Quantitative Big Imaging - Shape Analysis

Anders Kaestner

Mar 29, 2021

CONTENTS

This is the lecture notes fot the 6th lecture of the Quantitative big imaging class given during the spring semester 2021 at ETH Zurich, Switzerland.

CHAPTER

ONE

SHAPE ANALYSIS

Quantitative Big Imaging ETHZ: 227-0966-00L

Part 1

1.1 Previously on QBI . . .

- Highlighting the contrast of interest
- Minimizing Noise
- Understanding value histograms
- Dealing with multi-valued data
- Hysteresis Method
- K-Means Analysis
- Contouring

1.2 Learning Objectives

1.2.1 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?

1.3 Outline

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

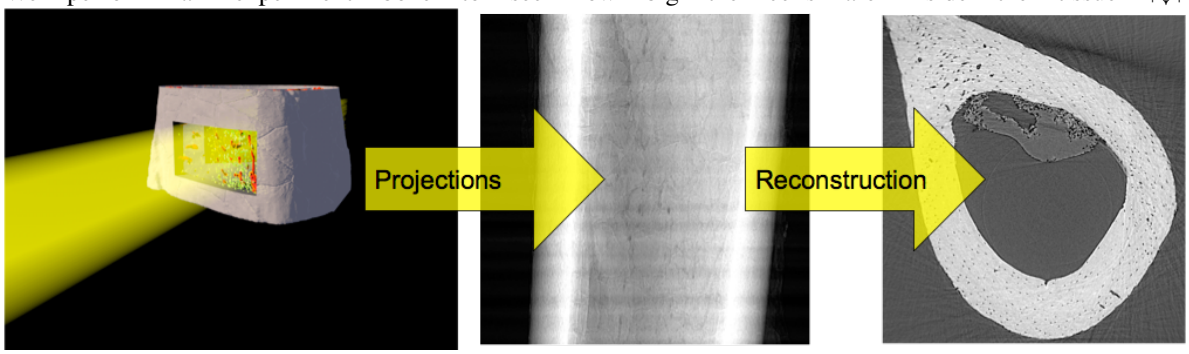
1.4 Metrics

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

1.5 Motivation

We have dramatically simplified our data, but there is still too much.

- We perform an experiment bone to see how big the cells are inside the tissue $↓$



2560 x 2560 x 2160 x 32 bit *56GB / sample*

- Filtering and Enhancement!$↓$
- 56GB of less noisy data

- **Segmentation**

↓

2560 x 2560 x 2160 x 1 bit (1.75GB / sample)

- Still an aweful lot of data

1.5.1 What did we want in the first place?

Single numbers :

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

1.6 Literature / Useful References

- Jean Claude, Morphometry with R
  - Online through ETHZ
- John C. Russ, "The Image Processing Handbook",(Boca Raton, CRC Press)
  - Available online within domain ethz.ch (or proxy.ethz.ch / public VPN)
- Principal Component Analysis
  - Venables, W. N. and B. D. Ripley (2002). Modern Applied Statistics with S, Springer-Verlag
- Shape Tensors
  - http://www.cs.utah.edu/~gk/papers/vissym04/
  - Doube, M.,et al. (2010). BoneJ: Free and extensible bone image analysis in ImageJ. Bone, 47, 1076–9. doi:10.1016/j.bone.2010.08.023
  - Mader, K. , et al. (2013). A quantitative framework for the 3D characterization of the osteocyte lacunar system. Bone, 57(1), 142–154. doi:10.1016/j.bone.2013.06.026
  - Wilhelm Burger, Mark Burge. Principles of Digital Image Processing: Core Algorithms. Springer-Verlag, London, 2009.
  - B. Jähne. Digital Image Processing. Springer-Verlag, Berlin-Heidelberg, 6. edition, 2005.
  - T. H. Reiss. Recognizing Planar Objects Using Invariant Image Features, from Lecture notes in computer science, p. 676. Springer, Berlin, 1993.
  - Image moments

1.7 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 IPython.display        import Markdown, display
from sklearn.neighbors      import KNeighborsClassifier
import webcolors

from collections import defaultdict

%matplotlib inline

plt.rcParams["figure.figsize"] = (8, 8)
plt.rcParams["figure.dpi"] = 150
plt.rcParams["font.size"] = 14
plt.rcParams['font.family'] = ['sans-serif']
plt.rcParams['font.sans-serif'] = ['DejaVu Sans']
plt.style.use('ggplot')
sns.set_style("whitegrid", {'axes.grid': False})
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-1-86d75419166a> in <module>
      1 import numpy as np
----> 2 import pandas as pd
      3 import seaborn as sns
      4 import matplotlib.pyplot as plt
      5 from matplotlib.collections import PatchCollection

ModuleNotFoundError: No module named 'pandas'
```

CHAPTER

TWO

COMPONENT LABELLING

2.1 Segmentation

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

2.2 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

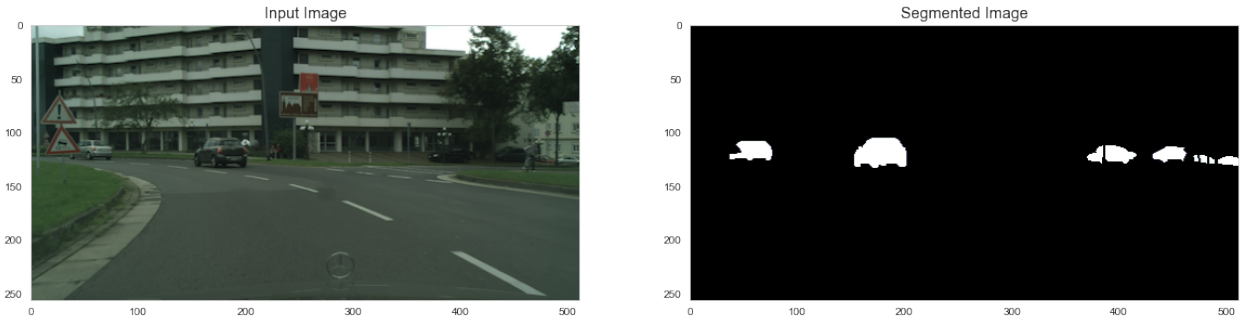
2.3 A cityscape image

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.

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

```
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)
```



2.4 Connected component labeling in python

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

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

label(input, neighbors=None, 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
    ----------
    input : ndarray of dtype int
        Image to label.
    neighbors : {4, 8}, int, optional
        Whether to use 4- or 8-"connectivity".
        In 3D, 4-"connectivity" means connected pixels have to share face,
```

(continues on next page)

(continued from previous page)

```
    whereas with 8-"connectivity", they have to share only edge or vertex.
    **Deprecated, use** ``connectivity`` **instead.**
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.
num : int, optional
    Number of labels, which equals the maximum label index and is only
    returned if return_num is `True`.

See Also
--------
regionprops

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]]
```

(continues on next page)

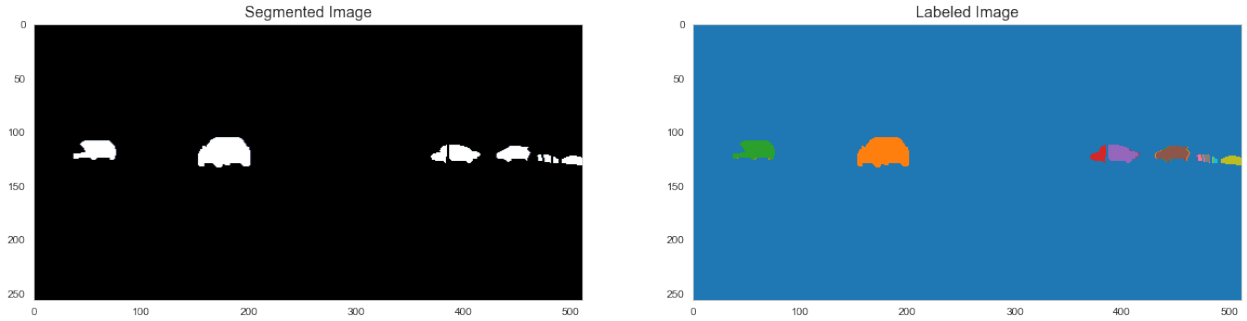
(continued from previous page)

```
    >>> x = np.array([[1, 0, 0],
    ...               [1, 1, 5],
    ...               [0, 0, 0]])
    >>> print(label(x))
    [[1 0 0]
     [1 1 2]
     [0 0 0]]
```

2.5 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.

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 8))
ax1.imshow(seg_img, cmap='bone')
ax1.set_title('Segmented Image')
lab_img = label(seg_img)
ax2.imshow(lab_img, cmap=plt.cm.tab10)
ax2.set_title('Labeled Image');
```



2.6 Area of each segment

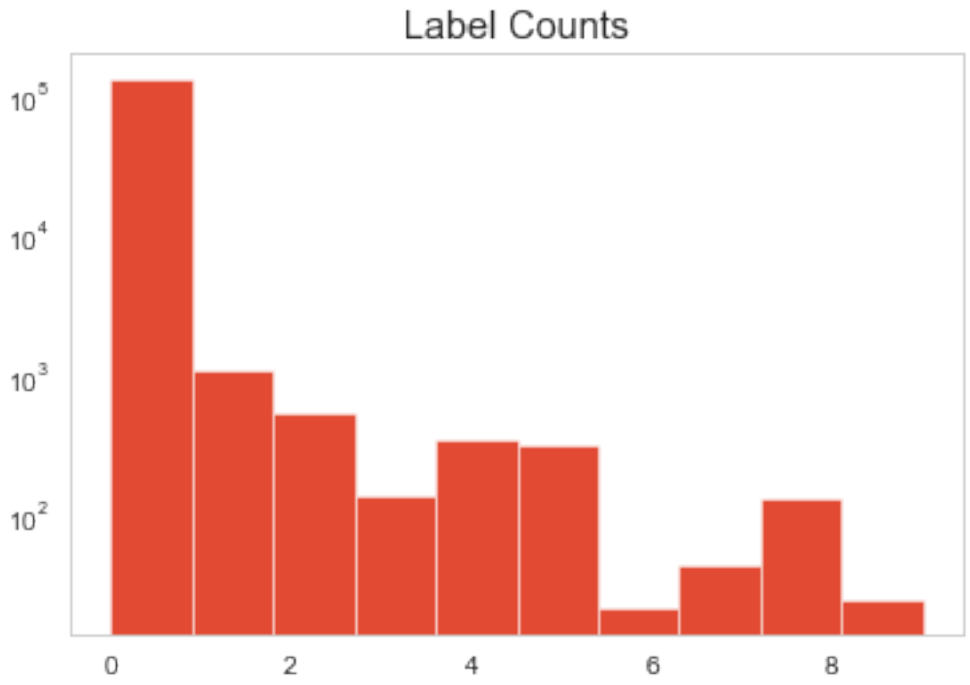
Now, we can start making measurements in the image. A first thing that comes to mind is to compute the area of the objects. This is a very simple operation; we only have to count the number of pixels belonging to each label.

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

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, (ax3) = plt.subplots(1, 1)
ax3.hist(lab_img.ravel())
ax3.set_title('Label Counts')
ax3.set_yscale('log')
```


Label Counts
10^5
10^4
10^3
10^2
0
2
4
6
8

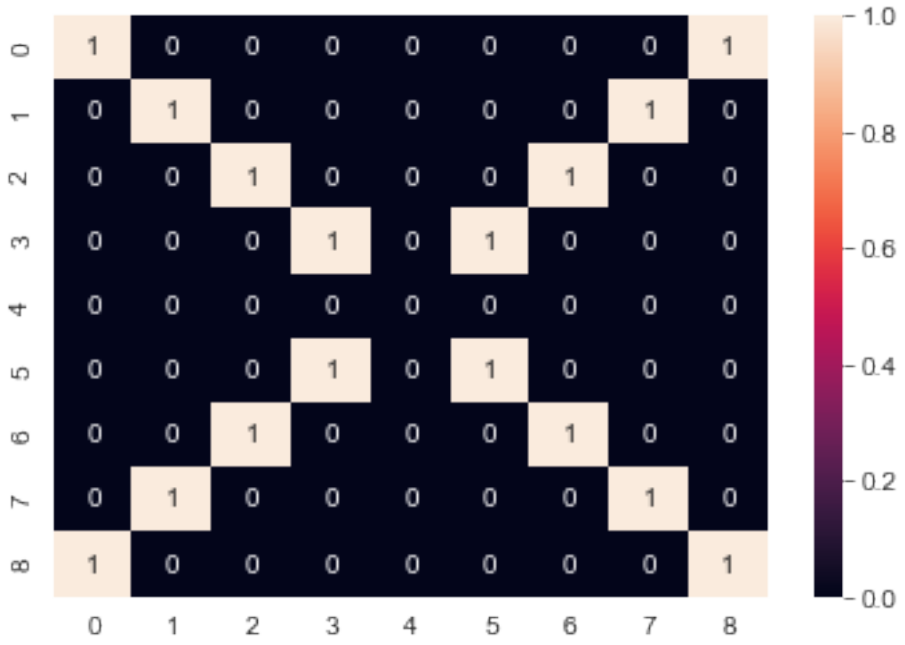
CHAPTER

THREE

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]
sns.heatmap(seg_img, annot=True, fmt="d");
```



3.1 Labeling initialization

Give each point in the image a unique label

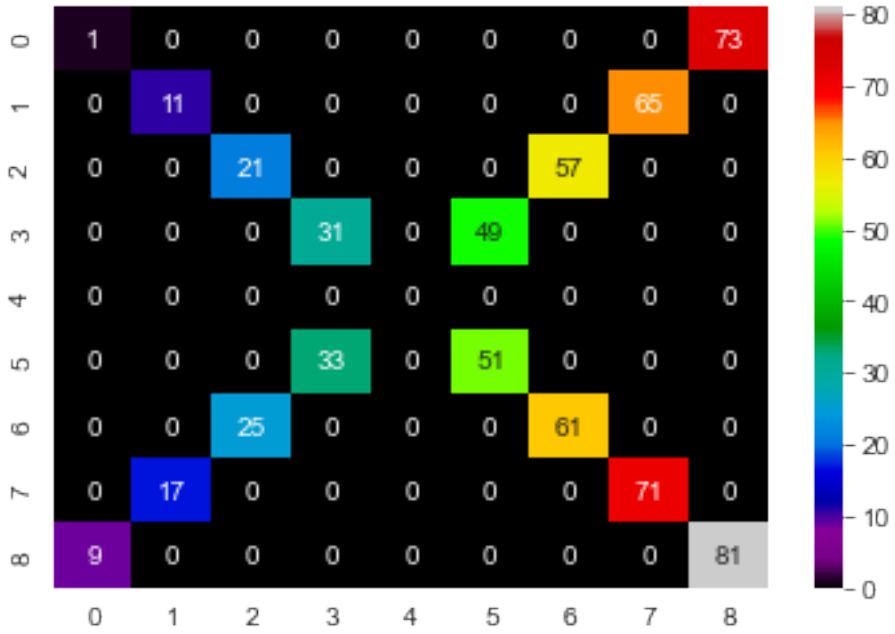
- For each point $(x, y) \in$ Foreground
- Set value to $I_{x,y} = x + y * width + 1$

```
idx_img = np.zeros_like(seg_img)
for x in range(seg_img.shape[0]):
    for y in range(seg_img.shape[1]):
```

(continues on next page)

(continued from previous page)

```
        if seg_img[x, y] > 0:
            idx_img[x, y] = x+y*seg_img.shape[0]+1
sns.heatmap(idx_img, annot=True,
            fmt="d", cmap='nipy_spectral');
```



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 $\min(I_{x,y}, I_{x',y'})$
- Repeat until no more labels have been changed

3.2.1 Implementation of the brush fire algorithm

```
last_img = idx_img.copy()
img_list = [last_img]
for iteration, c_ax in enumerate(m_axs.flatten(), 1):
    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]:
```

(continues on next page)

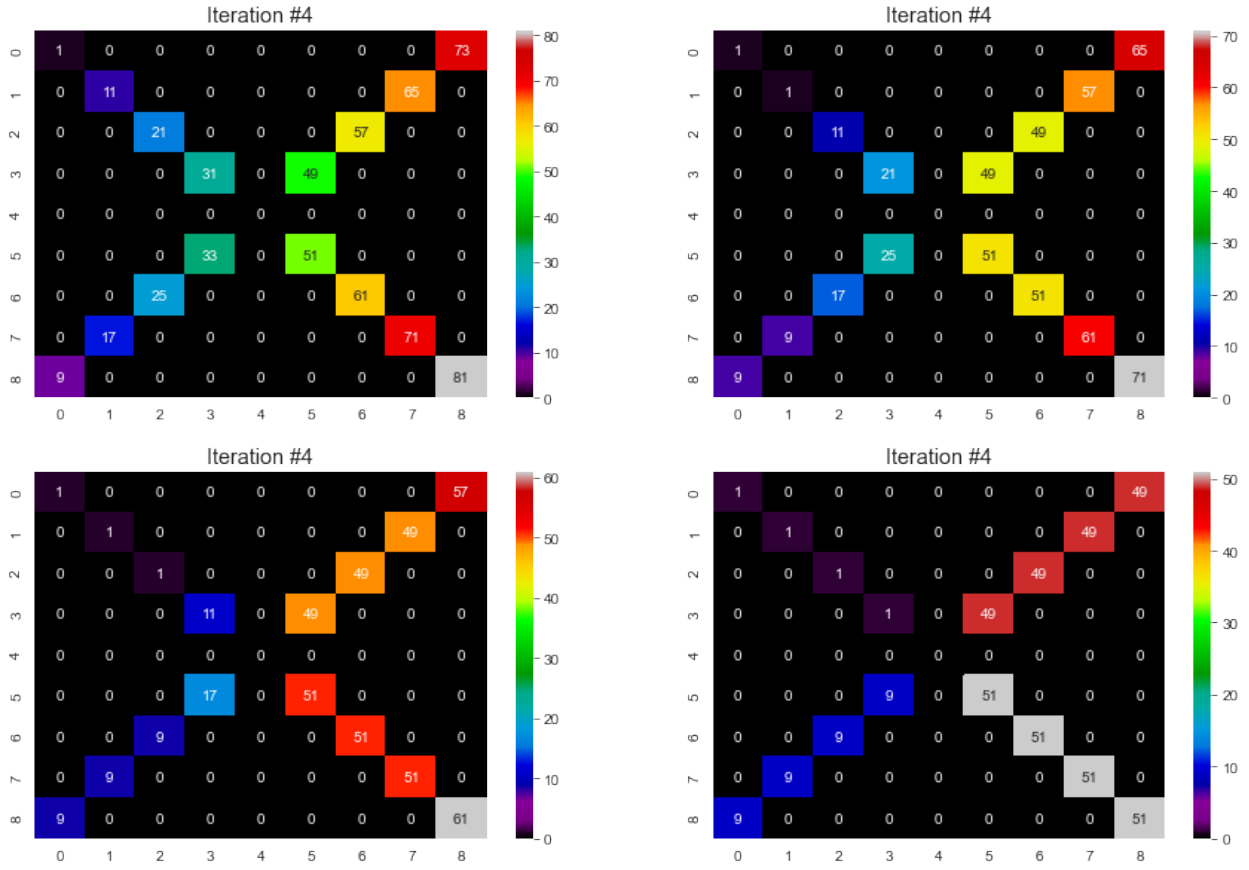
(continued from previous page)

```
                    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):
                                i_xpyp = last_img[x+xp, y+yp]
                                if i_xpyp > 0:

                                    new_val = min(i_xy, i_xpyp, cur_img[x, y])
                                    if cur_img[x, y] != new_val:
                                        print((x, y), i_xy, 'vs', (x+xp,
                                                                   y+yp), i_xpyp, '->
↪', new_val)
                                        cur_img[x, y] = new_val

    img_list += [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))
        last_img = cur_img
```

