1.0  5.642857  0.448799
```

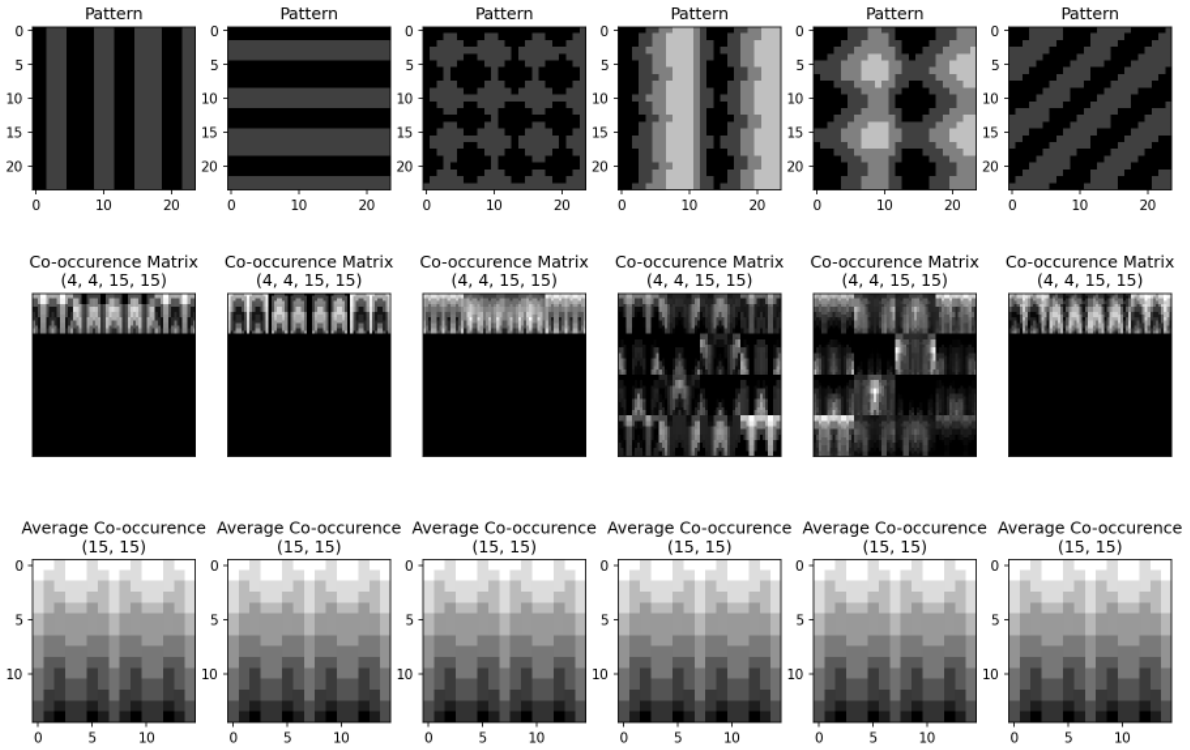
Co-occurence matrices for the test patterns

Using

- 4 levels
- 15 Distances
- 15 Directions

  The value P[i,j,d,theta] is the number of times that gray-level j occurs at a distance d and at an angle theta from gray-level i.

```
fig, m_axs = plt.subplots(3, 6, figsize=(15, 10), dpi=150)

for (c_ax, d_ax, f_ax), c_img in zip(m_axs.T, text_imgs):
    c_ax.imshow(c_img, vmin=0, vmax=4, cmap='gray'); c_ax.set_title('Pattern')
    full_coo_matrix = calc_coomatrix(c_img)
    d_ax.imshow(montage_nd(full_coo_matrix), cmap='gray')
    d_ax.set(title='Co-occurence Matrix\n{}'.format(full_coo_matrix.shape),xticks=[],
 ↪yticks=[])
    avg_coo_matrix = np.mean(full_coo_matrix*1.0, axis=(0, 1))
    f_ax.imshow(avg_coo_matrix, cmap='gray')
    f_ax.set_title('Average Co-occurence\n{}'.format(avg_coo_matrix.shape))
```



Here, you can see that the bi-level images only have a single row in the montage. This is because there is no additional co-occurance for greater gray level differences.

0.25.3 Simple Correlation

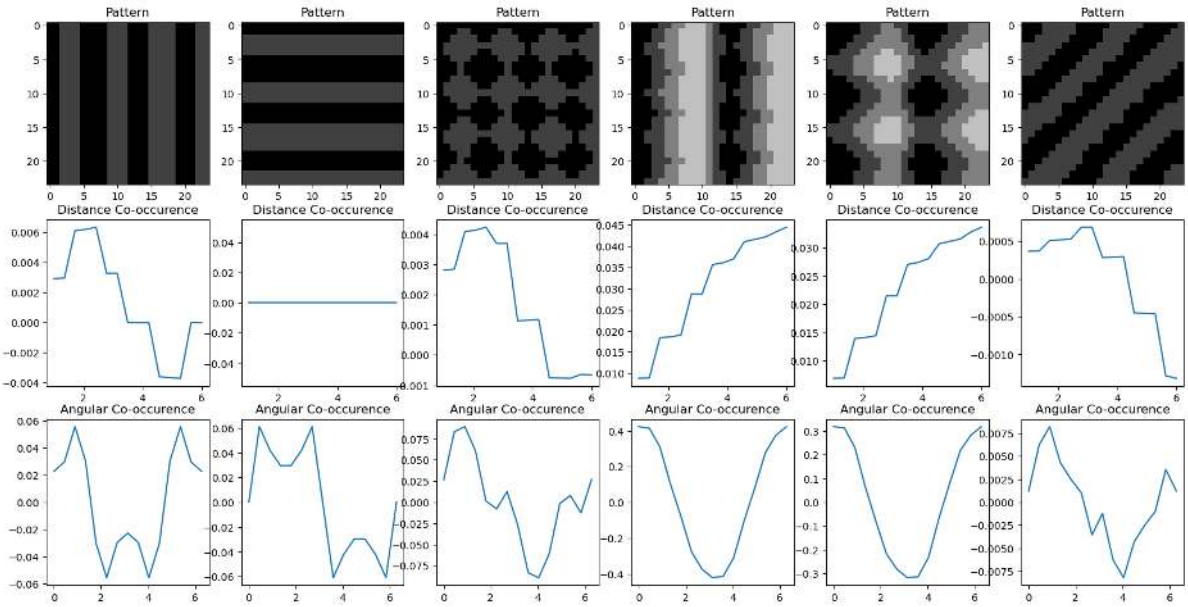
Using the mean difference ($E[i-j]$) instead of all of the i,j pair possiblities

