Quantitative Big Imaging - Complex shapes

Anders Kaestner

Apr 02, 2025

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.

0.1 Complex Objects and Distributions

0.1.1 Literature / Useful References

Books

- Jean Claude, Morphometry with R
- John C. Russ, The Image Processing Handbook,(Boca Raton, CRC Press)
- Pierre Soille,

Papers / Sites

Voronoi Tesselations

- Ghosh, S. (1997). Tessellation-based computational methods for the characterization and analysis of heterogeneous microstructures. Composites Science and Technology, 57(9-10), 1187–1210
- Wolfram Explanation

Self-Avoiding / Nearest Neighbor

- Schwarz, H., & Exner, H. E. (1983). The characterization of the arrangement of feature centroids in planes and volumes. Journal of Microscopy, 129(2), 155–169.
- Kubitscheck, U. et al. (1996). Single nuclear pores visualized by confocal microscopy and image processing. Biophysical Journal, 70(5), 2067–77.

Alignment / Distribution Tensor

- Mader, K. et al (2013). A quantitative framework for the 3D characterization of the osteocyte lacunar system. Bone, 57(1), 142–154
- Aubouy, M., et al. (2003). A texture tensor to quantify deformations. Granular Matter, 5, 67–70.

Two point correlation

- Dinis, L., et. al. (2007). Analysis of 3D solids using the natural neighbour radial point interpolation method. Computer Methods in Applied Mechanics and Engineering, 196(13-16)

```python
import seaborn as sns
from skimage.morphology import skeletonize
from skimage.morphology import skeletonize_3d
from skimage.morphology import binary_opening, binary_closing, disk
from skimage.morphology import grayreconstruct as gr
from scipy.ndimage       import distance_transform_edt
import numpy as np
from skimage.color       import hsv2rgb, rgb2hsv
from skimage.morphology import medial_axis
from skimage.morphology import skeletonize, skeletonize_3d
from skimage.filters     import laplace
from skimage.segmentation import mark_boundaries
import seaborn as sns
```

(continues on next page)

(continued from previous page)

```
from skimage.morphology import label
from scipy.ndimage import convolve
from matplotlib.patches import Rectangle
from matplotlib.patches import ConnectionPatch

from skimage.morphology import opening, closing, disk  # for removing small objects
import matplotlib.pyplot as plt  # for showing plots
from skimage.io   import imread    # for reading images
import pandas as pd               # for reading the swc files (tables of somesort)
import sys
sys.path.append('../common/')
import plotsupport as ps

import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

%matplotlib inline

from matplotlib.colors import ListedColormap
```

0.1.2 Outline

- Motivation (Why and How?)
- Skeletons
- Tortuosity
- Watershed Segmentation
- Connected Objects

0.1.3 Previously on QBI …

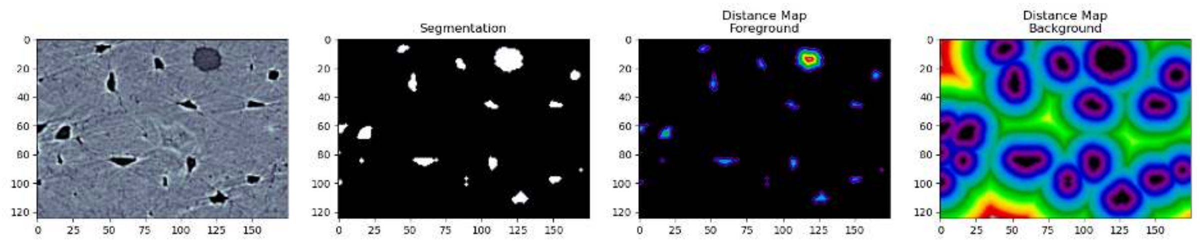
- Image Enhancement
- Understanding image histograms
- Automatic Methods
- Component Labeling
- Single Shape Analysis
- Complicated Shapes (Thickness Maps)

0.1.4 Learning Objectives

Motivation (Why and How?)

- How can we extract topology of a structure?
- How do we identify seperate objects when they are connected?
- How can we compare shape of complex objects when they grow?