```
(0, 8) 73 vs (1, 7) 65 -> 65
(1, 1) 11 vs (0, 0) 1 -> 1
(1, 7) 65 vs (2, 6) 57 -> 57
(2, 2) 21 vs (1, 1) 11 -> 11
(2, 6) 57 vs (3, 5) 49 -> 49
(3, 3) 31 vs (2, 2) 21 -> 21
(5, 3) 33 vs (6, 2) 25 -> 25
(6, 2) 25 vs (7, 1) 17 -> 17
(6, 6) 61 vs (5, 5) 51 -> 51
(7, 1) 17 vs (8, 0) 9 -> 9
(7, 7) 71 vs (6, 6) 61 -> 61
(8, 8) 81 vs (7, 7) 71 -> 71
Iteration 1 Groups 12 Changes 12
(0, 8) 65 vs (1, 7) 57 -> 57
(1, 7) 57 vs (2, 6) 49 -> 49
(2, 2) 11 vs (1, 1) 1 -> 1
(3, 3) 21 vs (2, 2) 11 -> 11
(5, 3) 25 vs (6, 2) 17 -> 17
(6, 2) 17 vs (7, 1) 9 -> 9
(7, 7) 61 vs (6, 6) 51 -> 51
(8, 8) 71 vs (7, 7) 61 -> 61
Iteration 2 Groups 8 Changes 8
(0, 8) 57 vs (1, 7) 49 -> 49
(3, 3) 11 vs (2, 2) 1 -> 1
(5, 3) 17 vs (6, 2) 9 -> 9
(8, 8) 61 vs (7, 7) 51 -> 51
Iteration 3 Groups 4 Changes 4
Done
```

3.2.2 Looking at the iterations

```
fig, m_axs = plt.subplots(2, 2, figsize=(15, 10)); m_axs=m_axs.ravel()
for c_ax,cur_img in zip(m_axs,img_list):
    sns.heatmap(cur_img,
                annot=True,
                fmt="d",
                cmap='nipy_spectral',
                ax=c_ax)
    c_ax.set_title('Iteration #{}'.format(iteration))
```



3.2.3 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

3.2.4 Let's animate the iterations

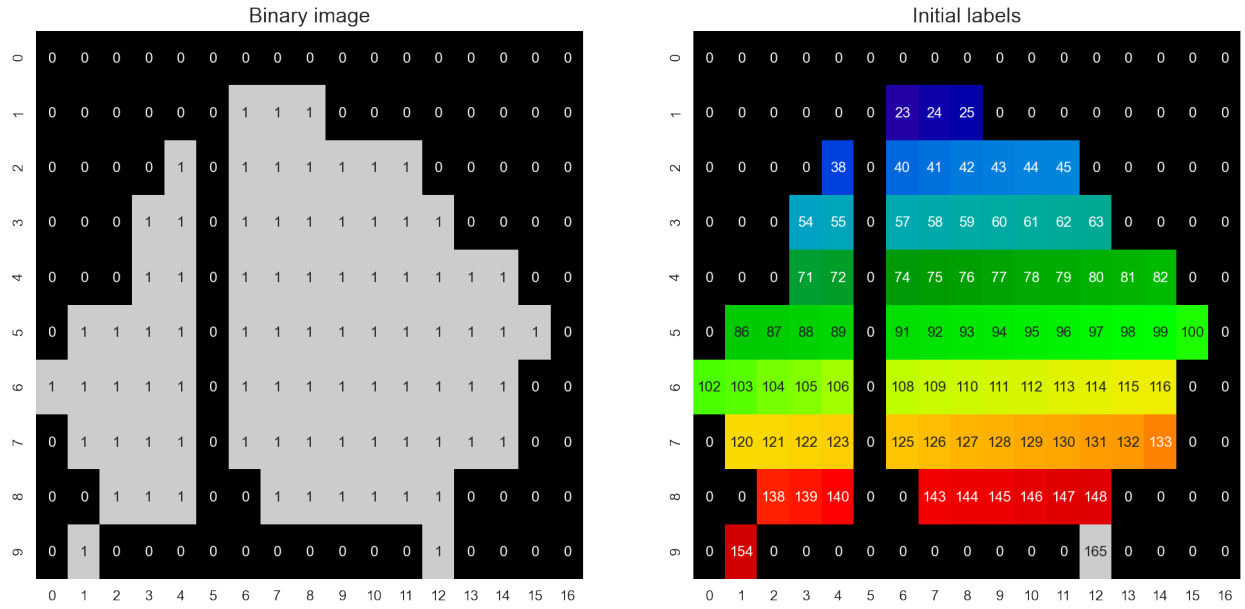
```
fig, c_ax = plt.subplots(1, 1, figsize=(5, 5), dpi=100)

def update_frame(i):
    plt.cla()
    sns.heatmap(img_list[i],
                annot=True,
                fmt="d",
                cmap='nipy_spectral',
                ax=c_ax,
                cbar=False,
                vmin=img_list[0].min(),
                vmax=img_list[0].max())
    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>
```

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)
sns.heatmap(seg_img, annot=True, fmt="d", ax=ax1,
            cmap='nipy_spectral', cbar=False)
ax1.set_title('Binary image')

idx_img = seg_img * np.arange(len(seg_img.ravel())).reshape(seg_img.shape)
sns.heatmap(idx_img, annot=True, fmt="d", ax=ax2,
            cmap='nipy_spectral', cbar=False)
ax2.set_title('Initial labels');
```



3.3.1 Run the labelling on the car image

```
last_img = idx_img.copy()
img_list = [last_img]
for iteration in range(99):
    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]:
                    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):
                                i_xpyp = last_img[x+xp, y+yp]
                                if i_xpyp > 0:
                                    new_val = min(i_xy, i_xpyp, cur_img[x, y])
                                    if cur_img[x, y] != new_val:
                                        cur_img[x, y] = new_val

    img_list += [cur_img] # stores the current image in the iteration list
    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))
        last_img = cur_img
```

```
Iteration 0 Groups 62 Changes 79
Iteration 1 Groups 46 Changes 74
Iteration 2 Groups 32 Changes 68
Iteration 3 Groups 22 Changes 59
```

(continues on next page)

(continued from previous page)

```
Iteration 4 Groups 14 Changes 46
Iteration 5 Groups 8 Changes 31
Iteration 6 Groups 5 Changes 17
Iteration 7 Groups 3 Changes 6
Iteration 8 Groups 2 Changes 1
Done
```

3.3.2 Let's animate the iterations

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

def update_frame(i):
    plt.cla()
    sns.heatmap(img_list[i],
                annot=True,
                fmt="d",
                cmap='nipy_spectral',
                ax=c_ax,
                cbar=False,
                vmin=img_list[0].min(),
                vmax=img_list[0].max())
    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=500,
                          repeat_delay=1000).to_html5_video()
plt.close('all')
HTML(anim_code)
```

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

3.4 Different Neighborhoods

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

```
last_img = idx_img.copy()
img_list = [last_img]
for iteration in range(99):
    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 [-2, -1, 0, 1, 2]:
                    if (x+xp < last_img.shape[0]) and (x+xp >= 0):
                        for yp in [-2, -1, 0, 1, 2]:
```

(continues on next page)

(continued from previous page)

```
                            if (y+yp < last_img.shape[1]) and (y+yp >= 0):
                                i_xpyp = last_img[x+xp, y+yp]
                                if i_xpyp > 0:
                                    new_val = min(i_xy, i_xpyp, cur_img[x, y])
                                    if cur_img[x, y] != new_val:
                                        cur_img[x, y] = new_val

    img_list += [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))
        last_img = cur_img
```

```
Iteration 0 Groups 49 Changes 81
Iteration 1 Groups 24 Changes 71
Iteration 2 Groups 8 Changes 51
Iteration 3 Groups 2 Changes 20
Iteration 4 Groups 1 Changes 1
Done
```

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

def update_frame(i):
    plt.cla()
    sns.heatmap(img_list[i],
                annot=True,
                fmt="d",
                cmap='nipy_spectral',
                ax=c_ax,
                cbar=False,
                vmin=img_list[0].min(),
                vmax=img_list[0].max())
    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=500,
                          repeat_delay=1000).to_html5_video()
plt.close('all')
HTML(anim_code)
```

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

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

```
last_img = idx_img.copy()
img_list = [last_img]
for iteration in range(99):
    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]:
                    if (x+xp < last_img.shape[0]) and (x+xp >= 0):
                        for yp in [-1, 0, 1]:
                            if np.abs(xp)+np.abs(yp) <= 1:
                                if (y+yp < last_img.shape[1]) and (y+yp >= 0):
                                    i_xpyp = last_img[x+xp, y+yp]
                                    if i_xpyp > 0:
                                        new_val = min(
                                            i_xy, i_xpyp, cur_img[x, y])
                                        if cur_img[x, y] != new_val:
                                            cur_img[x, y] = new_val

    img_list += [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))
        last_img = cur_img
```

```
Iteration 0 Groups 68 Changes 76
Iteration 1 Groups 54 Changes 73
Iteration 2 Groups 42 Changes 67
Iteration 3 Groups 31 Changes 62
Iteration 4 Groups 21 Changes 57
Iteration 5 Groups 13 Changes 49
Iteration 6 Groups 11 Changes 39
Iteration 7 Groups 9 Changes 34
Iteration 8 Groups 8 Changes 25
Iteration 9 Groups 7 Changes 20
Iteration 10 Groups 6 Changes 15
Iteration 11 Groups 5 Changes 10
Iteration 12 Groups 4 Changes 6
Iteration 13 Groups 3 Changes 2
```

(continues on next page)

(continued from previous page)

```
Done
```

```
fig, c_ax = plt.subplots(1, 1, figsize=(6, 6), dpi=100)


def update_frame(i):
    plt.cla()
    sns.heatmap(img_list[i],
                annot=True,
                fmt="d",
                cmap='nipy_spectral',
                ax=c_ax,
                cbar=False,
                vmin=img_list[0].min(),
                vmax=img_list[0].max())
    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=500,
                          repeat_delay=1000).to_html5_video()
plt.close('all')
HTML(anim_code)
```

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

3.6 Comparing different neighborhoods

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

CHAPTER

FOUR

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

4.1 What we would like to to do

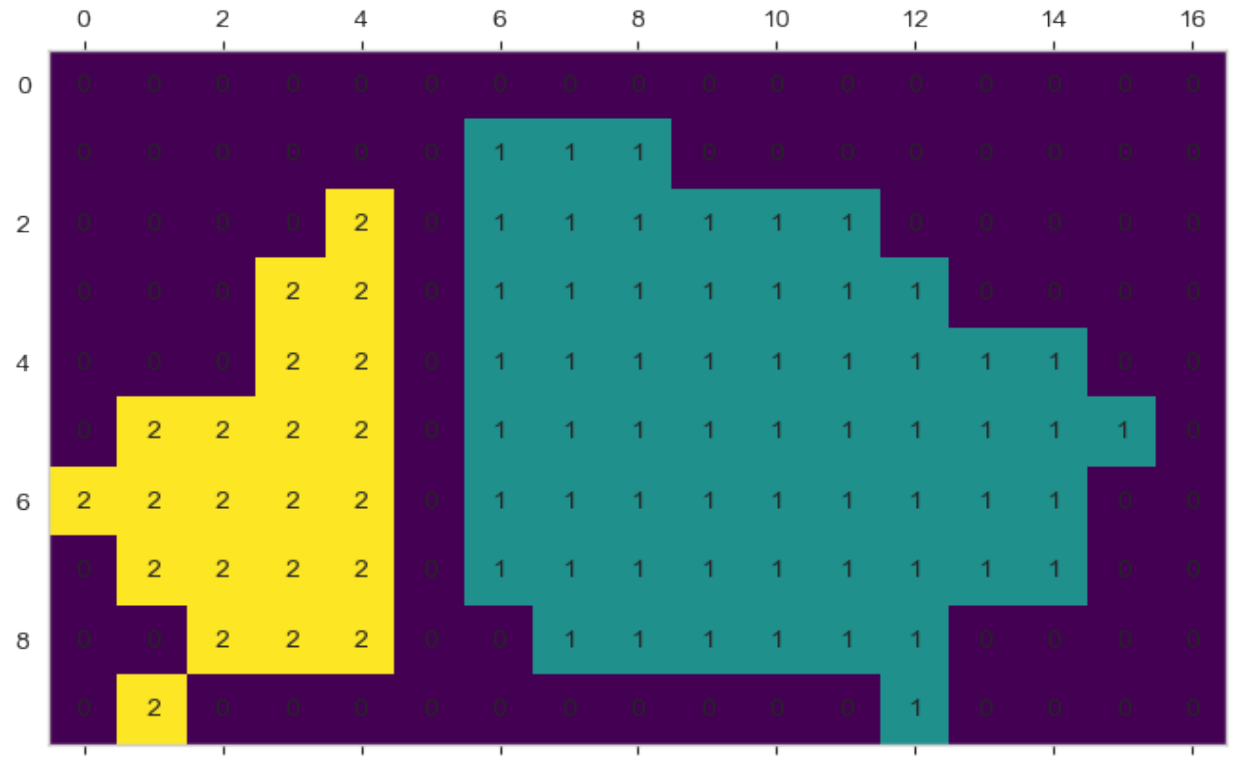
- Count the cells
- Say something about the cells
- Compare the cells in this image to another image

. . . But where do we start?

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

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

```
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)
fig, ax = plt.subplots(figsize=[8,6],dpi=100)
# Using matshow here just because it sets the ticks up nicely. imshow is faster.
ax.matshow(lab_img,cmap='viridis')

for (i, j), z in np.ndenumerate(lab_img):
    ax.text(j, i, '{}'.format(z), ha='center', va='center')
```



4.2.1 Define a center

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

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

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

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

4.2.2 Center of a labeled item

```
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]
items = pd.DataFrame.from_dict({'x': x_coord, 'y': y_coord})

fig, ax = plt.subplots(1,2,figsize=[12,6],dpi=100)
# Using matshow here just because it sets the ticks up nicely. imshow is faster.
ax[1].matshow(lab_img,cmap='viridis')
ax[1].plot(np.mean(y_coord),np.mean(x_coord),'rX',
```

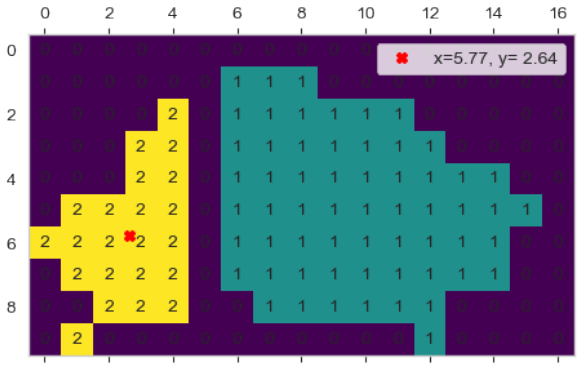
(continues on next page)

(continued from previous page)

```
        label="x={0:0.2f}, y= {1:0.2f}".format(np.mean(x_coord), np.mean(y_coord)))
ax[1].legend()
for (i, j), z in np.ndenumerate(lab_img):
    ax[1].text(j, i, '{}'.format(z), ha='center', va='center')
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



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)

4.3.1 Define a center

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

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

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

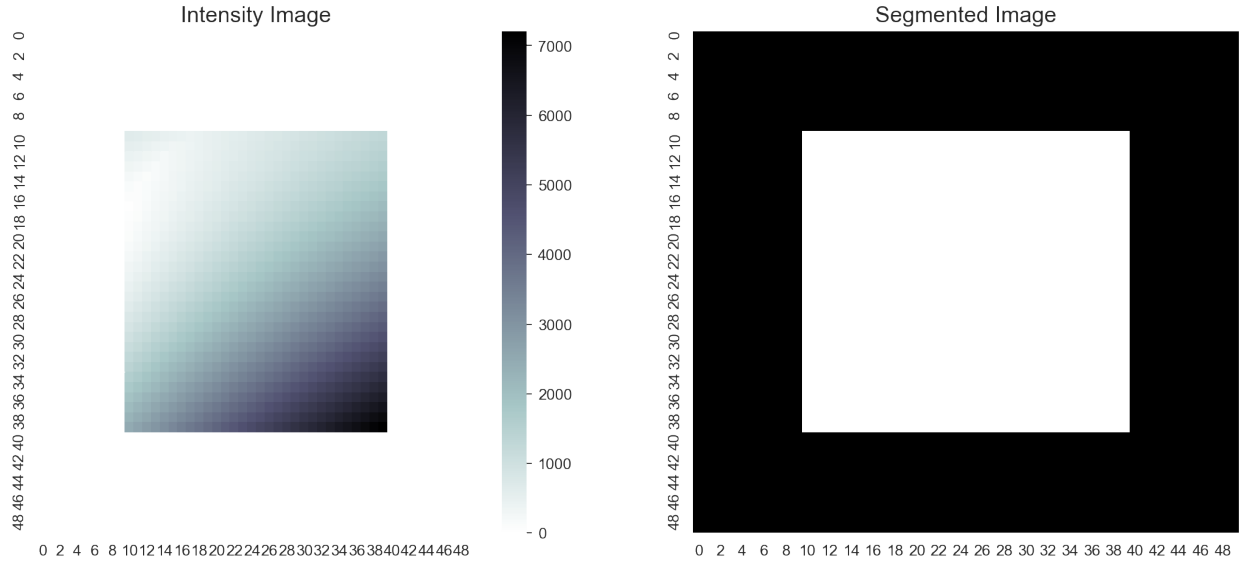
$$\bar{z} = \frac{1}{\Sigma I_{gy}} \sum_{\vec{v} \in I_{id}} (\vec{v} \cdot \vec{k}) I_{gy}(\vec{v})$$

```
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
```

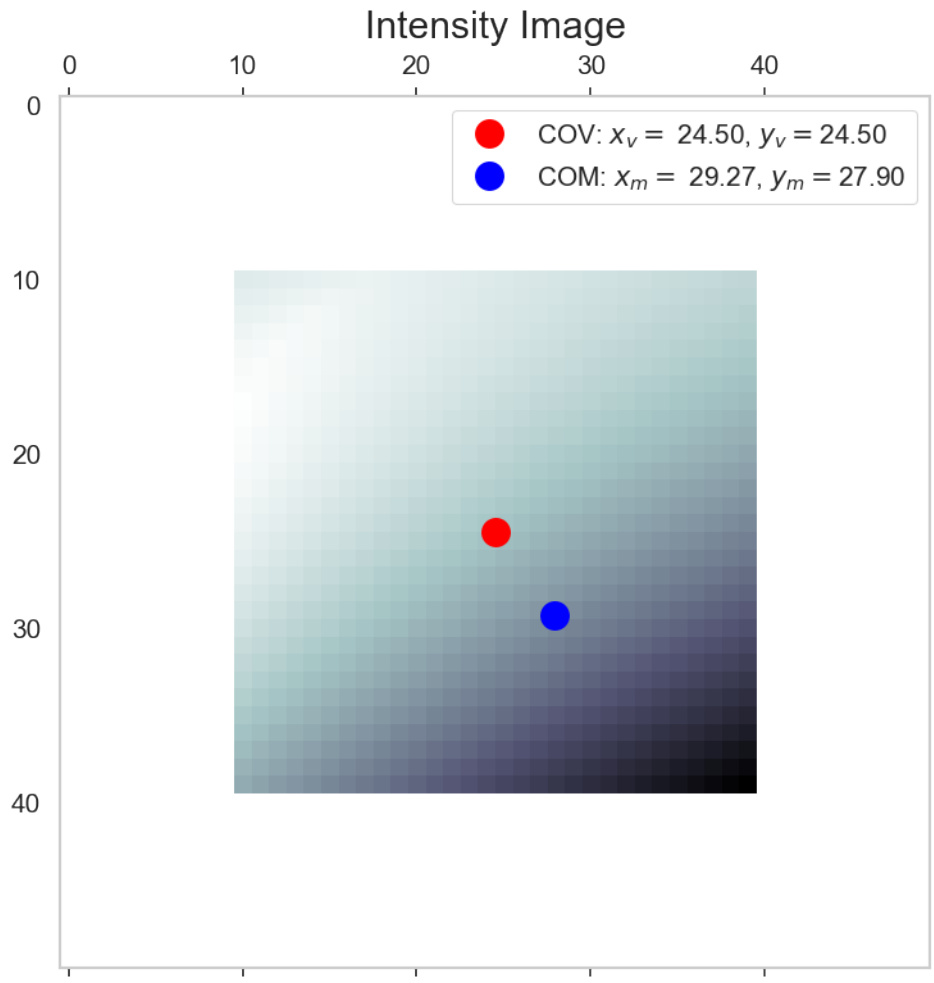
(continues on next page)

(continued from previous page)

```
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), dpi=150)

sns.heatmap(gray_img,ax=ax1, cmap='bone_r', cbar=True)
ax1.set_title('Intensity Image')

sns.heatmap(seg_img, ax=ax2, cmap='bone', cbar=False)
ax2.set_title('Segmented Image');
```




```
# 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)