```
text_df = coo_tensor_to_df(calc_coomatrix(text_imgs[0]))
text_df['ij_diff'] = text_df.apply(lambda x: x['i']-x['j'], axis=1)

simple_corr_df = text_df.groupby(['ij_diff', 'd', 'theta']).agg({'E': 'mean'}).reset_
↪index()
simple_corr_df.sample(5).round(decimals=3)
```

	ij_diff	d	theta	E
725	0.0	2.071	2.244	88.000
1249	2.0	3.857	1.795	0.000
598	-1.0	4.214	5.834	58.667
1164	2.0	1.714	4.039	0.000
137	-3.0	4.214	0.898	0.000

Inspecting the mean difference

```
def grouped_weighted_avg(values, weights, by):
    return (values * weights).groupby(by).sum() / weights.groupby(by).sum()


fig, m_axs = plt.subplots(3, 6, figsize=(20, 10))

for (c_ax, d_ax, f_ax), c_img in zip(m_axs.T, text_imgs):
    c_ax.imshow(c_img, vmin=0, vmax=4, cmap='gray')
    c_ax.set_title('Pattern')
    full_coo_matrix = calc_coomatrix(c_img)
    text_df = coo_tensor_to_df(full_coo_matrix)
    text_df['ij_diff'] = text_df.apply(lambda x: x['i']-x['j'], axis=1)

    simple_corr_df = text_df.groupby(['ij_diff', 'd', 'theta']).agg({
        'E': 'sum'}).reset_index()
    gwa_d = grouped_weighted_avg(
        simple_corr_df.ij_diff, simple_corr_df.E, simple_corr_df.d)
    d_ax.plot(gwa_d.index, gwa_d.values)
    d_ax.set_title('Distance Co-occurence')

    gwa_theta = grouped_weighted_avg(
        simple_corr_df.ij_diff, simple_corr_df.E, simple_corr_df.theta)
    f_ax.plot(gwa_theta.index, gwa_theta.values)
    f_ax.set_title('Angular Co-occurence')
```

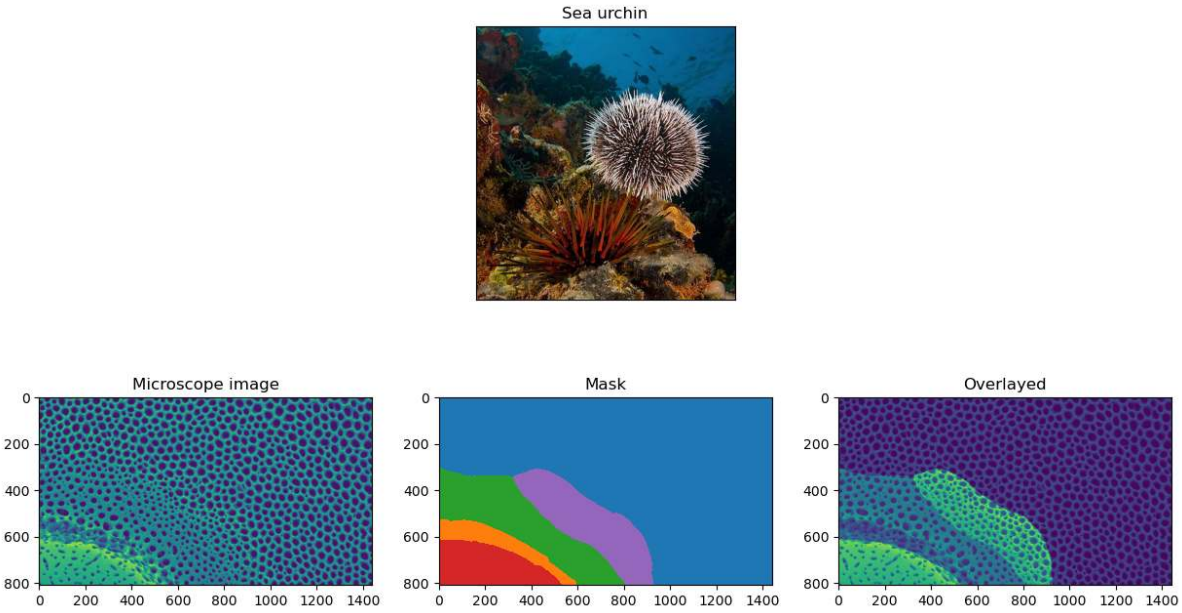


0.25.4 Applying texture analysis to microscope image

We want to segment different regions in a microscope image from a sea urchin

```
photo = io.imread('figures/Tripneustes_ventricosus.jpg')
img   = io.imread('data/sea-urchin-skeleton.jpg').mean(axis=2).astype(int)
mask  = io.imread('data/sea-urchin-skeleton_feat.png')
```

```
fig, ax = plt.subplots(2,3,figsize=(15,8))
mask[0,0]=10
ax=ax.ravel()
ax[0].axis('off')
ax[1].imshow(photo)
ax[1].set(title="Sea urchin",xticks=[],yticks=[])
ax[2].axis('off')
ax[3].imshow(img,cmap='viridis')
ax[3].set_title('Microscope image')
ax[4].imshow(mask,cmap='tab10',interpolation='none')
ax[4].set_title('Mask');
ax[5].imshow((20*mask+50)*img,cmap='viridis')
ax[5].set_title('Overlayed');
```



The regions are among others characterized by the pore shape and size distributions.

Let's try texture analysis!

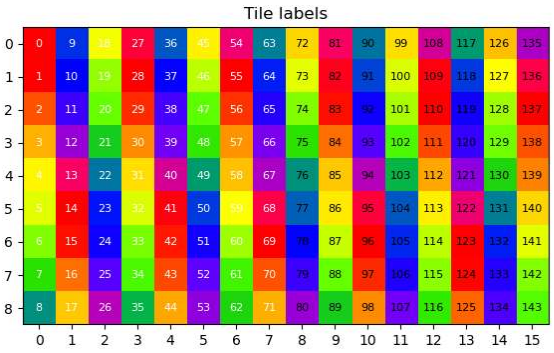
Analysis workflow

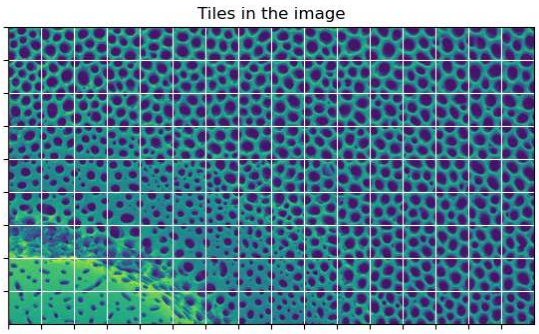
0.25.5 Tiling the image

Here we divide the image up into unique tiles for further processing. Each tile is 48×48 pixels.