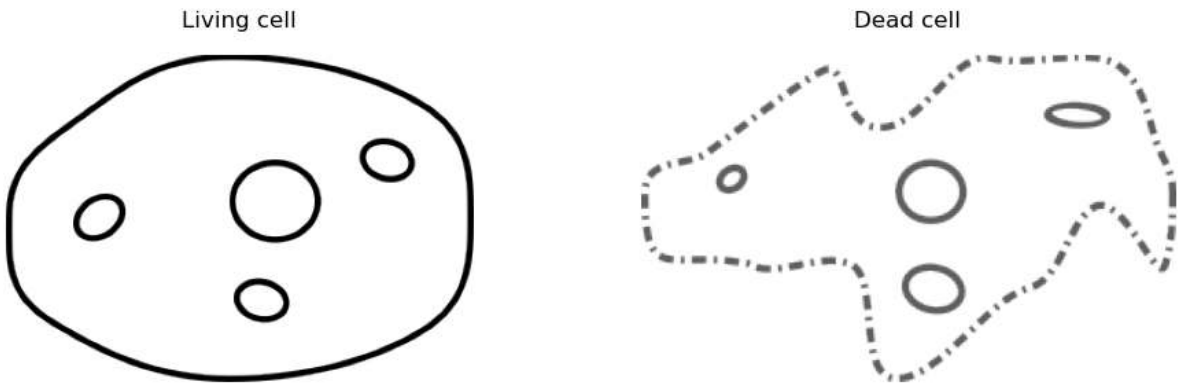
```
bw_img = imread("../Lecture-05/figures/bonegfiltslice.png")[::2, ::2]
thresh_img = binary_closing(binary_opening(bw_img < 90, disk(1)), disk(2))
fg_dmap = distance_transform_edt(thresh_img)
bg_dmap = distance_transform_edt(1-thresh_img)
fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(20, 6), dpi=100)
ax1.imshow(bw_img, cmap='bone')
ax2.imshow(thresh_img, cmap='bone');         ax2.set_title('Segmentation');
ax3.imshow(fg_dmap, cmap='nipy_spectral');   ax3.set_title('Distance Map\nForeground')
ax4.imshow(bg_dmap, cmap='nipy_spectral');   ax4.set_title('Distance Map\nBackground');
```



0.1.5 Distribution Objectives - finding the right questions

We want to know how many cells are alive

```
fig,axs = plt.subplots(1,2,figsize=(10,3))
imgs=['figures/living_cell.png',"figures/dead_cell.png"]
titles=['Living cell','Dead cell']
for ax,img,title in zip(axs,imgs,titles) :
    ax.imshow(plt.imread(img))
    ax.set(xticks=[],yticks=[])
    ax.set_title(title,y=1.05)
    ax.axis('off')

plt.tight_layout()
```

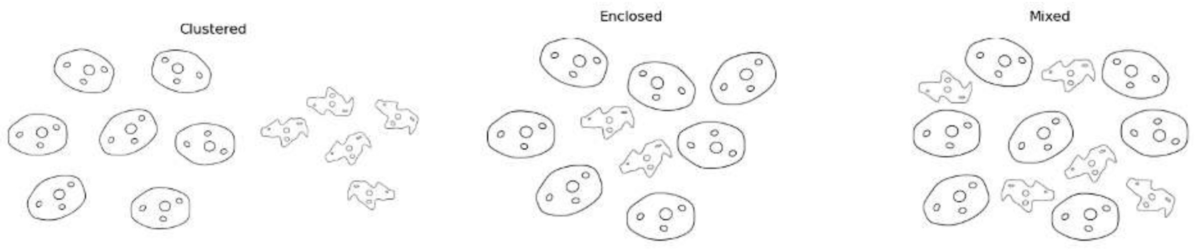


Maybe…

- small cells are dead and larger cells are alive → examine the volume distribution
- living cells are round and dead cells are really spiky and pointy → examine anisotropy

Different distributions

```
fig,axs = plt.subplots(1,3,figsize=(15,3))
imgs=['figures/cells1.png',"figures/cells2.png",'figures/cells3.png']
titles=['Clustered','Enclosed','Mixed']
for ax,img,title in zip(axs,imgs,titles) :
    ax.imshow(plt.imread(img))
    ax.set(xticks=[],yticks=[])
    ax.set_title(title,y=1.05)
    ax.axis('off')

plt.tight_layout()
```



We want to know where the cells are alive or most densely packed

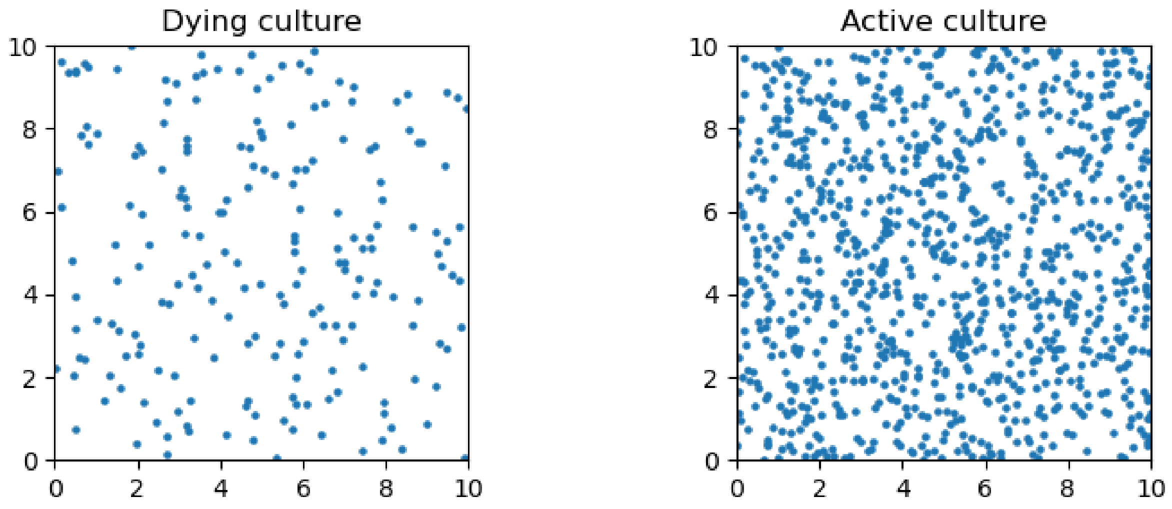
- We can visually inspect the sample (maybe even color by volume)
- We can examine the raw positions (x,y,z) but what does that really tell us?
- We can make boxes and count the cells inside each one
- How do we compare two regions in the same sample or even two samples?