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

ax1.matshow(gray_img,cmap='bone_r')
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();
```



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!

4.4.1 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, coordinates=None)
    Measure properties of labeled image regions.

    Parameters
    ----------
    label_image : (N, M) 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 : (N, M) ndarray, optional
        Intensity (i.e., input) image with same size as labeled image.
        Default is None.
    cache : bool, optional
        Determine whether to cache calculated properties. The computation is
        much faster for cached properties, whereas the memory consumption
        increases.
    coordinates : DEPRECATED
        This argument is deprecated and will be removed in a future version
        of scikit-image.

        See :ref:`Coordinate conventions <numpy-images-coordinate-conventions>`
        for more details.

        .. deprecated:: 0.16.0
            Use "rc" coordinates everywhere. It may be sufficient to call
            ``numpy.transpose`` on your label image to get the same values as
            0.15 and earlier. However, for some properties, the transformation
            will be less trivial. For example, the new orientation is
            :math:`\frac{\pi}{2}` plus the old orientation.


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

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

    **area** : int
        Number of pixels of the region.
    **bbox** : tuple
        Bounding box ``(min_row, min_col, max_row, max_col)``.
```

(continues on next page)

(continued from previous page)

```
        Pixels belonging to the bounding box are in the half-open interval
        ``[min_row; max_row)`` and ``[min_col; max_col)``.
    **bbox_area** : int
        Number of pixels of bounding box.
    **centroid** : array
        Centroid coordinate tuple ``(row, col)``.
    **convex_area** : int
        Number of pixels of convex hull image, which is the smallest convex
        polygon that encloses the region.
    **convex_image** : (H, J) ndarray
        Binary convex hull image which has the same size as bounding box.
    **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** : float
        The diameter of a circle with the same area as the region.
    **euler_number** : int
        Euler characteristic of region. Computed as number of objects (= 1)
        subtracted by number of holes (8-connectivity).
    **extent** : float
        Ratio of pixels in the region to pixels in the total bounding box.
        Computed as ``area / (rows * cols)``
    **filled_area** : int
        Number of pixels of the region will all the holes filled in. Describes
        the area of the filled_image.
    **filled_image** : (H, J) ndarray
        Binary region image with filled holes which has the same size as
        bounding box.
    **image** : (H, J) ndarray
        Sliced binary region image which has the same size as 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_image** : ndarray
        Image inside region bounding box.
    **label** : int
        The label in the labeled input image.
    **local_centroid** : array
        Centroid coordinate tuple ``(row, col)``, relative to region bounding
        box.
    **major_axis_length** : float
        The length of the major axis of the ellipse that has the same
        normalized second central moments as the region.
    **max_intensity** : float
        Value with the greatest intensity in the region.
    **mean_intensity** : float
        Value with the mean intensity in the region.
    **min_intensity** : float
        Value with the least intensity in the region.
    **minor_axis_length** : float
        The length of the minor axis of the ellipse that has the same
```

(continues on next page)

(continued from previous page)

```
    normalized second central moments as the region.
**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.
**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.
**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.
**weighted_centroid** : array
    Centroid coordinate tuple ``(row, col)`` weighted with intensity
    image.
**weighted_local_centroid** : array
    Centroid coordinate tuple ``(row, col)``, relative to region bounding
    box, weighted with intensity image.
**weighted_moments** : (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.
**weighted_moments_central** : (3, 3) ndarray
    Central moments (translation invariant) of intensity image up to
    3rd order::

        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.
**weighted_moments_hu** : tuple
    Hu moments (translation, scale and rotation invariant) of intensity
    image.
```

(continues on next page)

(continued from previous page)

```
    **weighted_moments_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).

    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

    Examples
    --------
    >>> from skimage import data, util
    >>> from skimage.measure import label
    >>> 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)
```

4.4.2 Let's try regionprops on our image

```
from skimage.measure import regionprops
all_regs = regionprops(seg_img, intensity_image=gray_img)
attr={}
for c_reg in all_regs:
    for k in dir(c_reg):
        if not k.startswith('_') and ('image' not in k):
            attr[k]=getattr(c_reg, k)
attr_df=pd.DataFrame.from_dict(attr, orient="index")
attr_df
```

```
                                                                      0
area                                                                900
bbox                                                 (10, 10, 40, 40)
bbox_area                                                           900
centroid                                                 (24.5, 24.5)
convex_area                                                         900
coords                  [[10, 10], [10, 11], [10, 12], [10, 13], [10, ...
eccentricity                                                          0
equivalent_diameter                                             33.8514
euler_number                                                          1
extent                                                                1
filled_area                                                         900
inertia_tensor          [[74.91666666666667, -0.0], [-0.0, 74.91666666...
inertia_tensor_eigvals              [74.91666666666667, 74.91666666666667]
label                                                                 1
local_centroid                                           (14.5, 14.5)
major_axis_length                                                34.6218
max_intensity                                                    7207.62
mean_intensity                                                   2223.91
min_intensity                                                    41.4433
minor_axis_length                                                34.6218
moments                 [[900.0, 13050.0, 256650.0, 5676750.0], [13050...
moments_central         [[900.0, 0.0, 67425.0, 0.0], [0.0, 0.0, 0.0, 0...
moments_hu              [0.16648148148148148, 0.0, 0.0, 0.0, 0.0, 0.0,...
moments_normalized      [[nan, nan, 0.08324074074074074, 0.0], [nan, 0...
orientation                                                     0.785398
perimeter                                                          116
slice                          (slice(10, 40, None), slice(10, 40, None))
solidity                                                              1
weighted_centroid                 (29.27320794892042, 27.898230520694007)
weighted_local_centroid           [19.27320794892042, 17.898230520694007]
weighted_moments        [[2001517.0866305707, 35823614.20762183, 76860...
weighted_moments_central  [[2001517.0866305707, -4.3655745685100555e-09,...
weighted_moments_hu       [6.219233313138948e-05, 2.8996448632098516e-11...
weighted_moments_normalized  [[nan, nan, 3.1809307780004445e-05, -8.2096428...
```

4.4.3 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?"

4.5 Extents: With a single object

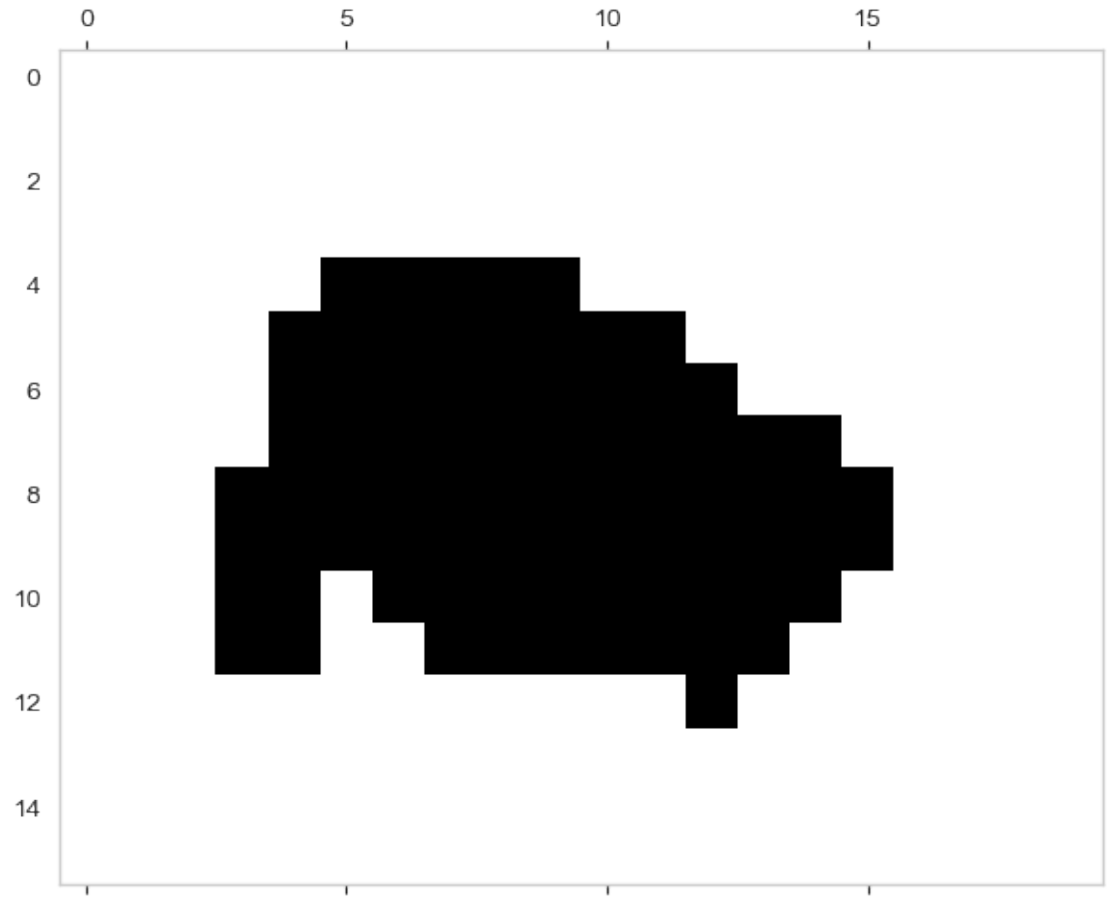
Exents or caliper lenghts are the size of the object in a given direction. Since the coordinates of our image our x and y the extents are calculated in these directions

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

4.5.1 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');
```

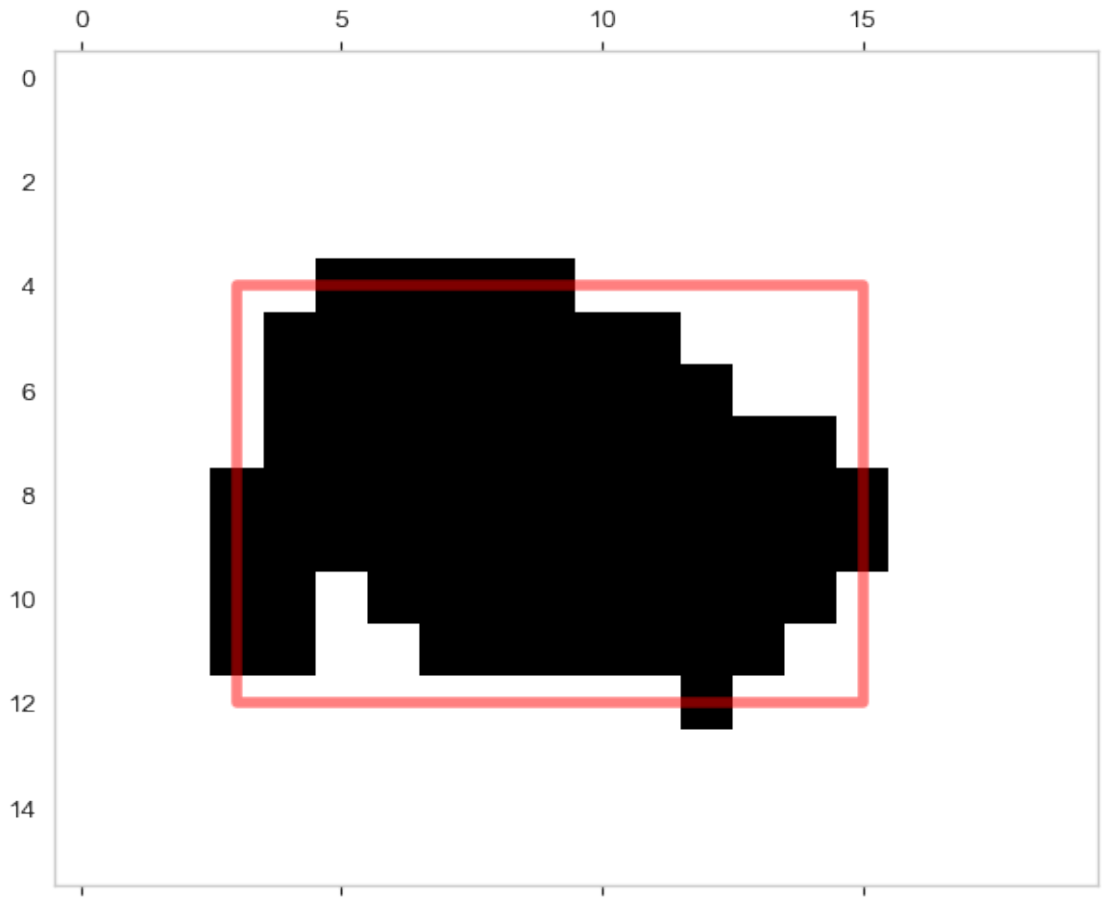


4.5.2 Finding a bounding box

```
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
```

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

ax1.matshow(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);
```


0
5
10
15
0
2
4
6
8
10
12
14

CHAPTER

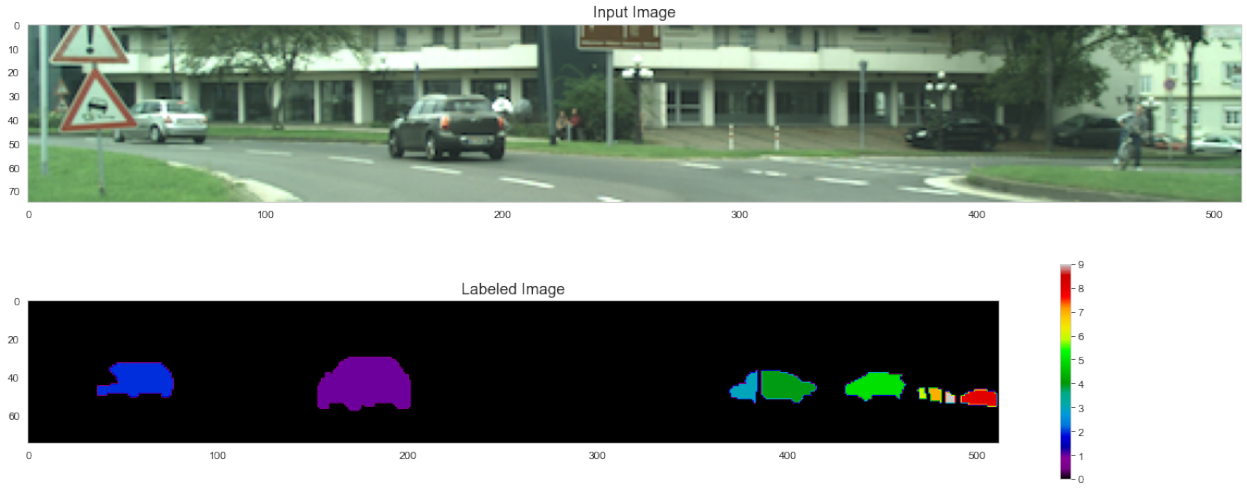
FIVE

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'))
ax2.set_title('Labeled Image');
```



5.1 Shape Analysis

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

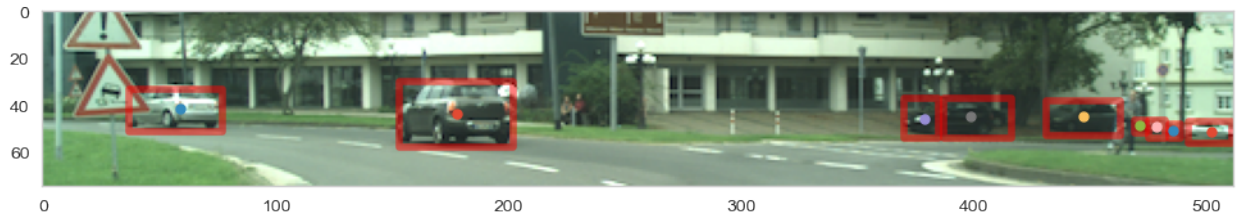
```
all_regions = regionprops(lab_img)
fig, ax1 = plt.subplots(1, 1, figsize=(12, 6), dpi=100)
ax1.imshow(car_img)
print('Found ', len(all_regions), 'regions')
bbox_list = []
for c_reg in all_regions:
```

(continues on next page)

(continued from previous page)

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

```
Found  9 regions
```



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


# guess color name based on rgb value
color_name_class = KNeighborsClassifier(1)
c_names = sorted(webcolors.CSS3_NAMES_TO_HEX.keys())
color_name_class.fit([tuple(webcolors.name_to_rgb(k)) for k in c_names],c_names)


reg_df = pd.DataFrame([dict(label    = c_reg.label,
                            bbox     = c_reg.bbox,
                            area     = c_reg.area,
                            centroid = c_reg.centroid,
                            color    = color_name_class.predict(np.mean(car_img[ed_
↪img(lab_img == c_reg.label)], 0)[:3].reshape((1, -1)))[0])
                       for c_reg in all_regions])

fig, m_axs = plt.subplots(np.floor(len(all_regions)/3).astype(int),3, figsize=(10,10))
for c_ax, c_reg in zip(m_axs.ravel(), all_regions):
```

(continues on next page)

(continued from previous page)

```
    c_ax.imshow(car_img[c_reg.bbox[0]:c_reg.bbox[2],
                        c_reg.bbox[1]:c_reg.bbox[3]
                        ])
    c_ax.axis('off')
    c_ax.set_title('Label {}'.format(c_reg.label))
reg_df
```

```
   label                bbox  area                                    centroid  \
0      1  (30, 153, 58, 202)  1091   (43.64527956003666, 177.82218148487627)
1      2    (33, 37, 51, 77)   535     (41.44299065420561, 58.7607476635514)
2      3  (37, 370, 54, 385)   140   (45.871428571428574, 378.6642857142857)
3      4  (37, 387, 54, 416)   342   (44.86257309941521, 398.66959064327483)
4      5  (38, 431, 53, 463)   323   (44.526315789473685, 446.9133126934984)
5      6  (46, 469, 52, 474)    22    (48.63636363636363, 471.1818181818182)
6      7  (46, 475, 54, 482)    44   (49.09090909090909, 478.22727272727275)
7      8  (47, 492, 56, 511)   134  (50.992537313432834, 501.82089552238807)
8      9  (48, 484, 54, 489)    25                          (50.92, 485.76)

           color
0        dimgrey
1         silver
2  darkslategray
3  darkslategray
4  darkslategray
5        dimgrey
6        dimgrey
7     powderblue
8           grey
```


Label 1
Label 2
Label 3
Label 4
Label 5
Label 6
Label 7
Label 8
Label 9

CHAPTER

SIX

OBJECT ANISOTROPY

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

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
```

6.2.1 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
```

6.2.2 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=(10, 7), dpi=100)

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))
    for j, (c_func, c_ax) in enumerate(zip(aiso_funcs, m_axs.flatten()[1:]), 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.flatten()[-1].axis('off')


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

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

CHAPTER

SEVEN

STATISTICAL TOOLS

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)

7.2 Principal Component Analysis

- 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.