```
xx, yy = np.meshgrid(
    np.arange(img.shape[1]),
    np.arange(img.shape[0]))
N=90 # one of [ 1,  2,  3,  5,  6,  9, 10, 15, 18, 30, 45, 90]
M=810/N
region_labels = (xx//N) * M+yy//N
region_labels = region_labels.astype(int)
```

```
fig,ax = plt.subplots(1,2,figsize=(15,7))

ps.heatmap(region_labels[::N, ::N].astype(int),precision=0,cmap="prism", fontsize=8,
 ↪ax=ax[0],bgbox=False)

ax[0].set_title('Tile labels')

ax[1].imshow(img,cmap='viridis')
ax[1].set(xticks=(np.arange(0,img.shape[1],N)),yticks=(np.arange(0,img.shape[0],N)),
          xticklabels=[],yticklabels=[],
          title='Tiles in the image')
ax[1].grid(visible=True, which='major', axis='both',color='white')
```





Gray-level colocation properties

```
grayco_prop_list = ['contrast',
                    'dissimilarity',
                    'homogeneity',
                    #'energy',    # = sqrt(ASM) ... skipped
                    'correlation',
                    'ASM']
```

Compute the feature vectors

For each tile compute

- Gray-level co-occurance matrix
- Average label
- Mean and standard deviation of gray-level
- Gray-level co-occurance propoerties

and store in a dictionary.

The code

```
# Initialization
prop_imgs = {}
for c_prop in grayco_prop_list:
    prop_imgs[c_prop] = np.zeros_like(img, dtype=np.float32)
score_img = np.zeros_like(img, dtype=np.float32)
out_df_list = []

# For each tile... compute colocation matrix and properties
for patch_idx in np.unique(region_labels):
    xx_box, yy_box = np.where(region_labels == patch_idx)

    glcm = graycomatrix(img[xx_box.min():xx_box.max(),yy_box.min():yy_box.max()],
                        [4,8,16], np.linspace(0,np.pi,6), 256, symmetric=False,␣
 ↪normed=True)

    mean_score = np.round(np.mean(mask[region_labels == patch_idx]))
```

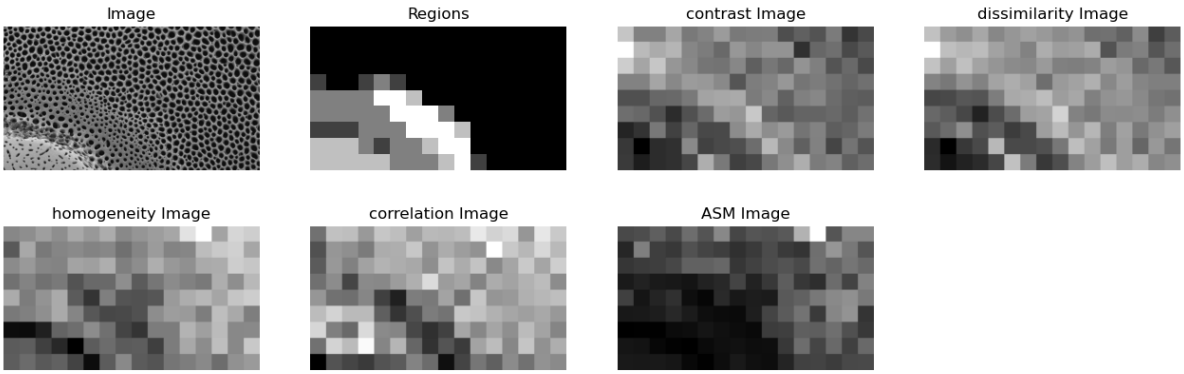
(continues on next page)

(continued from previous page)

```
    score_img[region_labels == patch_idx] = mean_score

    out_row = dict(
        intensity_mean = img[region_labels == patch_idx].mean(),
        intensity_std  = img[region_labels == patch_idx].std(),
        score          = mean_score)

    for c_prop in grayco_prop_list:
        out_row[c_prop] = graycoprops(glcm, c_prop)[0, 0]
        prop_imgs[c_prop][region_labels == patch_idx] = out_row[c_prop]

    out_df_list += [out_row];
```

The resulting feature images

```
# show the slice and threshold
fig, m_axs = plt.subplots(2, 4, figsize=(16, 5))
ax = m_axs.flatten()
ax[0].imshow(img, cmap='gray')
ax[0].axis('off')
ax[0].set_title('Image')
ax[1].imshow(score_img, cmap='gray',interpolation='none')
ax[1].axis('off')
ax[1].set_title('Regions')
for c_ax, c_prop in zip(ax[2:], grayco_prop_list):
    c_ax.imshow(prop_imgs[c_prop], cmap='gray')
    c_ax.axis('off')
    c_ax.set_title('{} Image'.format(c_prop))

ax[7].axis('off');
```



Summarizing the statistics

```
out_df = pd.DataFrame(out_df_list)
out_df.describe()
```

```
       intensity_mean  intensity_std       score     contrast  dissimilarity  \
count      144.000000     144.000000  144.000000   144.000000     144.000000
mean       124.780106      39.216082    0.888889   837.406576      19.228806
std         14.221951       4.056652    1.343782   163.129731       2.455831
min        109.707654      26.468436    0.000000   368.159022      11.707204
25%        116.853920      38.061432    0.000000   736.890218      17.876735
50%        120.748765      40.187202    0.000000   844.649570      19.582353
75%        126.503488      41.980307    2.000000   924.919002      20.876570
max        181.460617      45.414126    4.000000  1409.422340      26.451553

       homogeneity  correlation         ASM
count   144.000000   144.000000  144.000000
mean      0.134878     0.725910    0.002449
std       0.025904     0.049160    0.001557
min       0.058498     0.574036    0.000219
25%       0.119806     0.698575    0.001119
50%       0.136841     0.731594    0.002368
75%       0.152959     0.762922    0.003473
max       0.206492     0.834216    0.009360
```

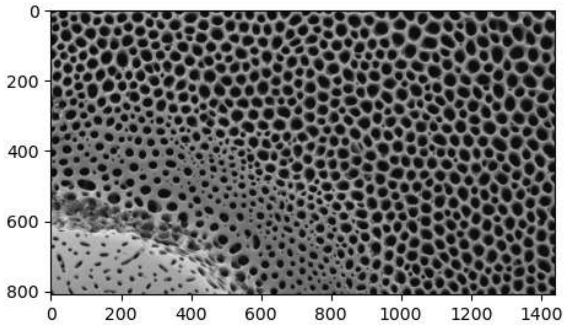
Let's try some clustering

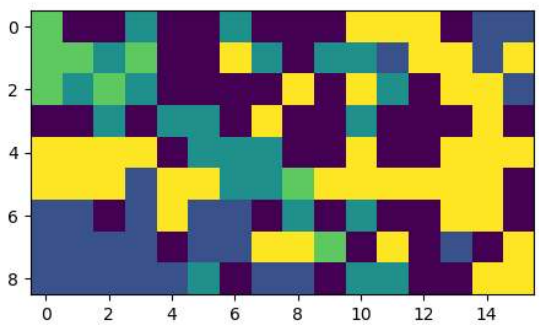
We use K-Means with 5 classes

```
from sklearn.cluster import KMeans

km = KMeans(5)