7.2.1 PCA on spectroscopy

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

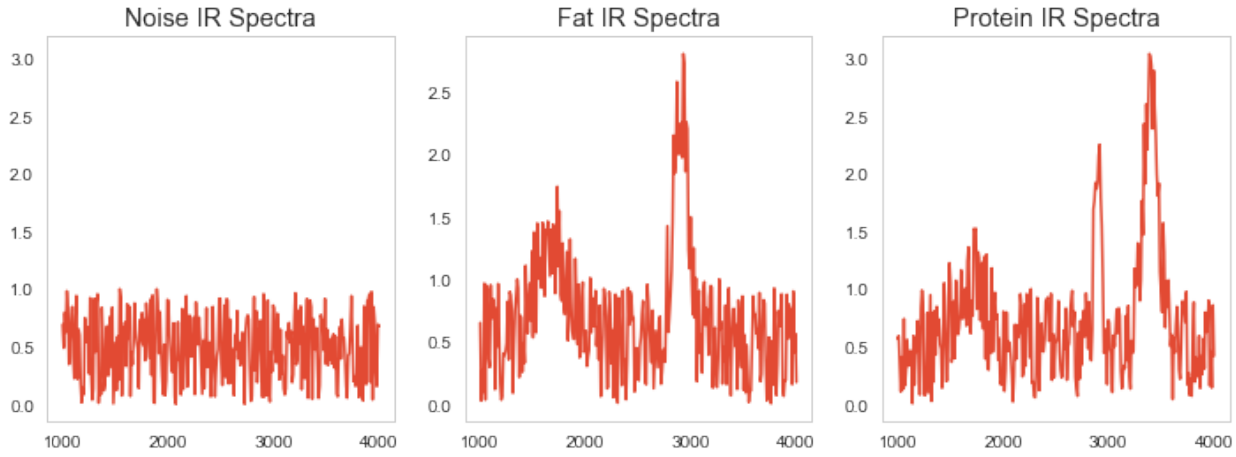
```
cm_dm = np.linspace(1000, 4000, 300)

# 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)]
```

(continues on next page)

(continued from previous page)

```
protein_curve = [(2900, 50, 200), (3400, 100, 600), (1680, 200, 300)]
noise_curve = [(3000, 50, 1)]

fig, (ax0, ax1, ax2) = plt.subplots(1, 3, figsize=(12, 4))

ax1.plot(cm_dm, peaks(fat_curve))
ax1.set_title('Fat IR Spectra')

ax2.plot(cm_dm, peaks(protein_curve))
ax2.set_title('Protein IR Spectra')

ax0.plot(cm_dm, peaks(noise_curve))
ax0.set_title('Noise IR Spectra')

ax0.set_ylim(ax2.get_ylim())
ax2.set_ylim(ax2.get_ylim())

pd.DataFrame({'cm^(-1)': cm_dm, 'intensity': peaks(protein_curve)}).head(5)
```

	cm^(-1)	intensity
0	1000.000000	0.082790
1	1010.033445	0.554745
2	1020.066890	0.340015
3	1030.100334	0.069390
4	1040.133779	0.083648



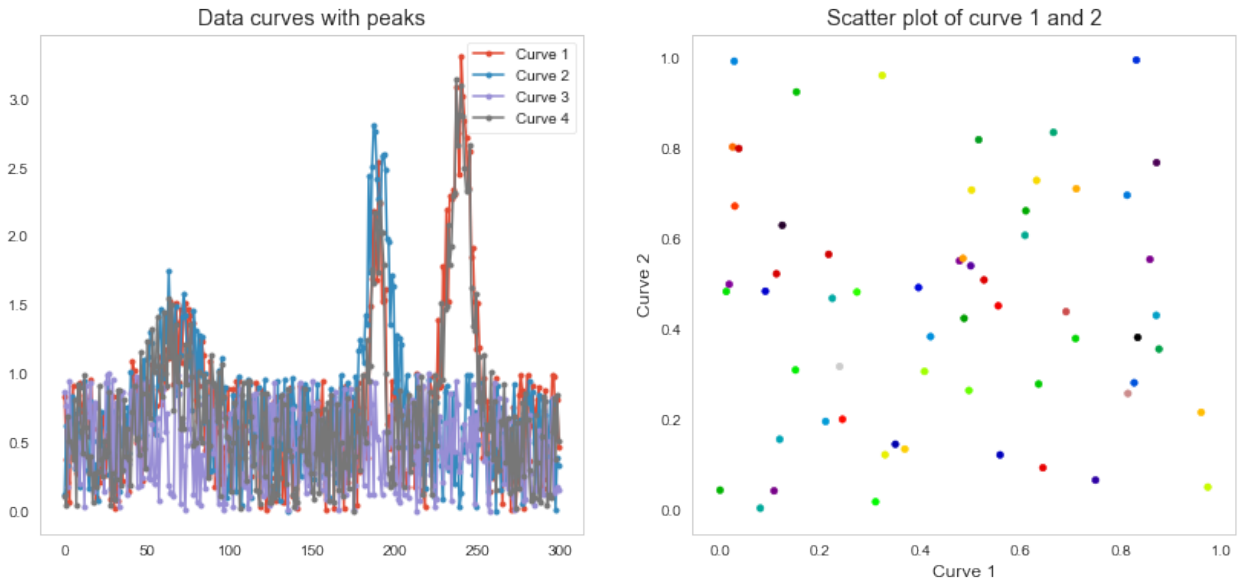
7.3 Test Dataset of a number of curves

We want to sort cells or samples into groups of being

- more fat like
- or more protein like.

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

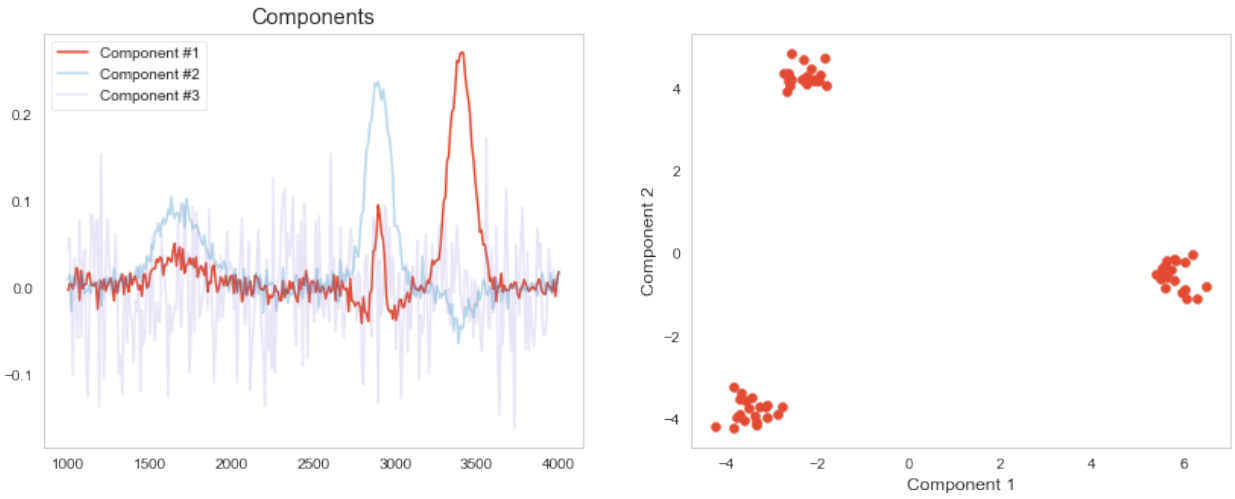
```
test_data = np.stack([peaks(c_curve) for _ in range(20)
                      for c_curve in [protein_curve, fat_curve, noise_curve]], 0)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

ax1.plot(test_data[:4].T, '.-')
ax1.legend(['Curve 1', 'Curve 2', 'Curve 3', 'Curve 4'])
ax1.set_title('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');
```



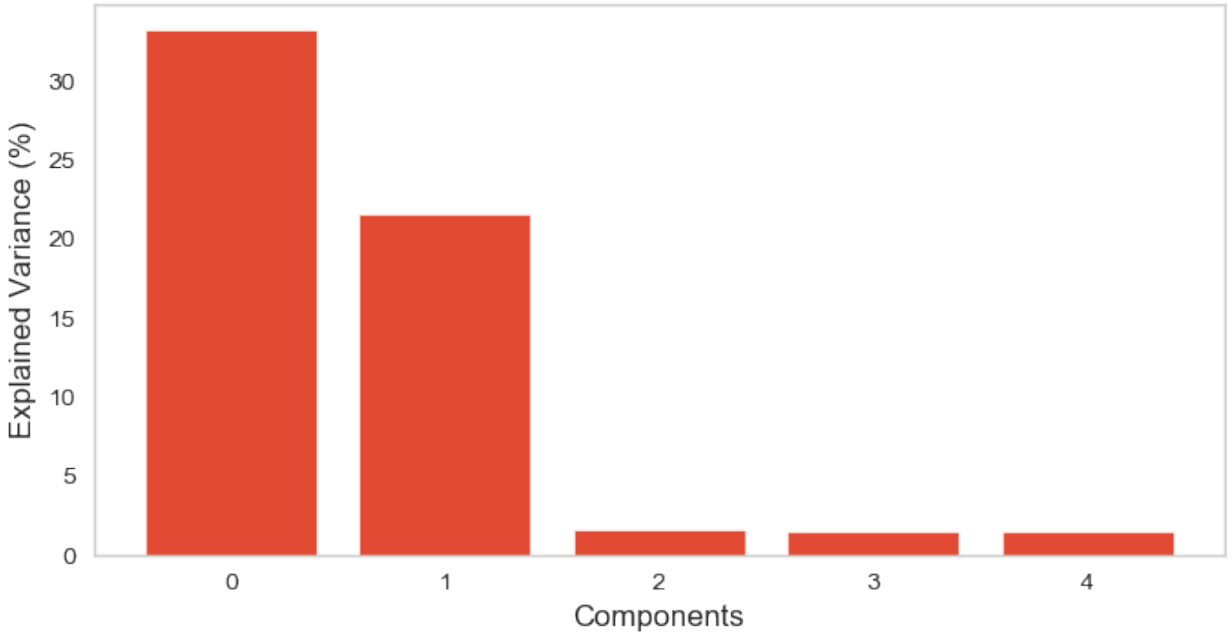

```
from sklearn.decomposition import PCA
pca_tool = PCA(5)
pca_tool.fit(test_data)
```

```
PCA(copy=True, iterated_power='auto', n_components=5, random_state=None,
    svd_solver='auto', tol=0.0, whiten=False)
```

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))
score_matrix = pca_tool.transform(test_data)
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');
```




```
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 (%)');
```



CHAPTER

EIGHT

PRINCIPAL COMPONENT ANALYSIS

8.1 scikit-learn Face Analyis

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)
```

```
downloading Olivetti faces from https://ndownloader.figshare.com/files/5976027 to .
Dataset consists of 400 faces
```

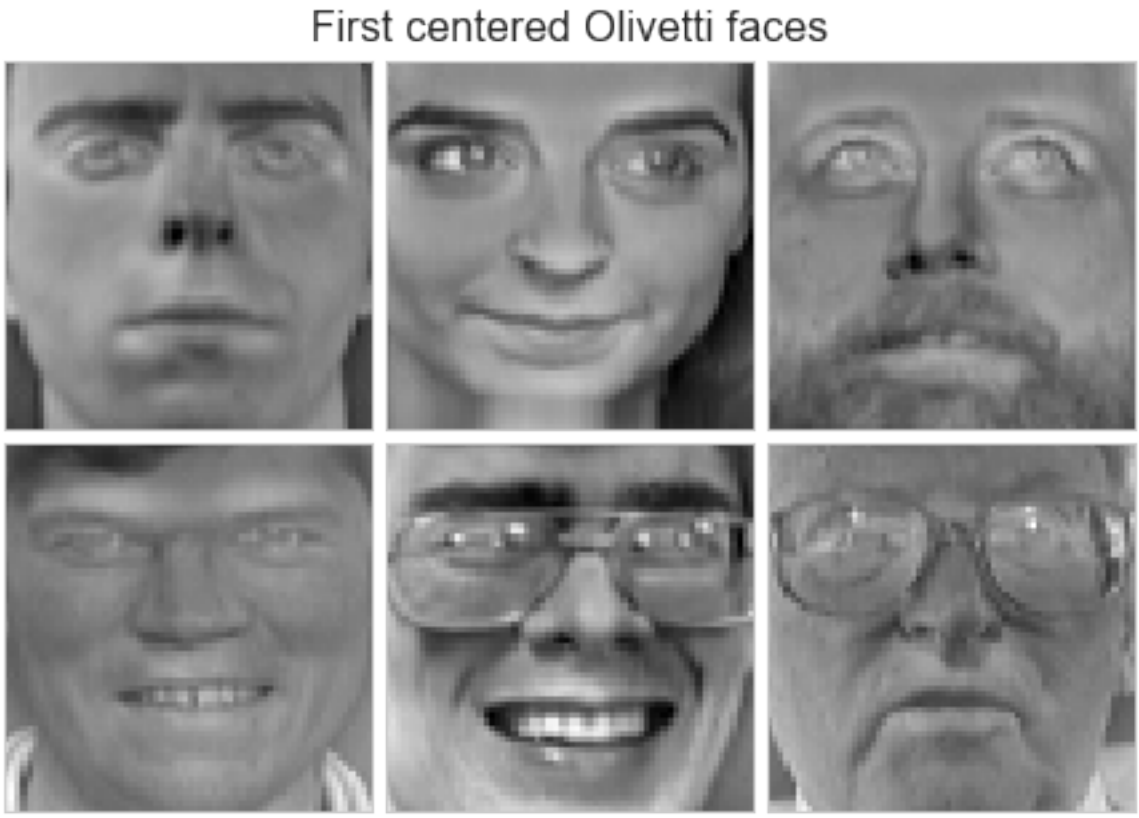
```
def plot_gallery(title, images, n_col=n_col, n_row=n_row):
    plt.figure(figsize=(2. * n_col, 2.26 * n_row))
    plt.suptitle(title, size=16)
    for i, comp in enumerate(images):
        plt.subplot(n_row, n_col, i + 1)
        vmax = max(comp.max(), -comp.min())
        plt.imshow(comp.reshape(image_shape),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

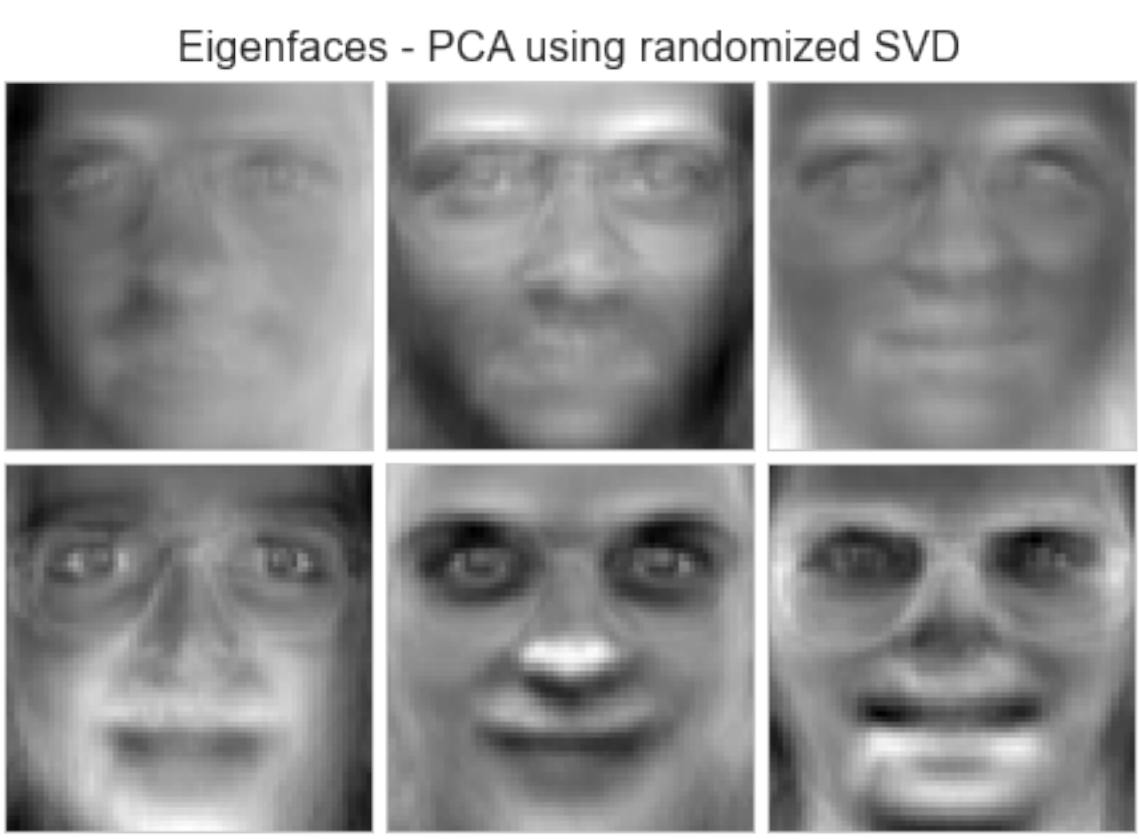
(continues on next page)

(continued from previous page)

```
        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

CHAPTER

NINE

APPLIED PCA: SHAPE TENSOR

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.

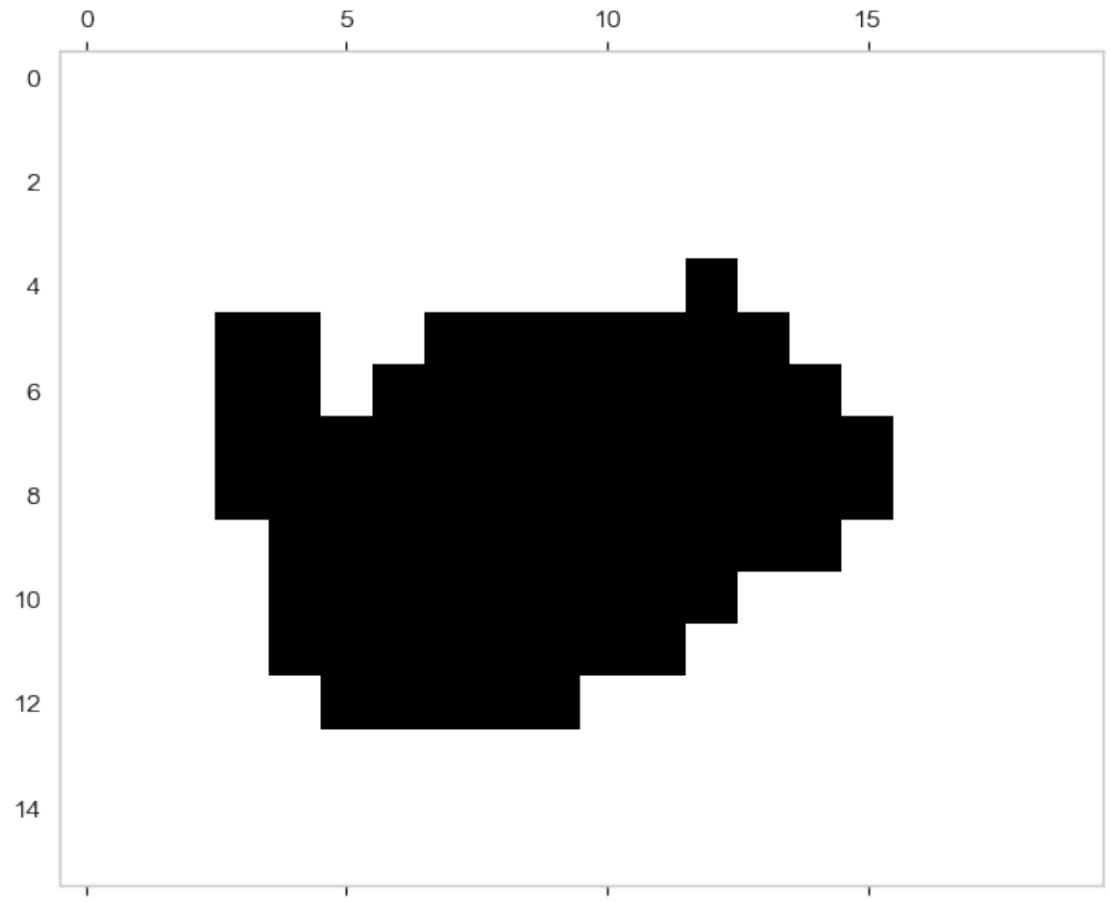
$$COV(I_{id}) = \frac{1}{N} \sum_{\forall \vec{v} \in I_{id}} \begin{bmatrix} \vec{v}_x \vec{v}_x & \vec{v}_x \vec{v}_y & \vec{v}_x \vec{v}_z \\ \vec{v}_y \vec{v}_x & \vec{v}_y \vec{v}_y & \vec{v}_y \vec{v}_z \\ \vec{v}_z \vec{v}_x & \vec{v}_z \vec{v}_y & \vec{v}_z \vec{v}_z \end{bmatrix}$$