km.fit(out_df)
```

```
KMeans(n_clusters=5)
```

```
fig,ax=plt.subplots(1,2,figsize=(12,3))
ax[0].imshow(img,cmap='gray')
ax[1].imshow(np.reshape(km.labels_,(region_labels.shape[1]//N,region_labels.shape[0]//
 ↪N)).transpose());
```

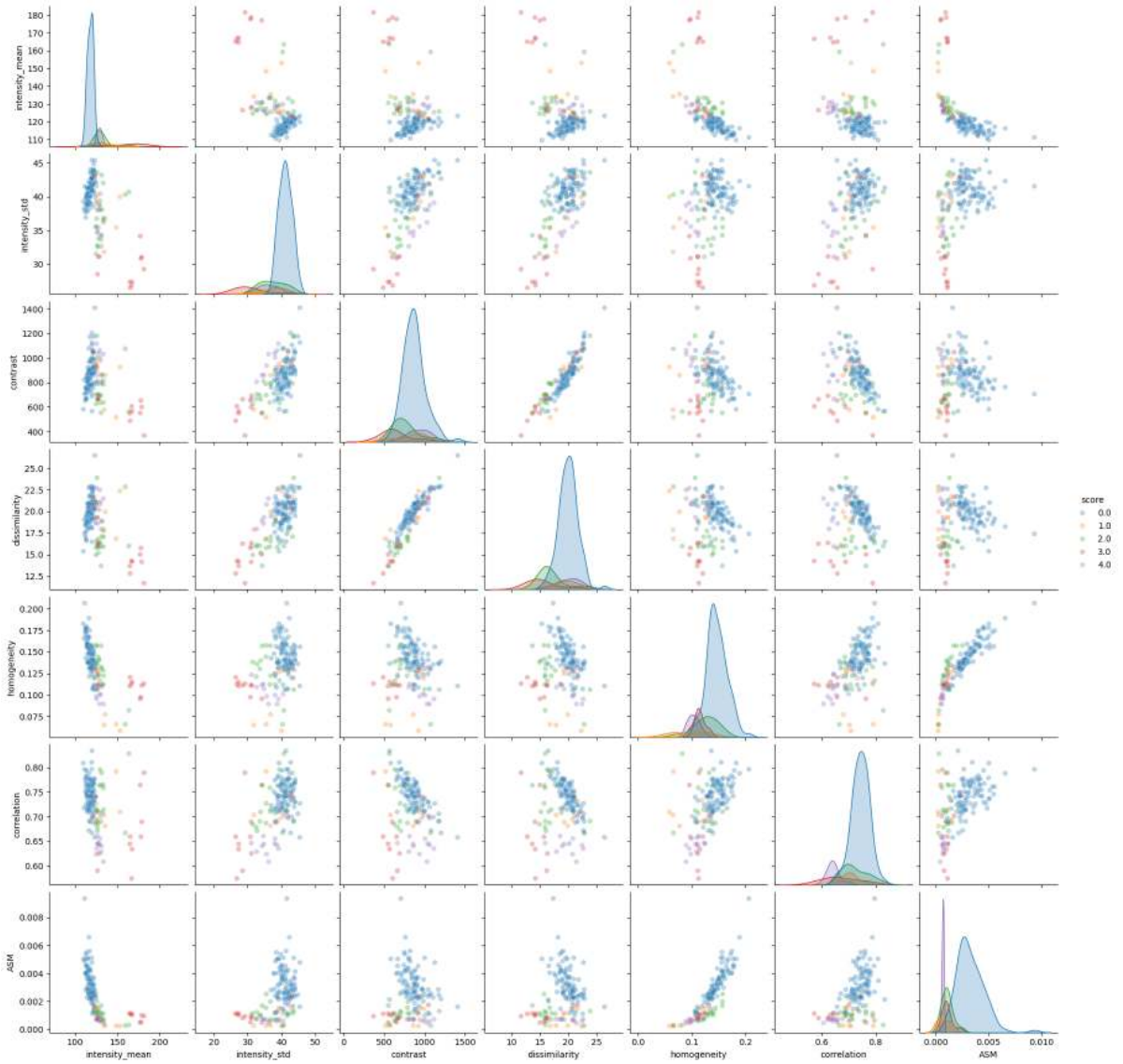




Comparing the features pairwise

The clusters were over fitted let's find relevant features.

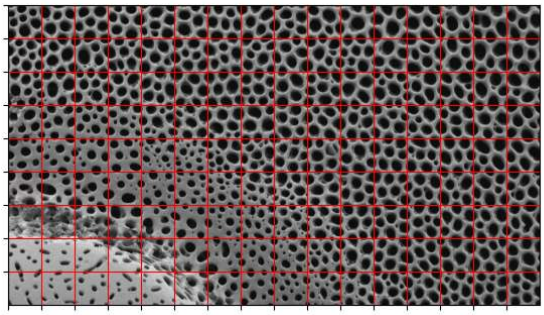
```
sns.pairplot(out_df,hue='score',plot_kws={'alpha': 0.3}, palette=sns.color_palette(
↪"tab10")[0:5]);
```

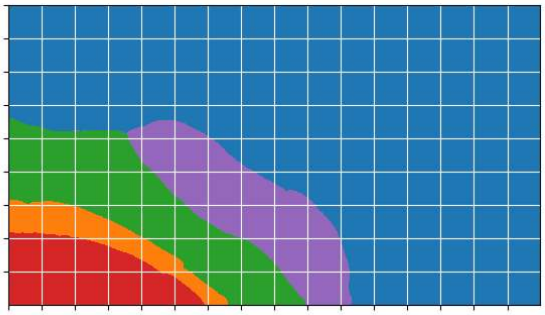


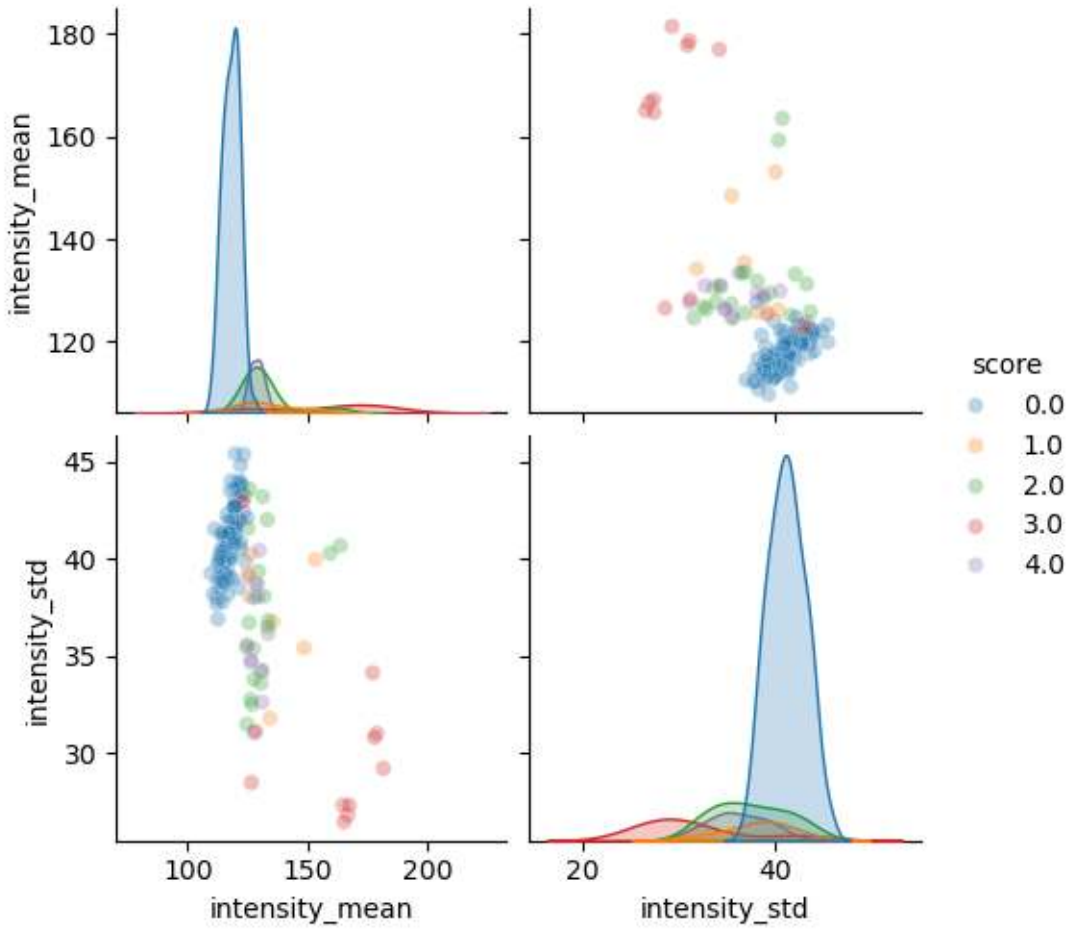
Reduce the number of features

Let's look at mean and std deviation only for each class

```
fig,ax=plt.subplots(1,2,figsize=(15,4))
ax[0].imshow(img, cmap='gray')
ax[0].set_xticks(np.arange(0,img.shape[1],N))
ax[0].set_yticks(np.arange(0,img.shape[0],N))
ax[0].set_xticklabels([])
ax[0].set_yticklabels([])
ax[0].grid(visible=True, which='major', axis='both',color='red')

ax[1].imshow(mask, cmap='tab10',interpolation='none')
ax[1].set_xticks(np.arange(0,img.shape[1],N))
ax[1].set_yticks(np.arange(0,img.shape[0],N))
ax[1].set_xticklabels([])
ax[1].set_yticklabels([])
ax[1].grid(visible=True, which='major', axis='both',color='white')