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

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

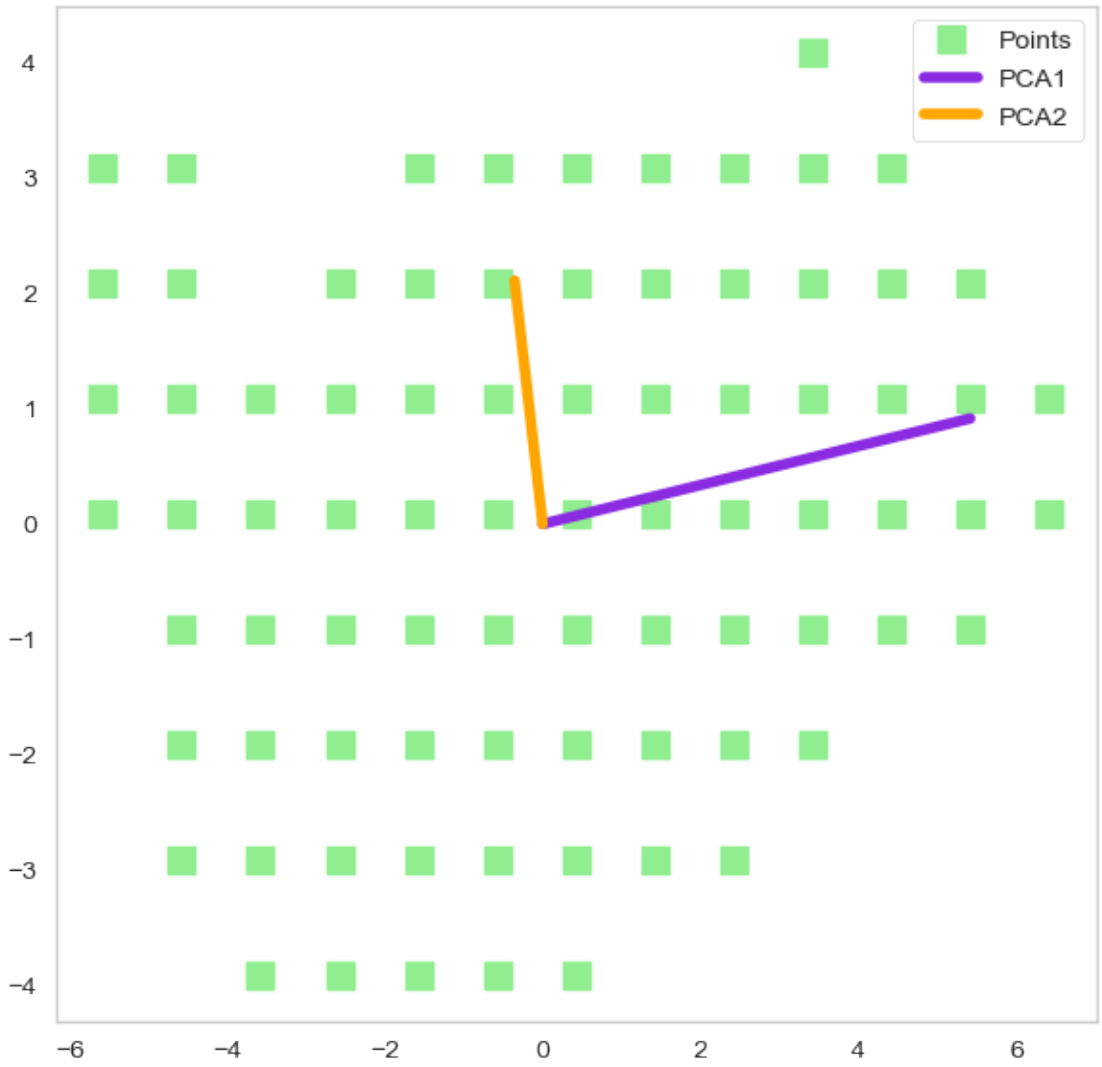
The principal components tell us about the orientation of the object and the scores tell us about the corresponding magnitude (or length) in that direction.

```
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');
```



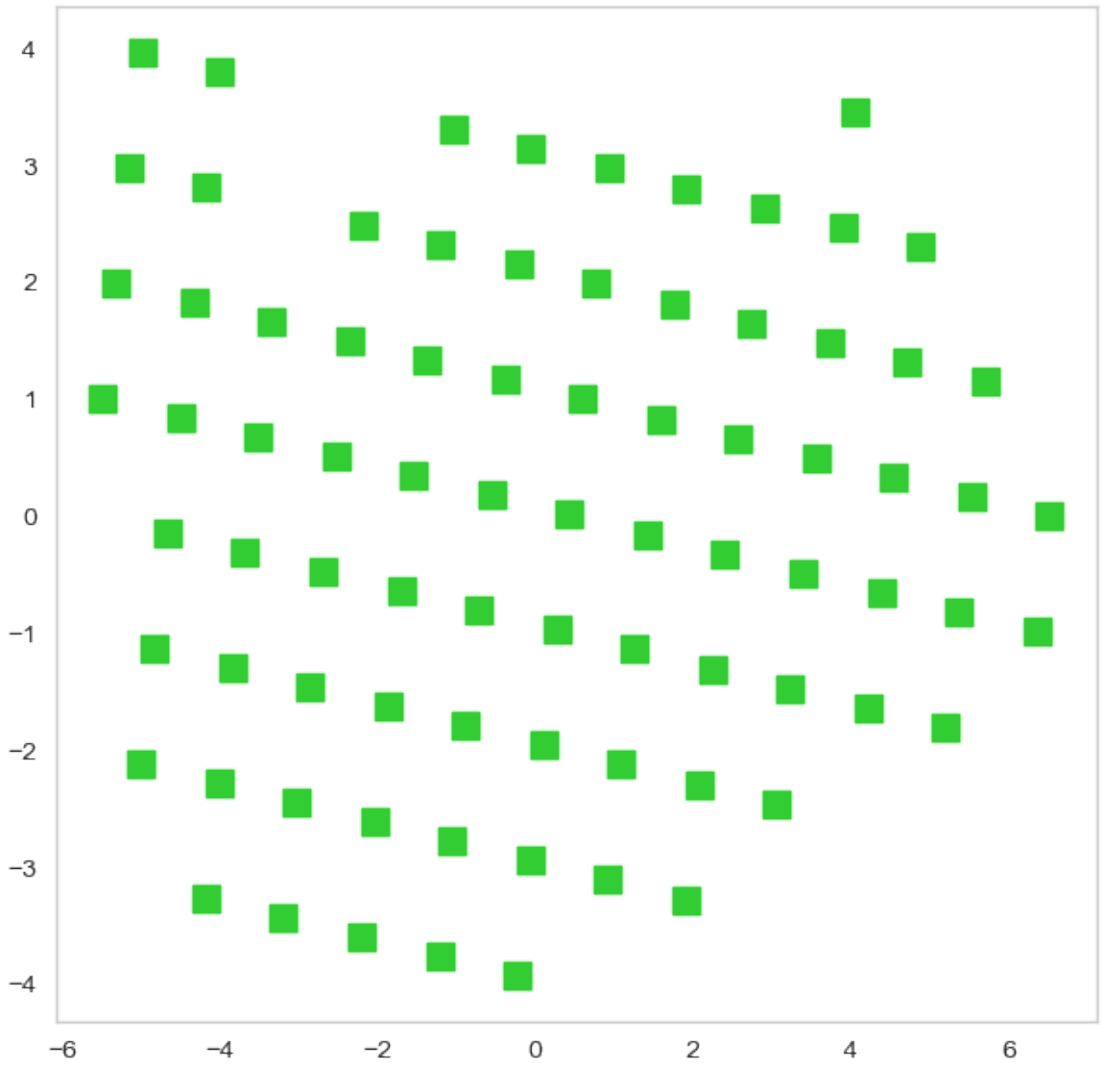
9.1.1 Eigenvectors of the positions

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


ax1.plot(xy_pts[:, 1]-np.mean(xy_pts[:, 1]),
         xy_pts[:, 0]-np.mean(xy_pts[:, 0]), 's', color='lightgreen', label='Points',␣
↪markersize=10)
ax1.plot([0, shape_pca.explained_variance_[0]/2*shape_pca.components_[0, 1]],
         [0, shape_pca.explained_variance_[0]/2*shape_pca.components_[0, 0]], '-',␣
↪color='blueviolet', linewidth=4,
         label='PCA1')
ax1.plot([0, shape_pca.explained_variance_[1]/2*shape_pca.components_[1, 1]],
         [0, shape_pca.explained_variance_[1]/2*shape_pca.components_[1, 0]], '-',␣
↪color='orange', linewidth=4,
         label='PCA2')
ax1.legend();
```



9.1.2 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], 's', color='limegreen', markersize=10);
```



9.2 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.

9.3 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

9.3.1 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 ($\updownarrow = \sqrt{5\lambda_i}$), derivation similar to moment of inertia tensor for ellipsoids.

CHAPTER

TEN

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.

10.1 Meshing

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)

10.2 Marching Cubes

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).

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

1. The image is processed one voxel at a time
2. The 2x2x2 neighborhood is checked at every voxel.

3. From this configuration of values, faces are added to the mesh to incorporate the most simple surface which would explain the values.

This algortihm is nicely explained in this video

Marching tetrahedra is for some applications a better suited approach

CHAPTER

ELEVEN

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

11.1 Why

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

11.2 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

CHAPTER

TWELVE

ADVANCED SHAPE AND TEXTURE

Quantitative Big Imaging ETHZ: 227-0966-00L

Part 2

12.1 Literature / Useful References

12.1.1 Books

- Jean Claude, Morphometry with R
- Online through ETHZ
- Buy it
- John C. Russ, "The Image Processing Handbook",(Boca Raton, CRC Press)
- Available online within domain ethz.ch (or proxy.ethz.ch / public VPN)

12.1.2 Papers / Sites

- Thickness
- 1 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. doi:10.1046/j.1365-2818.1997.1340694.x
- Curvature
- http://mathworld.wolfram.com/MeanCurvature.html
- [2] "Computation of Surface Curvature from Range Images Using Geometrically Intrinsic Weights"*, T. Kurita and P. Boulanger, 1992.
- http://radiomics.io

12.2 Outline

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

- Characteristic Shapes
- Texture Analysis

12.3 Learning Objectives

12.3.1 Motivation (Why and How?)

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?

Patterns

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

12.4 Let's load some modules for the notebook

```
import os
from tqdm import tqdm

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


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

(continues on next page)

(continued from previous page)

```
from scipy.ndimage import distance_transform_edt

import skimage.transform
from skimage.feature.texture import greycomatrix
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 plotly.offline as py
import plotly.graph_objs as go
import plotly.figure_factory as FF

import ipyvolume as p3




%matplotlib inline

plt.rcParams["figure.figsize"] = (8, 8)
plt.rcParams["figure.dpi"] = 150
plt.rcParams["font.size"] = 14
plt.rcParams['font.family'] = ['sans-serif']
plt.rcParams['font.sans-serif'] = ['DejaVu Sans']
plt.style.use('ggplot')
sns.set_style("whitegrid", {'axes.grid': False})
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-1-1954032d1812> in <module>
      3
      4 import numpy as np
----> 5 import pandas as pd
      6 import matplotlib.pyplot as plt
      7 import seaborn as sns

ModuleNotFoundError: No module named 'pandas'
```

CHAPTER

THIRTEEN

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)
- Different metrics are used:
  - Euclidean (and approximations)
  - City block (4-connected)
  - Checkerboard (8-connected)

13.1 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.

    In addition to the distance transform, the feature transform can
    be calculated. In this case the index of the closest background
    element is returned along the first axis of the result.

    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 int, or sequence of same, 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 return distance matrix. At least one of
        return_distances/return_indices must be True. Default is True.
    return_indices : bool, optional
        Whether to return indices matrix. Default is False.
```

(continues on next page)

(continued from previous page)

```
    distances : ndarray, optional
        Used for output of distance array, must be of type float64.
    indices : ndarray, optional
        Used for output of indices, must be of type int32.

    Returns
    -------
    distance_transform_edt : ndarray or list of ndarrays
        Either distance matrix, index matrix, or a list of the two,
        depending on `return_x` flags and `distance` and `indices`
        input parameters.

    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
    >>> 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],
```

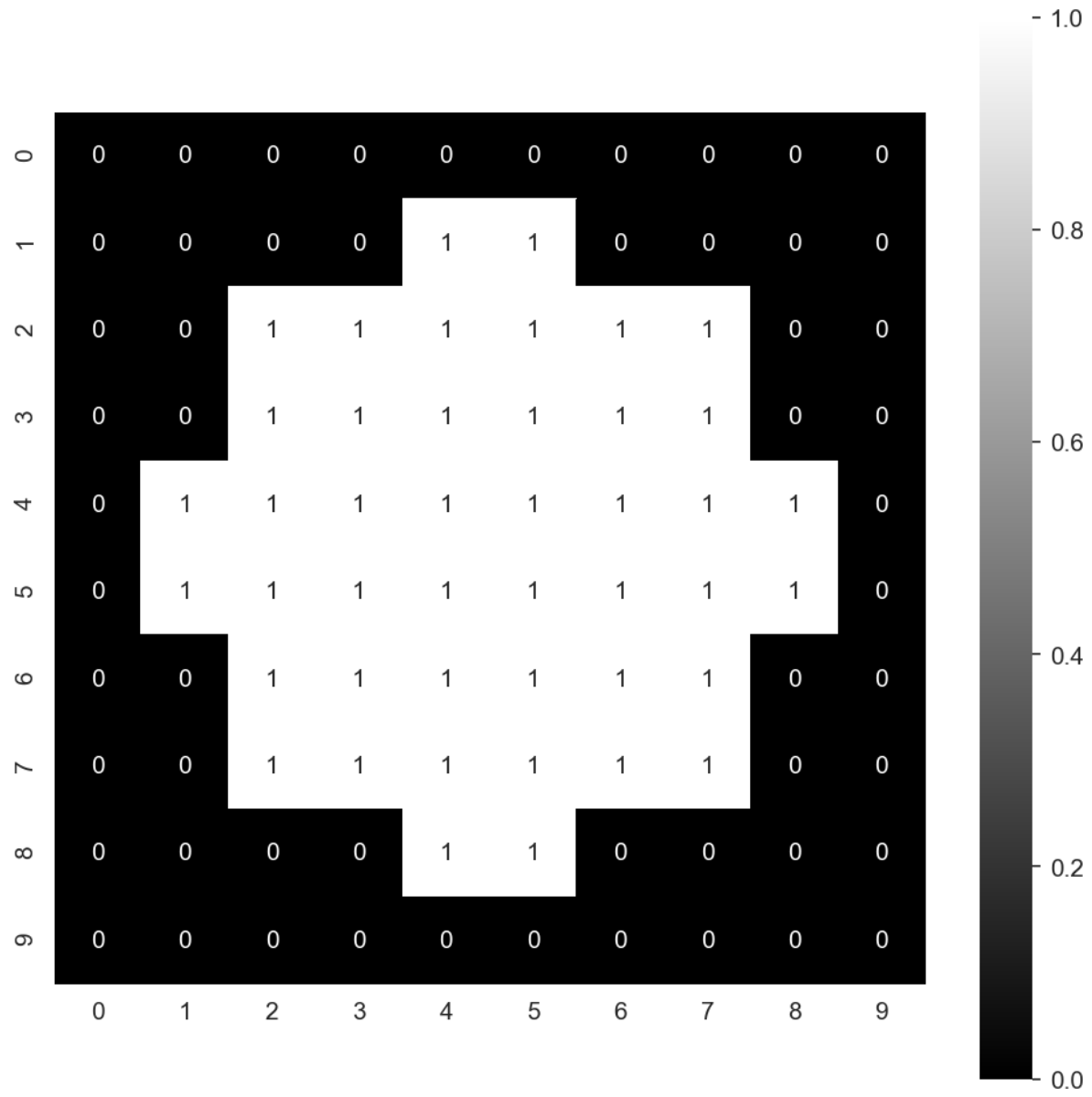
(continues on next page)

(continued from previous page)

```
            [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)
    >>> 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]]])
```

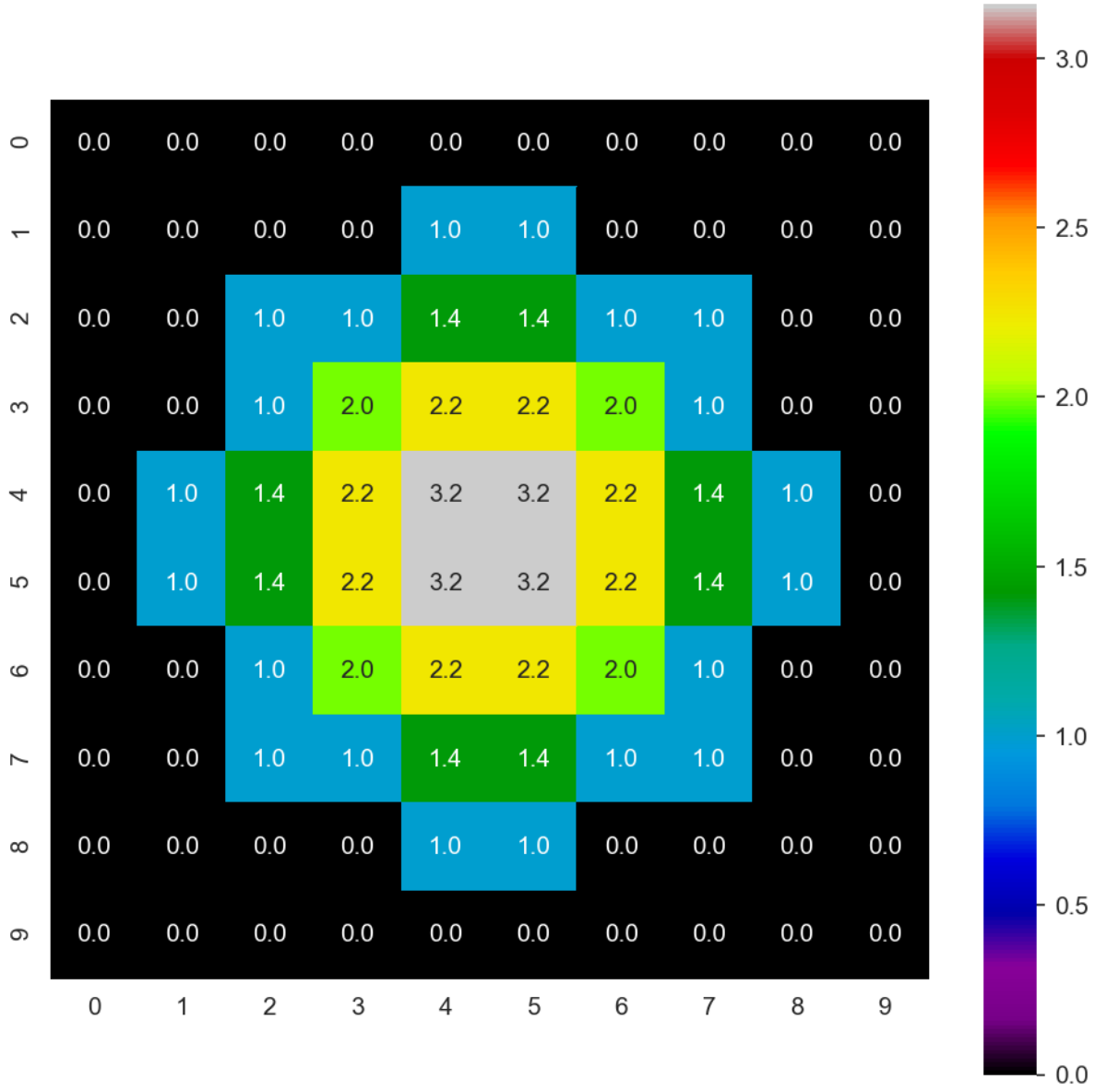
13.2 What does a distance map look like?

```
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)

sns.heatmap(img_bw, annot=True, fmt="d", cmap='gray'); plt.gca().set_aspect(aspect=
↪'equal')
```



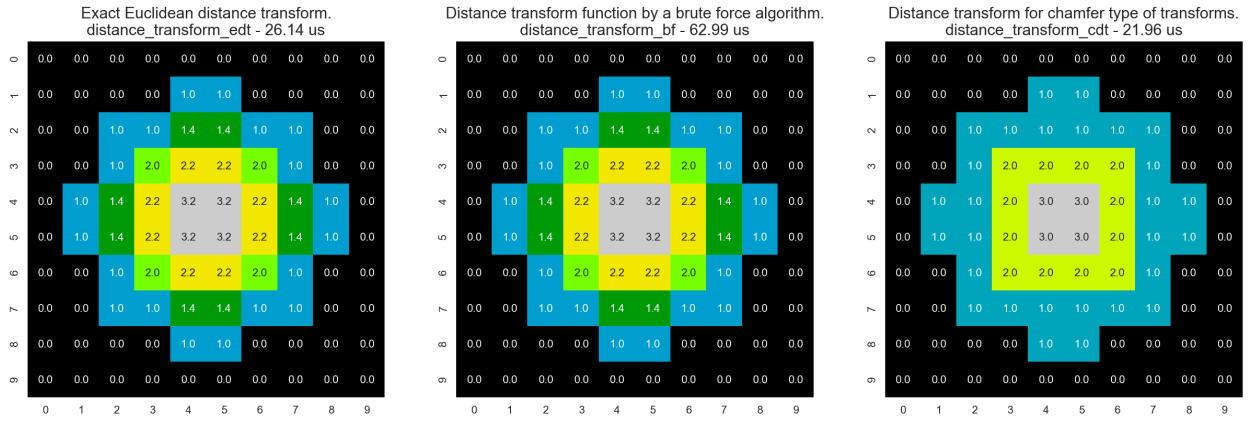
13.2.1 The Euclidean distance map

```
dmap = ndimage.distance_transform_edt(img_bw)
sns.heatmap(dmap, annot=True,
            fmt="2.1f", cmap='nipy_spectral'); plt.gca().set_aspect(aspect='equal')
```



13.2.2 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)
    sns.heatmap(dmap, annot=True,
                fmt="2.1f", cmap='nipy_spectral', ax=c_ax, cbar=False)
    c_ax.set_title('{}\n{} - {} us'.format(dmap_func.__doc__.split('\n')[1].strip(),
                                           dmap_func.__name__,
                                           '%2.2f' % ms_time))
```



13.3 Why does speed matter?

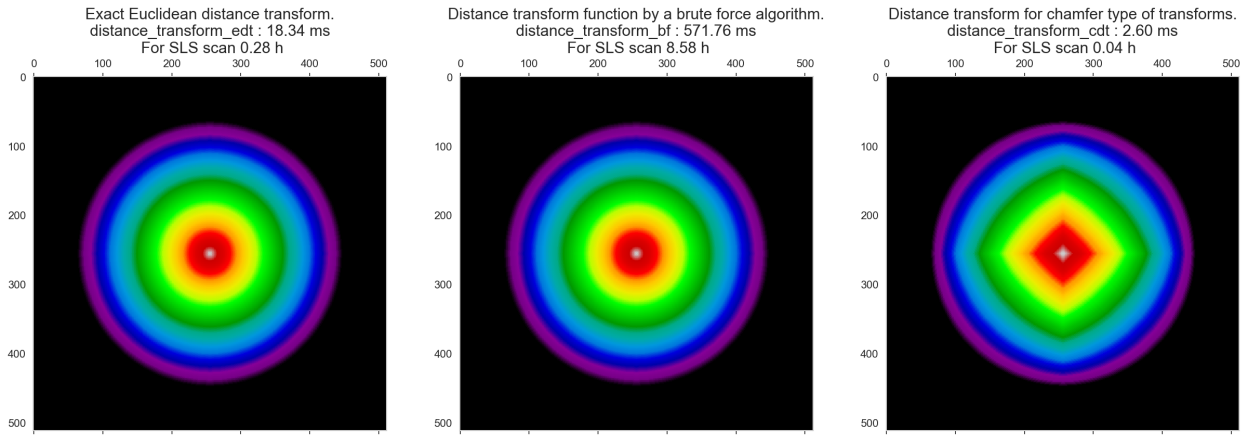
As for the question why speed matters,

- 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

13.3.1 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)))
```



13.4 Distance map - Definition

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

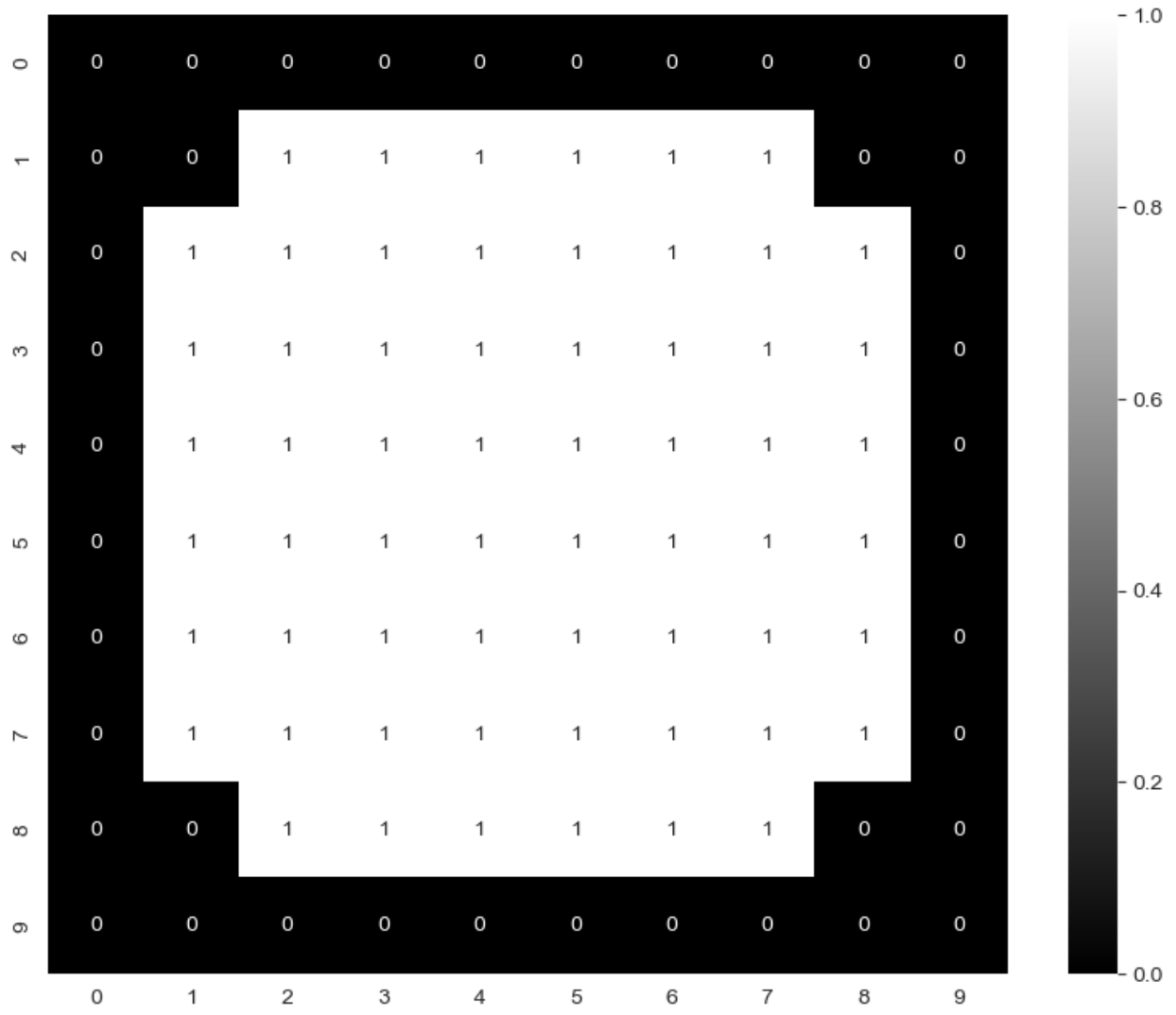
- $Im(x,y) = \{\text{Foreground, Background}\}$
- We can define a distance map operator ($dist$) that transforms the image into a distance map

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

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]:
                    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
```

```
fig, ax=plt.subplots(1,1, figsize=(10,8), dpi=100)
img_bw = generate_dot_image(10, 1.0)
sns.heatmap(img_bw, ax=ax, annot=True,
            fmt="d", cmap='gray');
ax.set_aspect(aspect='equal')
```



13.4.1 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()
    sns.heatmap(img_list[i],annot=True,
                fmt="2.1f",cmap='nipy_spectral',ax=c_ax, cbar=False,
                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()
```

(continues on next page)

(continued from previous page)

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

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

13.5 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:
                i_xy = last_img[x, y]
                for xp in [-1, 0, 1]:
                    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
```

13.5.1 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()
    sns.heatmap(img_list[i], annot=True, fmt="2.0f",
                cmap='nipy_spectral', ax=c_ax, cbar=False,
                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>
```

13.6 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**

13.6.1 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

13.7 What does the Distance Maps show?

We can create 2 distance maps

1. Foreground → Background

- Information about the objects size and interior

1. Background → Foreground

- Information about the distance / space between objects

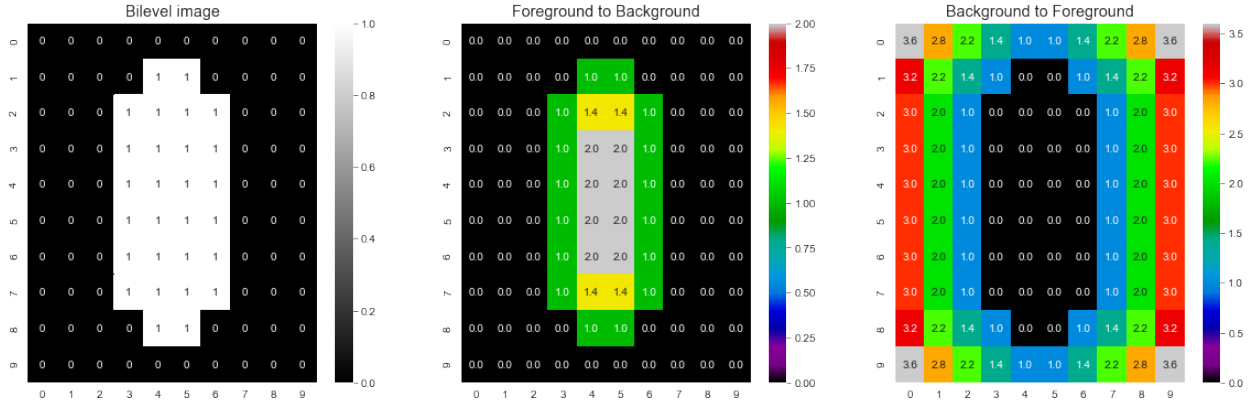
```
import os
import seaborn as sns
import matplotlib.pyplot as plt
from scipy import ndimage
import skimage.transform
import numpy as np
%matplotlib inline
```

```
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))
sns.heatmap(img_bw, annot=True,
```

(continues on next page)

(continued from previous page)

```
            fmt="d", cmap='gray', ax=ax1), ax1.set_title('Bilevel image')

sns.heatmap(ndimage.distance_transform_edt(img_bw),
            annot=True,
            fmt="2.1f", cmap='nipy_spectral', ax=ax2), ax2.set_title('Foreground to␣
↪Background')

sns.heatmap(ndimage.distance_transform_edt(1-img_bw),
            annot=True,
            fmt="2.1f", cmap='nipy_spectral', ax=ax3),ax3.set_title('Background to␣
↪Foreground');
```



13.8 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

```
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
    sns.heatmap(img_list[i],
                annot=True,
                fmt="2.0f",
                cmap='nipy_spectral',
                ax=c_ax,
                cbar=False,
                vmin=0,
                vmax=5)
    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])))
```

(continues on next page)

(continued from previous page)

```
    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]

    sns.heatmap(lab_img,
                annot=True,
                fmt="d",
                cmap='nipy_spectral',
                ax=c_lab,
                cbar=False,
                vmin=0,
                vmax=2)
    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')

fig, m_axs = plt.subplots(2, 3, figsize=(12, 8), dpi=100)

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)
    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')
```

13.8.1 Animate the distances

```
HTML(anim_code)
```

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

CHAPTER

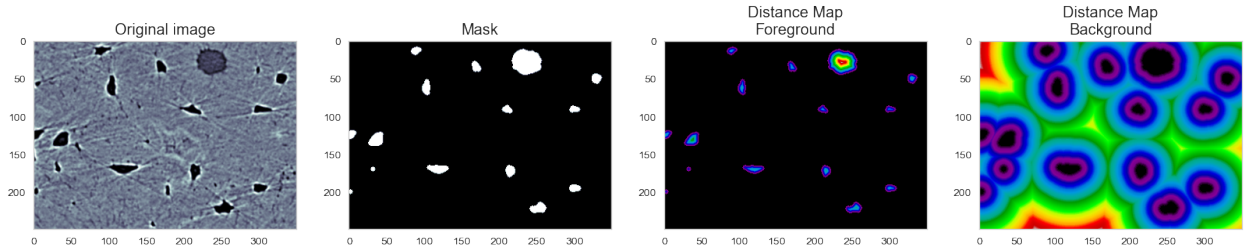
FOURTEEN

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

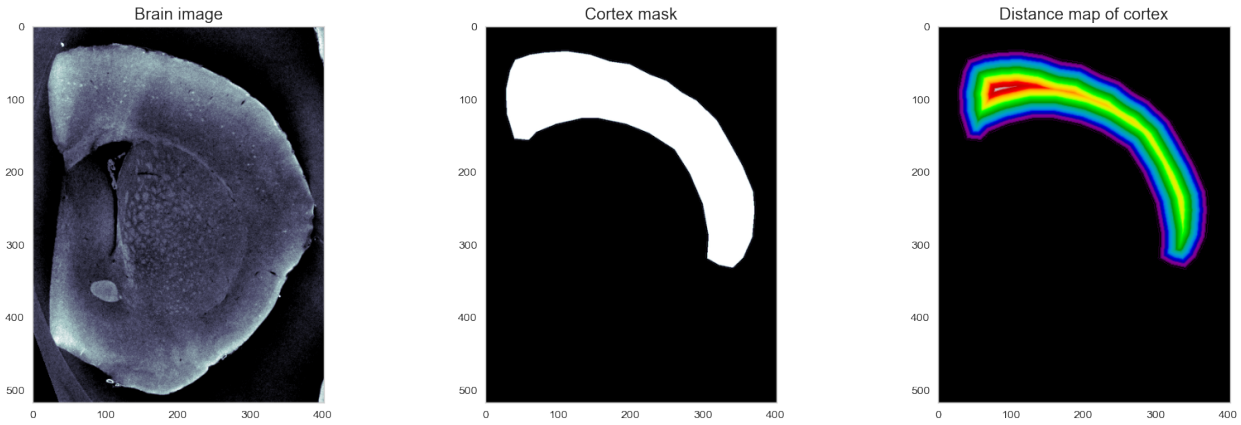
14.1 The bone slice

```
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');
```



14.2 The cortex

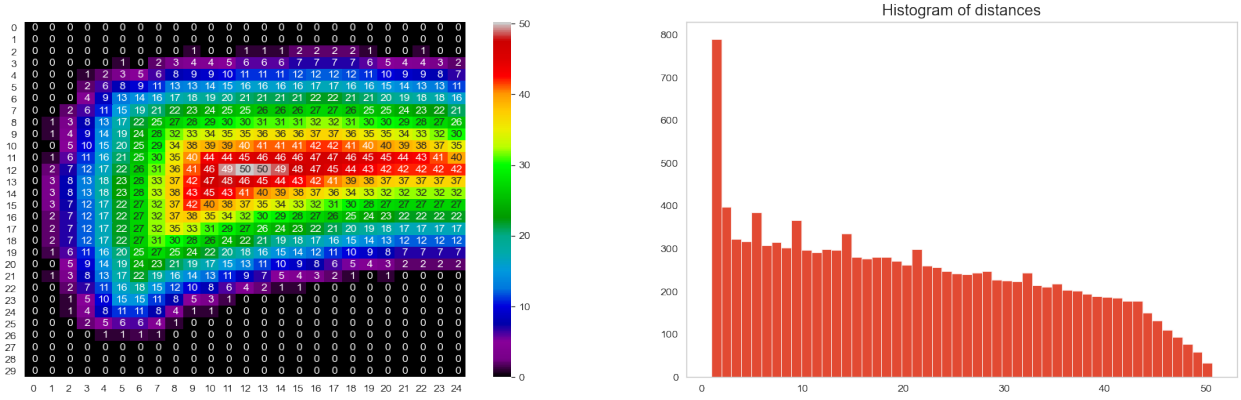
```
cortex_img = imread("figures/example_poster.tif")[::2, ::2]/2048
cortex_mask = imread("figures/example_poster_mask.tif")[::1, ::1, 0]/255.0
cortex_dmap = distance_transform_edt(cortex_mask)
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 6), dpi=100)

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
↪');
```



14.3 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), dpi=100)
cortex_dmap_roi = cortex_dmap[25:175, 25:150]
sns.heatmap(cortex_dmap_roi[::5, ::5],
            annot=True,
            fmt="2.0f",
            cmap='nipy_spectral',
            ax=ax1,
            cbar=True);

ax2.hist(cortex_dmap_roi[cortex_dmap_roi > 0].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
```

CHAPTER

FIFTEEN

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

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

15.1 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

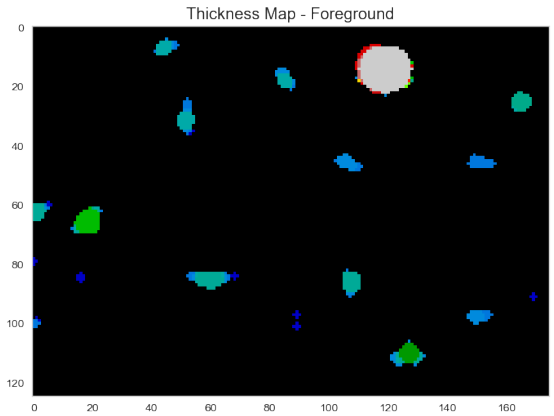
15.1.1 A thickness map implementation

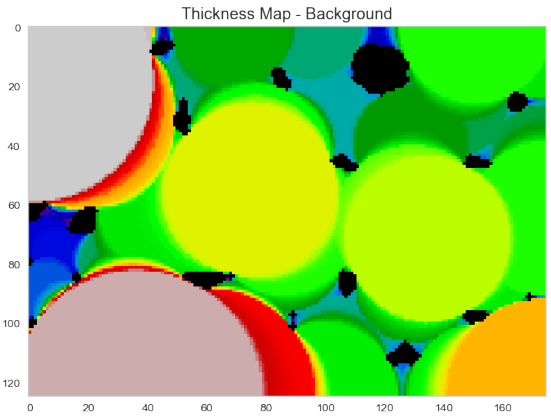
```
def simple_thickness_func(distance_map):
    dm = distance_map.ravel()
    th_map = distance_map.ravel().copy()
    xx, yy = [v.ravel() for v in np.meshgrid(range(distance_map.shape[1]),
                                             range(distance_map.shape[0]))]

    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)
```

15.2 Thickness map on foregrund and background

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

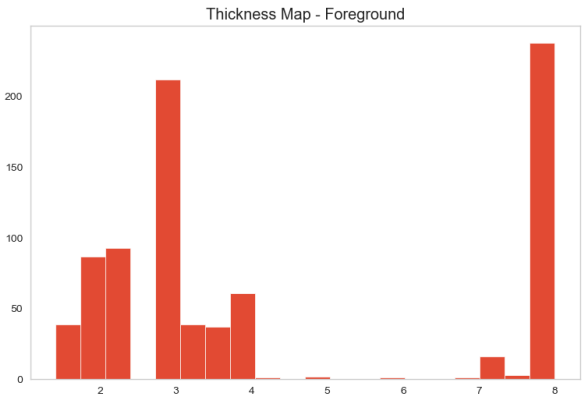
```
fig, (ax3, ax4) = plt.subplots(1, 2, figsize=(20, 6), dpi=100)

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
↪');
```

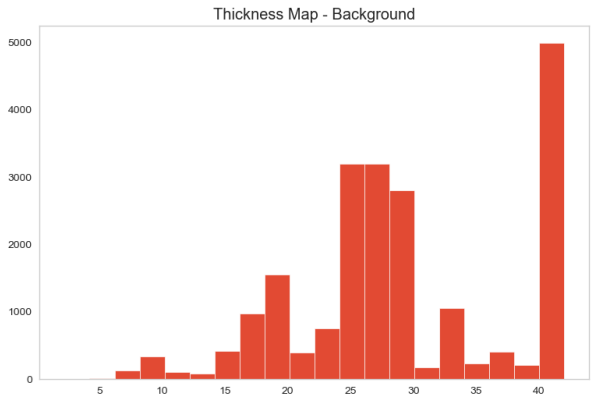




15.2.1 Thickness distributions

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

ax3.hist(fg_th_map[fg_th_map > 0].ravel(), 20); ax3.set_title('Thickness Map -␣
↪Foreground');
ax4.hist(bg_th_map[bg_th_map > 0].ravel(), 20); ax4.set_title('Thickness Map -␣
↪Background');
```