sns.pairplot(out_df,vars=['intensity_mean','intensity_std'],hue='score',plot_kws={
 ↪'alpha': 0.3}, palette=sns.color_palette("tab10")[0:5]);
```





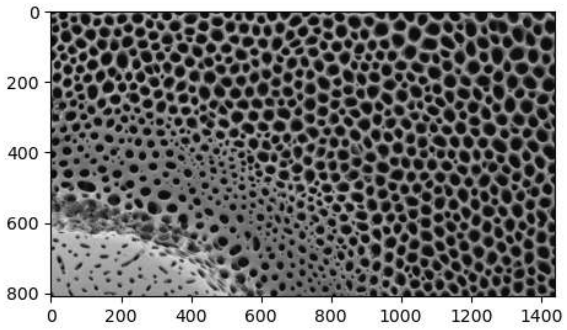


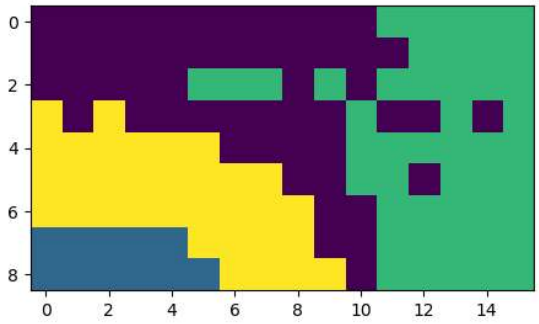
Clustering with less features

```
from sklearn.cluster import KMeans

km = KMeans(4)

km.fit(out_df[['intensity_mean','intensity_std']])
```

```
KMeans(n_clusters=4)
```

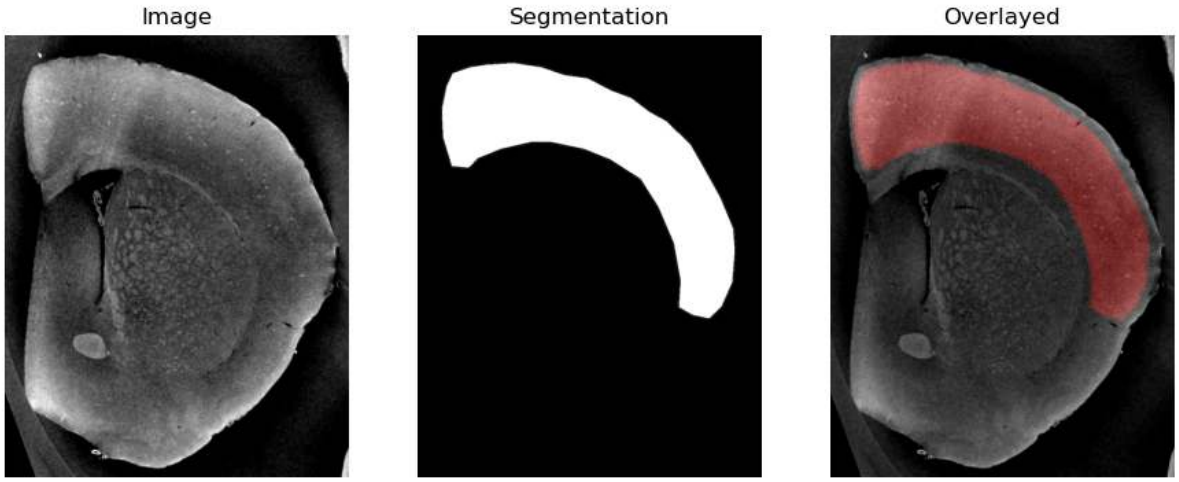
```
fig,ax=plt.subplots(1,2,figsize=(12,3))
ax[0].imshow(img,cmap='gray')
ax[1].imshow(np.reshape(km.labels_,(region_labels.shape[1]//N,region_labels.shape[0]//
↪N)).transpose());
```





0.25.6 Applying Texture analysis to Brain

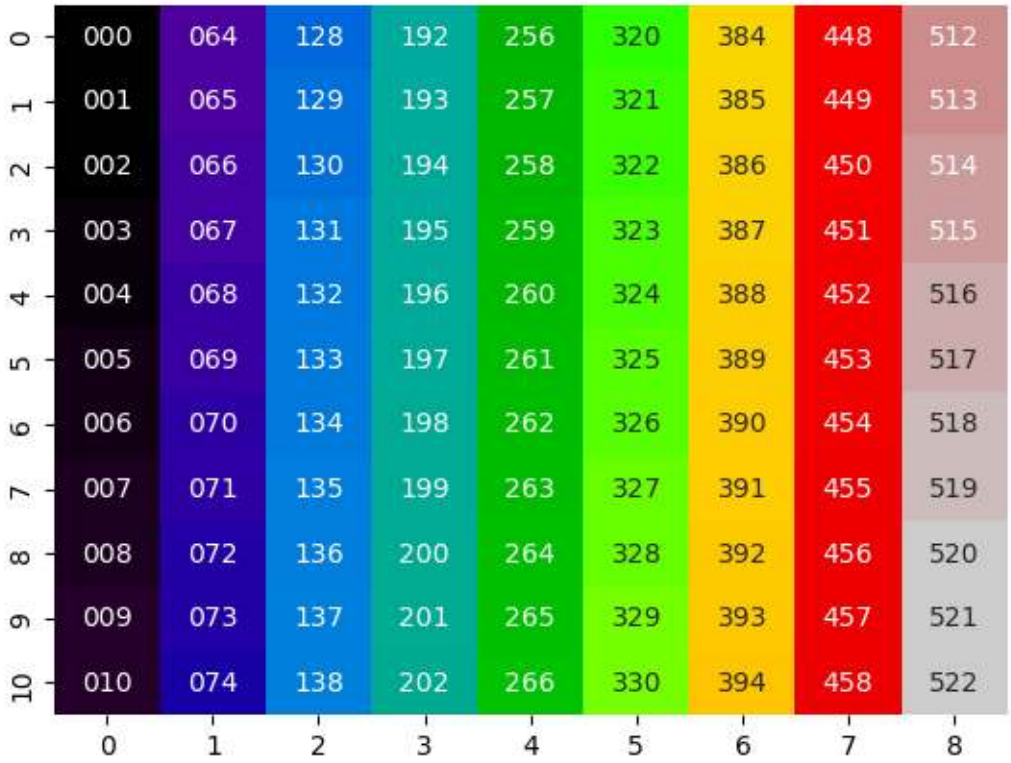
```
cortex_img = np.clip(imread("figures/example_poster.tif")
                     [::2, ::2]/270, 0, 255).astype(np.uint8)

cortex_mask = imread("figures/example_poster_mask.tif")[::1, ::1, 0]/255.0

# show the slice and threshold
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(11, 5),dpi=100)
ax1.imshow(cortex_img, cmap='gray'); ax1.axis('off') ; ax1.set_title('Image')
ax2.imshow(cortex_mask, cmap='gray'); ax2.axis('off'); ax2.set_title('Segmentation')
# here we mark the threshold on the original image

ax3.imshow(label2rgb(cortex_mask > 0, cortex_img, bg_label=0))
ax3.axis('off'); ax3.set_title('Overlayed');
```



0.25.7 Tiling the image

Here we divide the image up into unique tiles for further processing. Each tile is 48×48 pixels.

```
xx, yy = np.meshgrid(
    np.arange(cortex_img.shape[1]),
    np.arange(cortex_img.shape[0]))
region_labels = (xx//48) * 64+yy//48
region_labels = region_labels.astype(int)
sns.heatmap(region_labels[::48, ::48].astype(int),
            annot=True,
            fmt="03d",
            cmap='nipy_spectral',
            cbar=False,
            );
```



0.25.8 Calculating Texture

Here we calculate the texture by using a tool called the gray level co-occurrence matrix which are part of the features library in skimage. We focus on two metrics in this, specifically dissimilarity and correlation which we calculate for each tile. We then want to see which of these parameters correlated best with belonging to a nerve fiber.

```
# compute some gray level co-occurrence matrix (GLCM) properties each patch
from skimage.feature import greycomatrix, greycoprops
from tqdm.notebook import tqdm
grayco_prop_list = ['contrast', 'dissimilarity',
                    'homogeneity', 'energy',
```

(continues on next page)

(continued from previous page)

```
                   'correlation', 'ASM']

prop_imgs = {}
for c_prop in grayco_prop_list:
    prop_imgs[c_prop] = np.zeros_like(cortex_img, dtype=np.float32)
score_img = np.zeros_like(cortex_img, dtype=np.float32)
out_df_list = []
for patch_idx in np.unique(region_labels):
    xx_box, yy_box = np.where(region_labels == patch_idx)

    glcm = greycomatrix(cortex_img[xx_box.min():xx_box.max(),
                                   yy_box.min():yy_box.max()],
                        [5], [0], 256, symmetric=True, normed=True)

    mean_score = np.mean(cortex_mask[region_labels == patch_idx])
    score_img[region_labels == patch_idx] = mean_score

    out_row = dict(
        intensity_mean=np.mean(cortex_img[region_labels == patch_idx]),
        intensity_std=np.std(cortex_img[region_labels == patch_idx]),
        score=mean_score)

    for c_prop in grayco_prop_list:
        out_row[c_prop] = greycoprops(glcm, c_prop)[0, 0]
        prop_imgs[c_prop][region_labels == patch_idx] = out_row[c_prop]

    out_df_list += [out_row];
```

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
Cell In[56], line 2
      1 # compute some gray level co-occurrence matrix (GLCM) properties each patch
----> 2 from skimage.feature import greycomatrix, greycoprops
      3 from tqdm.notebook import tqdm
      4 grayco_prop_list = ['contrast', 'dissimilarity',
      5                     'homogeneity', 'energy',
      6                     'correlation', 'ASM']

ImportError: cannot import name 'greycomatrix' from 'skimage.feature' (/opt/
↪anaconda3/envs/qbi2025/lib/python3.9/site-packages/skimage/feature/__init__.py)
```

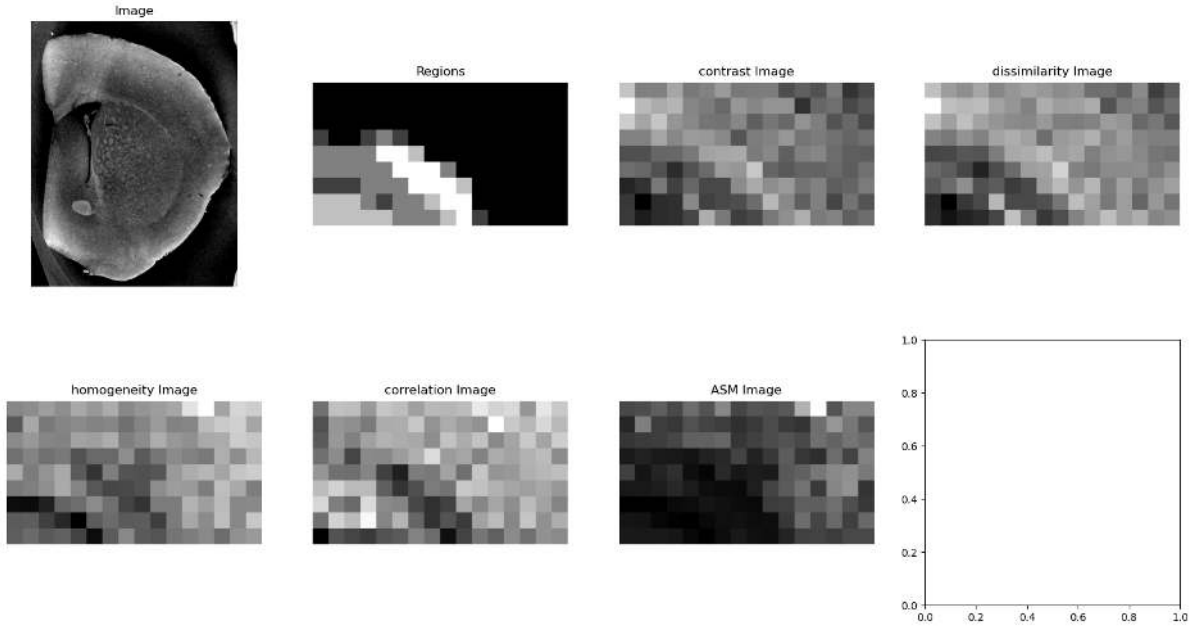
Resulting gray level co-occurrence properties

```
# show the slice and threshold
fig, m_axs = plt.subplots(2, 4, figsize=(20, 10))
n_axs = m_axs.flatten()
ax1 = n_axs[0]
ax2 = n_axs[1]
ax1.imshow(cortex_img, cmap='gray')
ax1.axis('off')
ax1.set_title('Image')
ax2.imshow(score_img, cmap='gray')
ax2.axis('off')
```