CHAPTER

SIXTEEN

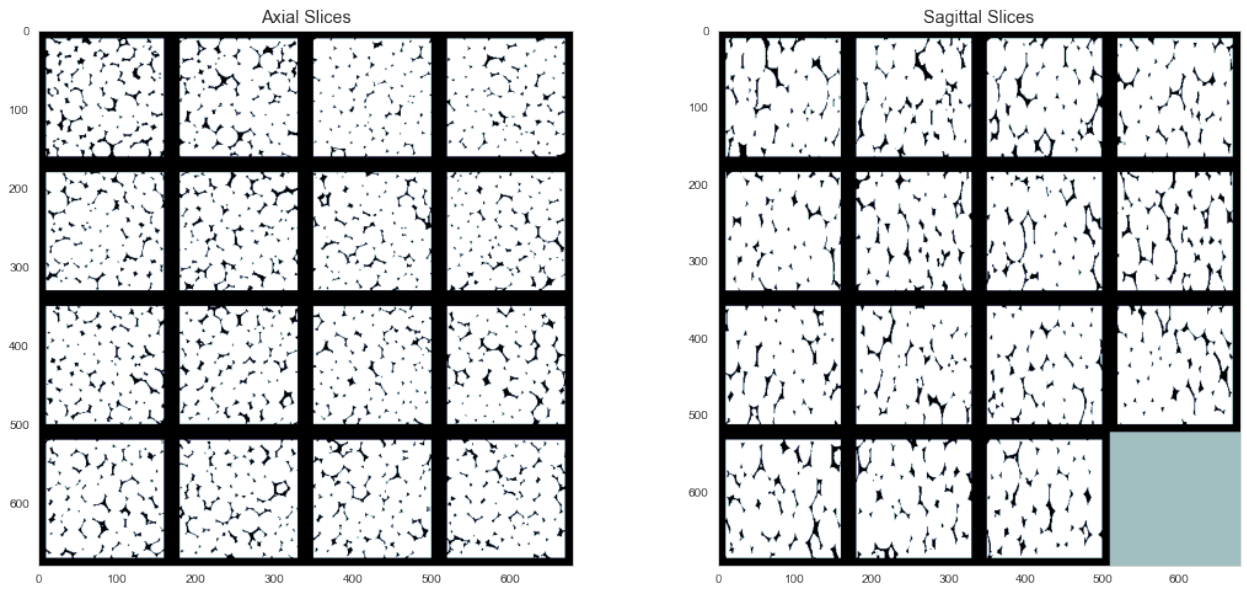
DISTANCE MAPS IN 3D

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

16.1 Let's load a 3D image

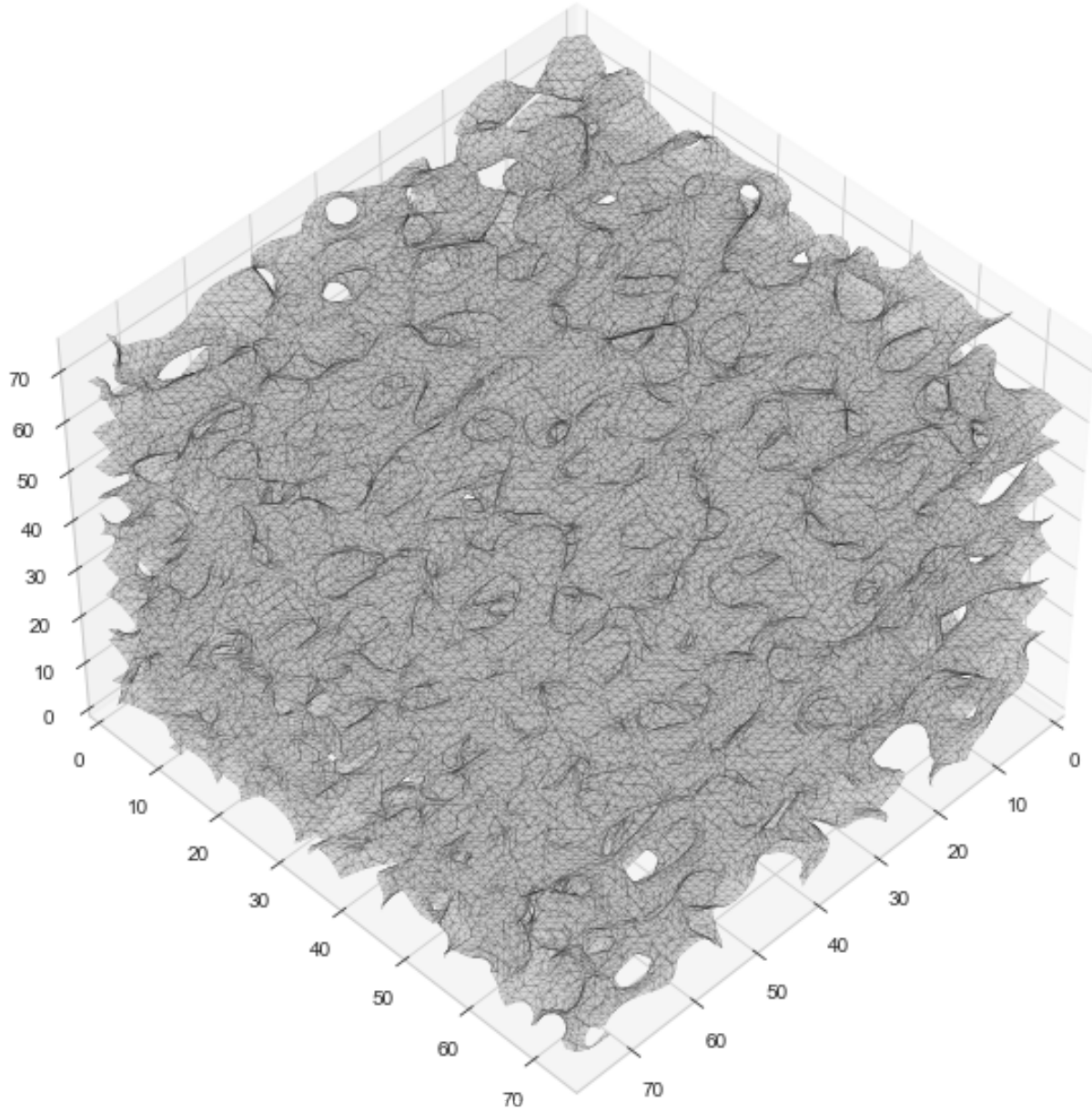
```
def montage_pad(x): return montage2d(
    np.pad(x, [(0, 0), (10, 10), (10, 10)], mode='constant', constant_values=0))

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
```



16.1.1 Inspecting the data in 3D

```
def show_3d_mesh(image, thresholds, edgecolor='none', alpha=0.5):
    p = image[::-1].swapaxes(1, 2)
    cmap = plt.cm.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)
        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');
```



16.1.2 Rendering the mesh as a surface

```
py.init_notebook_mode()
verts, faces, _, _ = measure.marching_cubes(
    # you can make it bigger but the file-size gets HUUUEGE
    smooth_pt_img[:20, :20, :20],
    smooth_pt_img.mean())
x, y, z = zip(*verts)
ff_fig = FF.create_trisurf(x=x, y=y, z=z,
                           simplices=faces,
                           title="Foam Plataeu Border",
                           aspectratio=dict(x=1, y=1, z=1),
```

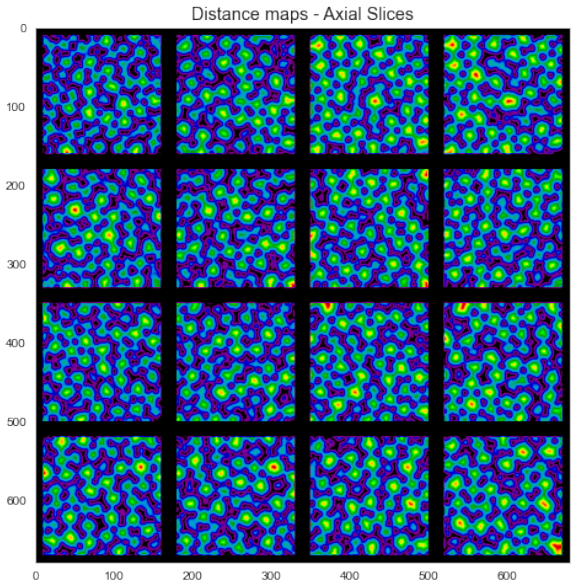
(continues on next page)

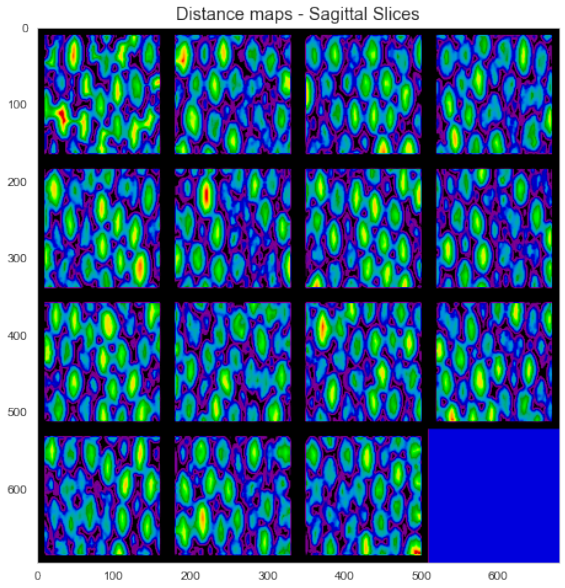
(continued from previous page)

```
                              plot_edges=False)
c_mesh = ff_fig['data'][0]
c_mesh.update(lighting=dict(ambient=0.18,
                            diffuse=1,
                            fresnel=0.1,
                            specular=1,
                            roughness=0.1,
                            facenormalsepsilon=1e-6,
                            vertexnormalsepsilon=1e-12))
c_mesh.update(flatshading=False)
py.iplot(ff_fig)
```

16.2 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');
```





CHAPTER

SEVENTEEN

THICKNESS IN 3D IMAGES

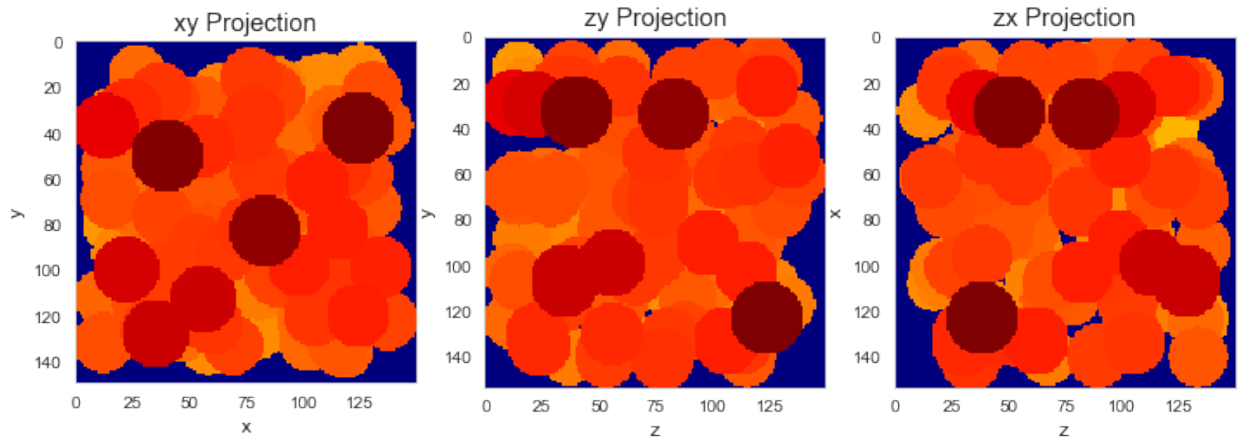
While the examples and demonstrations so far have been shown in 2D, the same exact technique can be applied to 3D data as well. For example for this liquid foam structure

- 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.

```
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 = (np.power(xx-float(y), 2) +
                  np.power(yy-float(x), 2) +
                  np.power(zz-float(z), 2)) <= np.power(cur_bubble_radius, 2)
    thickness_map[cur_bubble] = cur_bubble_radius
```

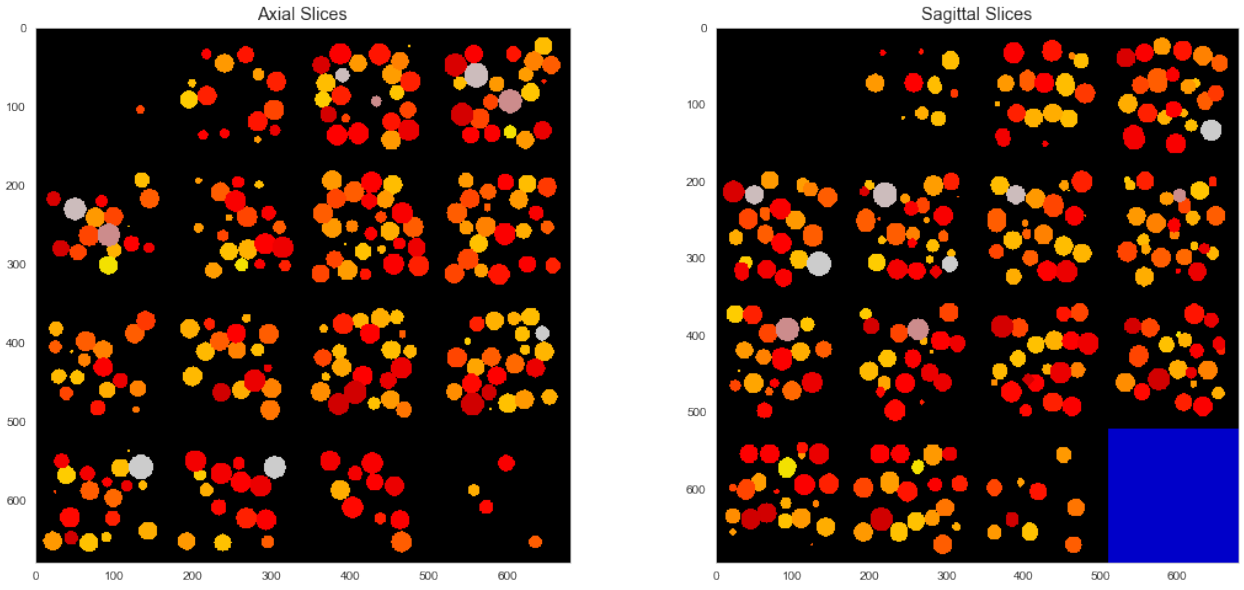
```
Found 180 bubbles
```

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 4))
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('%s Projection' % clabel)
    cax.set_xlabel(clabel[0])
    cax.set_ylabel(clabel[1])
```




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

```
Text(0.5, 1.0, 'Sagittal Slices')
```



CHAPTER

EIGHTEEN

INTERACTIVE 3D VIEWS

Here we can show the thickness map in an interactive 3D manner using the ipyvolume tools (probably only works in the Chrome browser)

```
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')
```

```
/Users/kaestner/opt/anaconda3/lib/python3.8/site-packages/ipyvolume/serialize.py:92:␣
↪RuntimeWarning:

invalid value encountered in true_divide
```

```
VBox(children=(Figure(camera=PerspectiveCamera(fov=45.0, position=(0.0, 0.0, 2.0),␣
↪projectionMatrix=(1.0, 0.0,...
```

CHAPTER

NINETEEN

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)

CHAPTER

TWENTY

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 affect of a single dilation or erosion step.

CHAPTER

TWENTYONE

MESHING (AGAIN)

Constructing a mesh for an image provides very different information than the image data itself. Most crucially this comes when looking at physical processes like deformation.

While the images are helpful for visualizing we rarely have models for quantifying how difficult it is to turn a pixel **off**

If the image is turned into a mesh we now have a list of vertices and edges. For these vertices and edges we can define forces. For example when 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.

CHAPTER

TWENTYTWO

MARCHING CUBES

22.1 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.

22.2 How

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.

Marching tetrahedra is for some applications a better suited approach

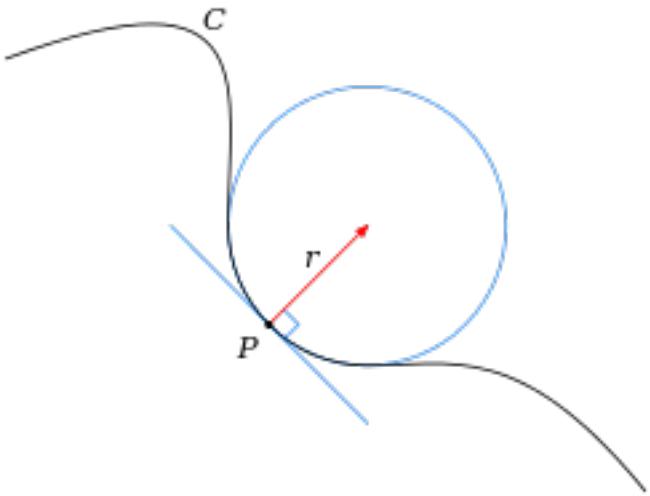
CHAPTER

TWENTYTHREE

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

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 $\rightarrow \vec{N}$

23.2 Curvature: 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

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

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

23.2.1 Mean Curvature

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

23.2.2 Gaussian Curvature

The mean 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

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.

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

CHAPTER

TWENTYFOUR

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.

CHAPTER

TWENTYFIVE

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 are not characterized well be ellipsoids or thickness maps
- Provides a scale-free metric for assessing structures
- Can provide visual indications of structural changes

CHAPTER

TWENTYSIX

OTHER TECHNIQUES

CHAPTER

TWENTYSEVEN

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

27.1 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

CHAPTER

TWENTYEIGHT

TEXTURE ANALYSIS

The world is full of patterns (aka Textures)

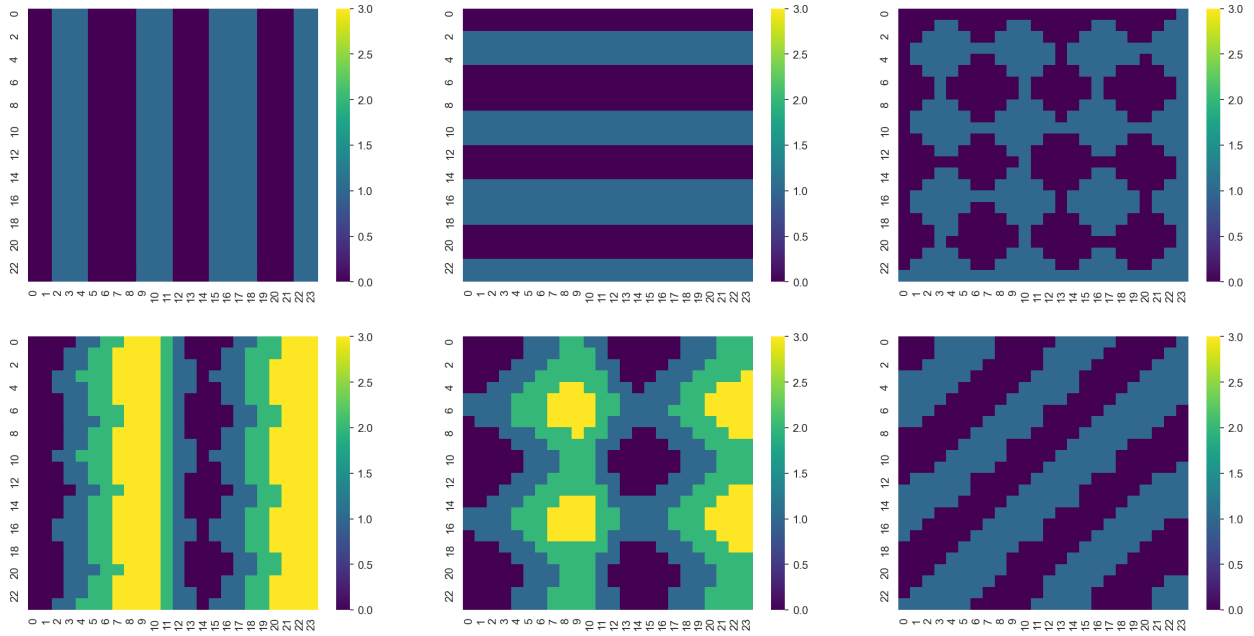
Example taken from

Some metrics to characterize textures Haralick texture features

28.1 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):
    sns.heatmap(c_img, annot=False, fmt='2d', ax=c_ax,
                cmap='viridis', vmin=0, vmax=3)
```



28.2 Methods to characterize textures: 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 $\rightarrow$ for every possible offset $\vec{r} = (\Delta x, \Delta y)$ 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 greycomatrix(image=in_img,
                        distances=dist_list,
                        angles=angle_list,
                        levels=4)
```