(continues on next page)

(continued from previous page)

```
ax2.set_title('Regions')
for c_ax, c_prop in zip(n_axs[2:], grayco_prop_list):
    c_ax.imshow(prop_imgs[c_prop], cmap='gray')
    c_ax.axis('off')
    c_ax.set_title('{} Image'.format(c_prop))
```



Resulting gray level co-occurrence properties

```
import pandas as pd
out_df = pd.DataFrame(out_df_list)
out_df['positive_score'] = out_df['score'].map(
    lambda x: 'FG' if x > 0 else 'BG')
out_df.describe()
```

```
       intensity_mean  intensity_std       score     contrast  dissimilarity  \
count      144.000000     144.000000  144.000000   144.000000     144.000000
mean       124.780106      39.216082    0.888889   837.406576      19.228806
std         14.221951       4.056652    1.343782   163.129731       2.455831
min        109.707654      26.468436    0.000000   368.159022      11.707204
25%        116.853920      38.061432    0.000000   736.890218      17.876735
50%        120.748765      40.187202    0.000000   844.649570      19.582353
75%        126.503488      41.980307    2.000000   924.919002      20.876570
max        181.460617      45.414126    4.000000  1409.422340      26.451553

       homogeneity  correlation         ASM
count   144.000000   144.000000  144.000000
mean      0.134878     0.725910    0.002449
std       0.025904     0.049160    0.001557
min       0.058498     0.574036    0.000219
25%       0.119806     0.698575    0.001119
50%       0.136841     0.731594    0.002368
```

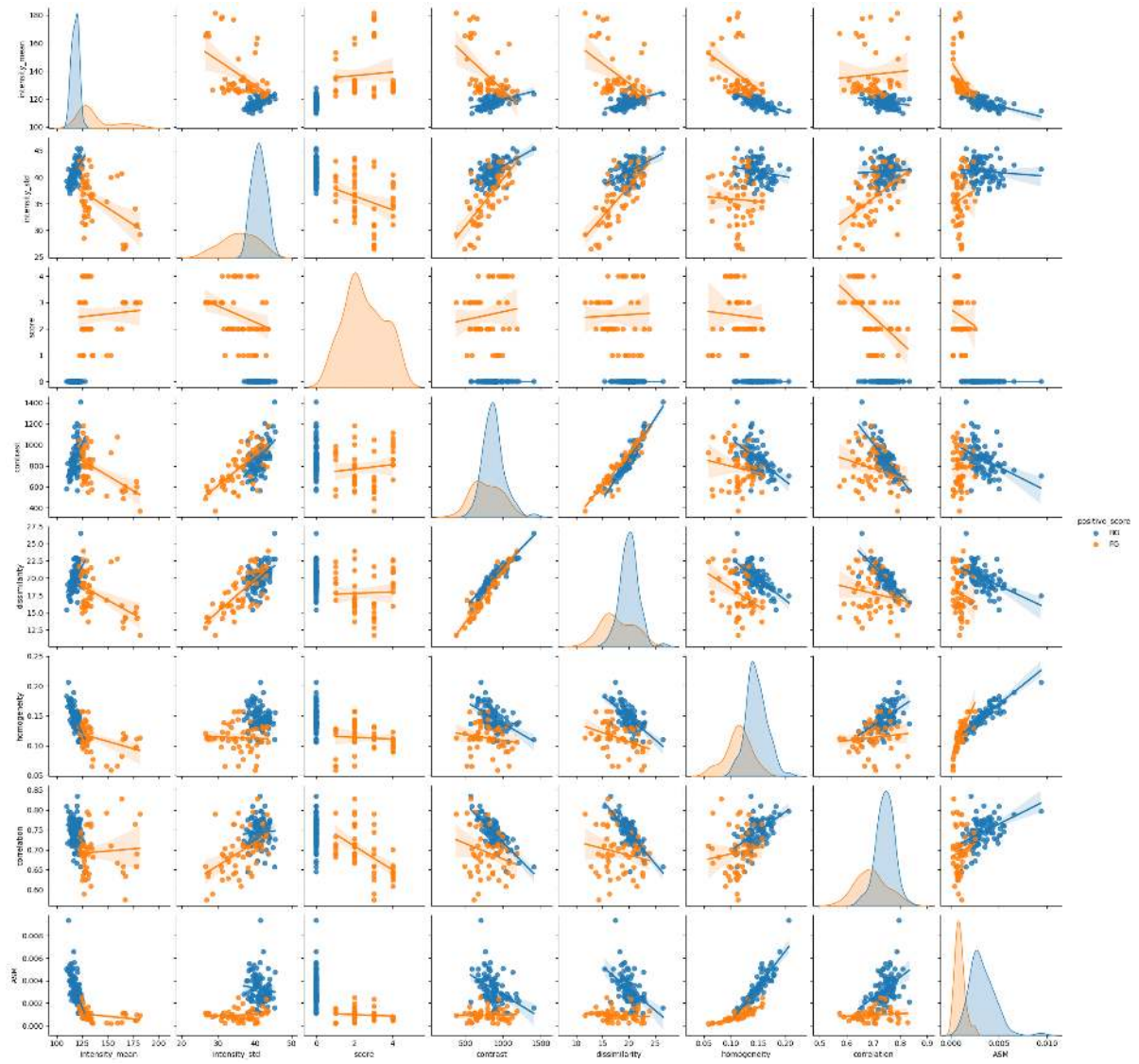
(continues on next page)

(continued from previous page)

```
75%         0.152959      0.762922      0.003473
max         0.206492      0.834216      0.009360
```

A pair plot to show how the properties are related

```
sns.pairplot(out_df, hue='positive_score', kind="reg");
```



0.26 Summary

0.26.1 Distance maps

- Definition
- Importance of neighborhood
- Thickness/Pore radius map

0.26.2 Visualizing 3D data

- Meshes
- Surfaces
- Tools in python

0.26.3 Surface description

- Curvatures

0.26.4 Texture

- Textures are patterns in images
- Measure texture characteristics

0.26.5 Next week on QBI

- Structure description using skeletons
- Advanced item labeling