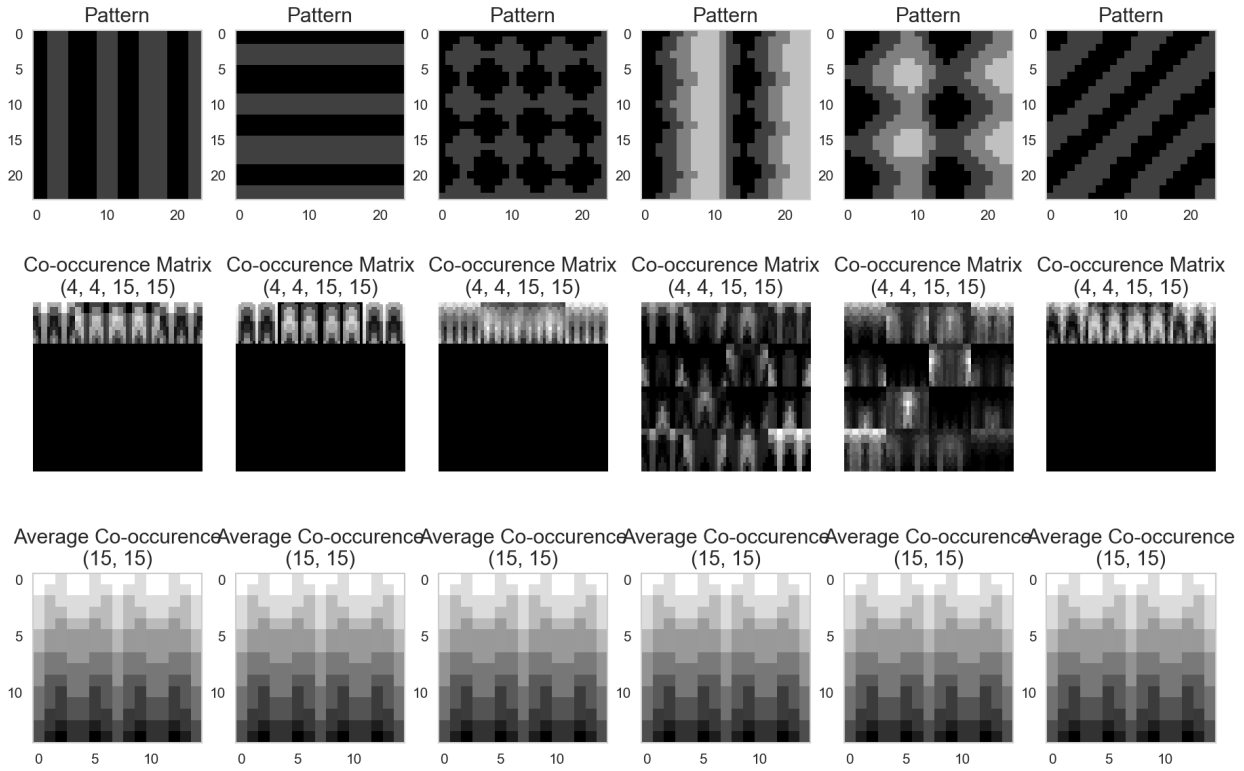
(continues on next page)

(continued from previous page)

```
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])).head(5)
```

```
       E    i    j    d     theta
0  192.0  0.0  0.0  1.0  0.000000
1  192.0  0.0  0.0  1.0  0.448799
2  184.0  0.0  0.0  1.0  0.897598
3  276.0  0.0  0.0  1.0  1.346397
4  276.0  0.0  0.0  1.0  1.795196
```

```
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))
    d_ax.axis('off')
    avg_coo_matrix = np.mean(full_coo_matrix*1.0, (0, 1))
    f_ax.imshow(avg_coo_matrix, cmap='gray')
    f_ax.set_title('Average Co-occurence\n{}'.format(avg_coo_matrix.shape))
```



28.3 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.head(5)
```

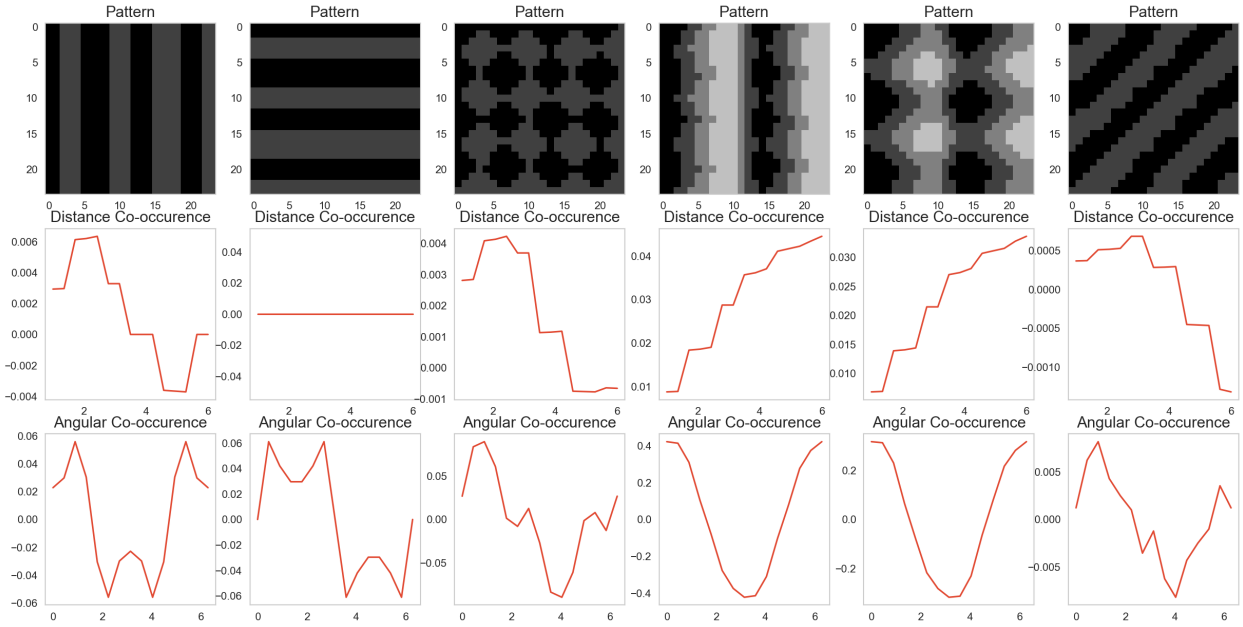
```
   ij_diff    d     theta    E
0     -3.0  1.0  0.000000  0.0
1     -3.0  1.0  0.448799  0.0
2     -3.0  1.0  0.897598  0.0
3     -3.0  1.0  1.346397  0.0
4     -3.0  1.0  1.795196  0.0
```

```
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')
```

(continues on next page)

(continued from previous page)

```
    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')
```



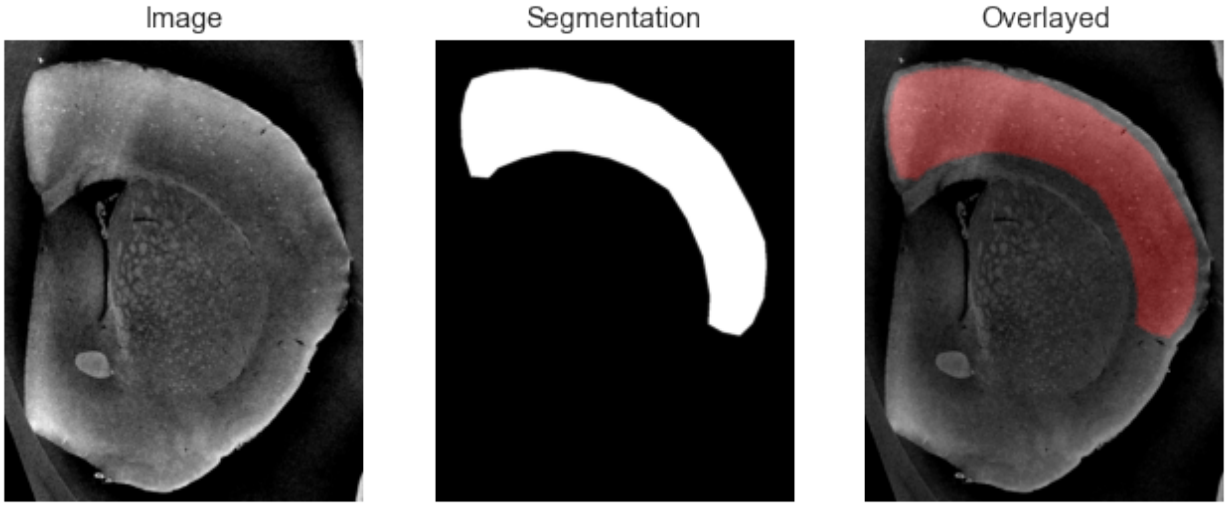
28.4 Applying Texture to Brain

```
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))
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
```

(continues on next page)

(continued from previous page)

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



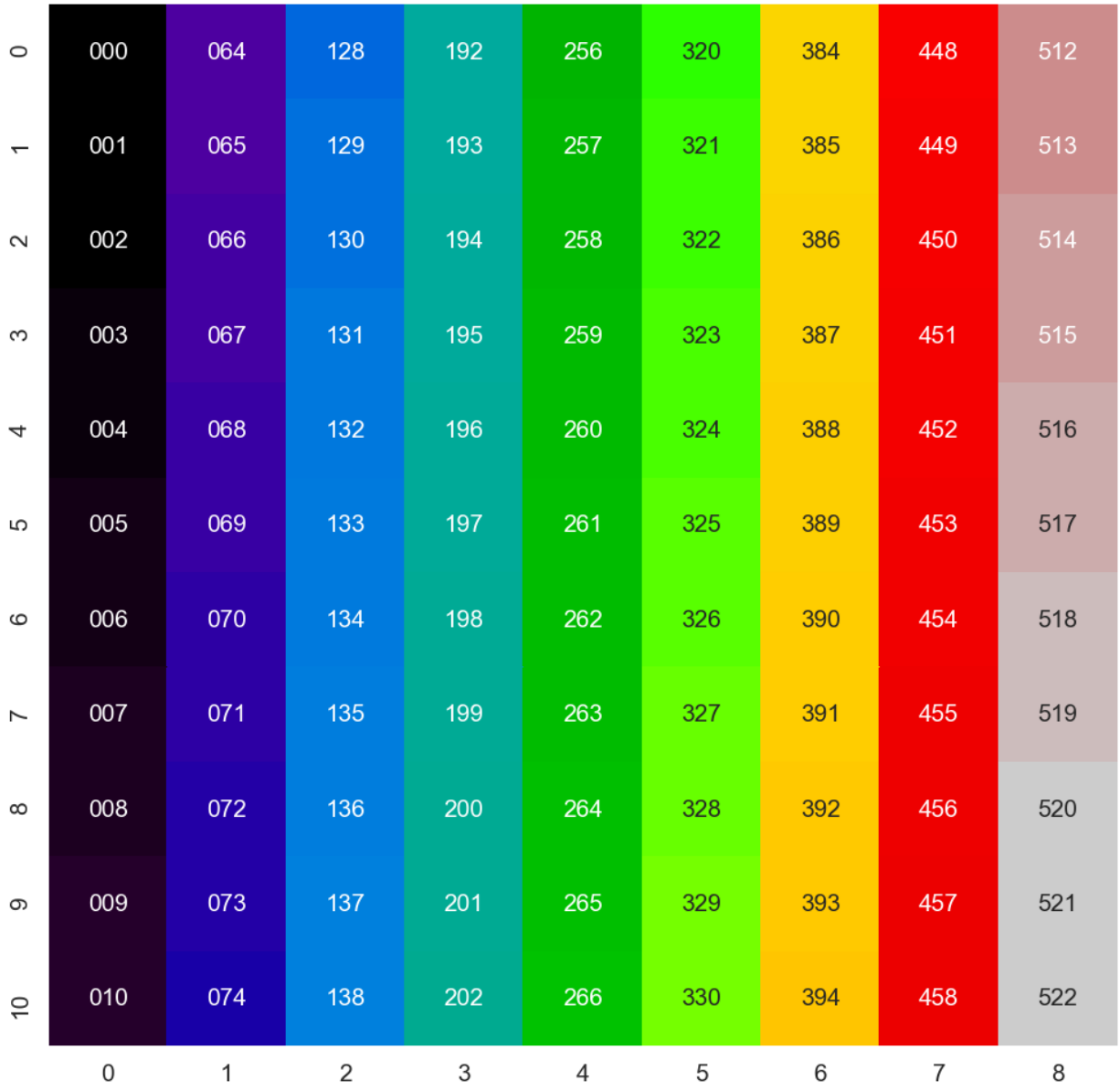
CHAPTER

TWENTYNINE

TILING

Here we divide the image up into unique tiles for further processing

```
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,
            );
```


0 000 064 128 192 256 320 384 448 512
1 001 065 129 193 257 321 385 449 513
2 002 066 130 194 258 322 386 450 514
3 003 067 131 195 259 323 387 451 515
4 004 068 132 196 260 324 388 452 516
5 005 069 133 197 261 325 389 453 517
6 006 070 134 198 262 326 390 454 518
7 007 071 135 199 263 327 391 455 519
8 008 072 136 200 264 328 392 456 520
9 009 073 137 201 265 329 393 457 521
10 010 074 138 202 266 330 394 458 522
0 1 2 3 4 5 6 7 8

CHAPTER

THIRTY

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 GLCM properties each patch
from skimage.feature import greycomatrix, greycoprops
from tqdm.notebook import tqdm
grayco_prop_list = ['contrast', 'dissimilarity',
                    'homogeneity', 'energy',
                    '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 tqdm(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]
```

```
HBox(children=(HTML(value=''), FloatProgress(value=0.0, max=99.0), HTML(value='')))
```

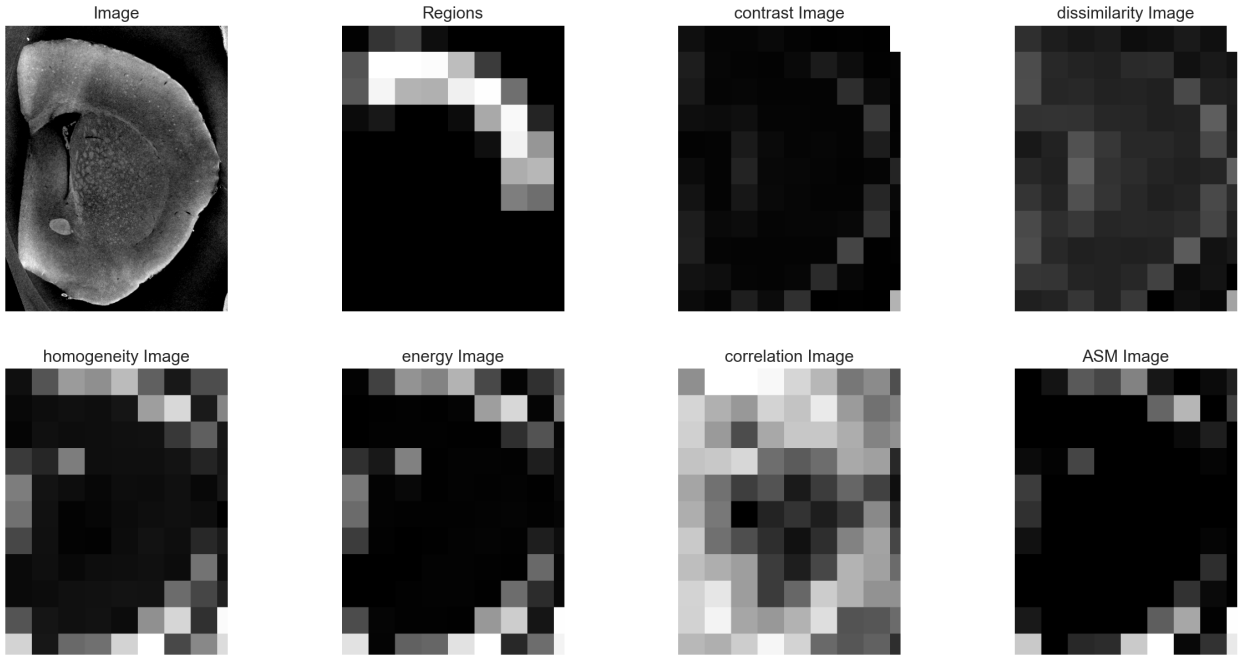
```

```

```
# show the slice and threshold
fig, m_axs = plt.subplots(2, 4, figsize=(20, 10))
```

(continues on next page)

(continued from previous page)

```
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')
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))
```




```
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       99.000000      99.000000  99.000000     99.000000      99.000000
mean        61.214533      28.922878   0.159290    888.349768      15.933864
std         32.747924      20.736974   0.310034   1736.226189      10.016607
min          1.274123       3.241296   0.000000     14.721364       1.666023
25%         39.180681      14.174057   0.000000    249.179078      11.847264
50%         62.908854      20.017975   0.000000    334.321682      13.734549
75%         80.434896      35.653407   0.078170    779.776342      18.028875
max        131.299479      94.529072   1.000000  13619.651391      82.260229

       homogeneity     energy  correlation        ASM
count    99.000000  99.000000    99.000000  99.000000
```

(continues on next page)

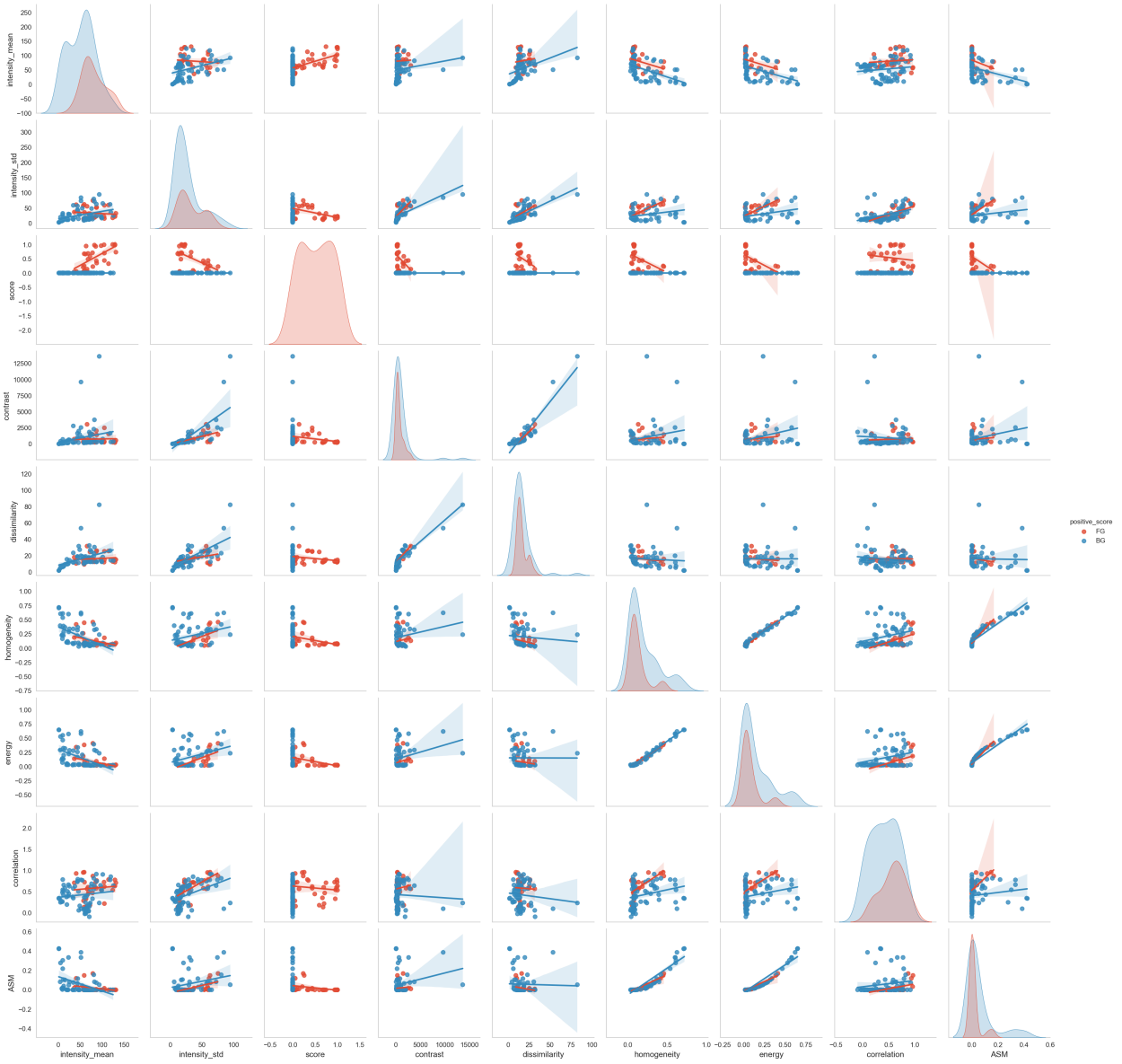
(continued from previous page)

```
mean       0.179556   0.128972     0.468967   0.045523
std        0.173307   0.170833     0.268096   0.096791
min        0.033100   0.019123    -0.094922   0.000366
25%        0.070740   0.025010     0.247538   0.000626
50%        0.082297   0.030073     0.504085   0.000904
75%        0.250455   0.191400     0.677313   0.036697
max        0.718922   0.651809     0.958805   0.424854
```

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

```
/Users/kaestner/opt/anaconda3/lib/python3.8/site-packages/seaborn/distributions.
↪py:305: UserWarning:

Dataset has 0 variance; skipping density estimate.
```

```
<seaborn.axisgrid.PairGrid at 0x7f9bc6d676d0>
```


intensity_mean
intensity_std
score
contrast
dissimilarity
homogeneity
energy
correlation
ASM
positive_score
FG
BG

CHAPTER

THIRTYONE

SUMMARY

31.1 Distance maps

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

31.2 Visualizing 3D data

- Meshes
- Surfaces
- Tools in python

31.3 Surface description

- Curvatures

31.4 Texture

- Textures are patterns in images
- Measure texture characteristics