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

def point_field(intensity=10,width=10,height=10,ax=None,title = None) :
```

(continues on next page)

(continued from previous page)

```
    # Total expected number of points
    area = width * height
    num_points = np.random.poisson(intensity * area)

    # Generate uniform random points
    x = np.random.uniform(0, width, num_points)
    y = np.random.uniform(0, height, num_points)

    # Plot
    if ax is None:
        fig,ax=plt.subplots(1,figsize=(6, 6))

    ax.scatter(x, y, s=5)
    ax.set_xlim(0, width)
    ax.set_ylim(0, height)
    if title is None :
        ax.set_title(f"2D Poisson Point Field ({num_points} points)")
    else :
        ax.set_title(title)

    ax.set_aspect('equal')


fig,ax = plt.subplots(1,2,figsize=(9,3))
point_field(intensity=2,ax=ax[0],title='Dying culture')
point_field(intensity=10,ax=ax[1], title='Active culture')
```



0.1.6 So what do we still need

1. A way for counting cells in a region and estimating density without creating arbitrary boxes
2. A way for finding out how many cells are *near* a given cell, it's nearest neighbors
3. A way for quantifying how far apart cells are and then comparing different regions within a sample
4. A way for quantifying and comparing orientations

What would be really great?

A tool which could be adapted to answering a large variety of problems

- multiple types of structures
- multiple phases

0.1.7 Metrics

We examined a number of different metrics in this lecture and additionally to classifying them as **Local** and **Global** we can define them as point and voxel-based operations.

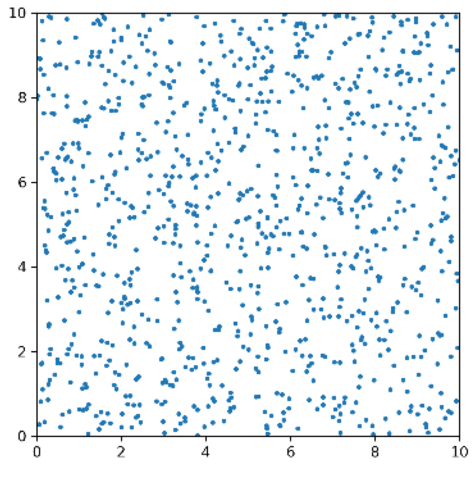
Point Operations

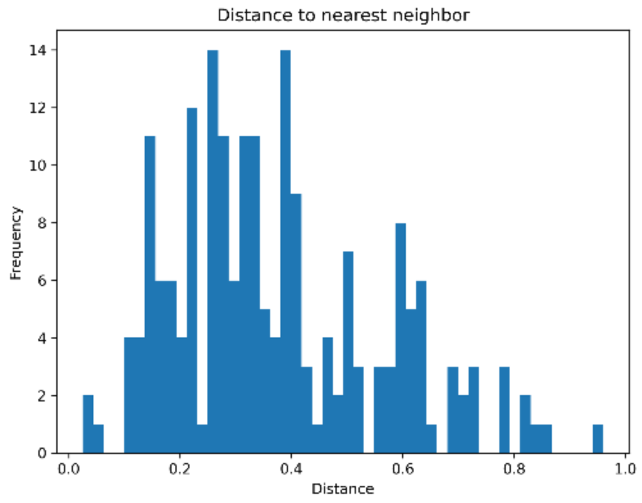
- Nearest Neighbor
- Point (Center of Volume)-based Voronoi Tesselation
- Alignment

Voxel Operation

- Voronoi Tesselation
- Neighbor Counting
- 2-point (N-point) correlation function

Nearest neighbor distribution

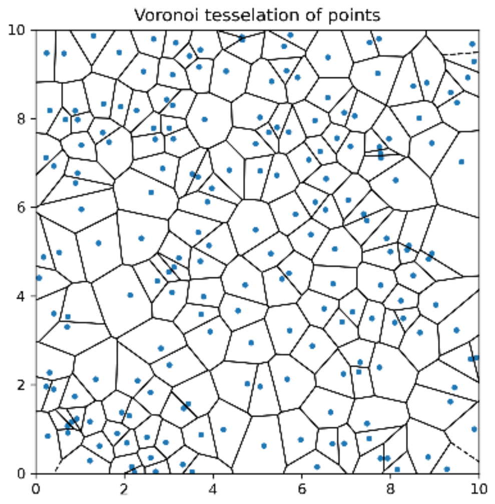


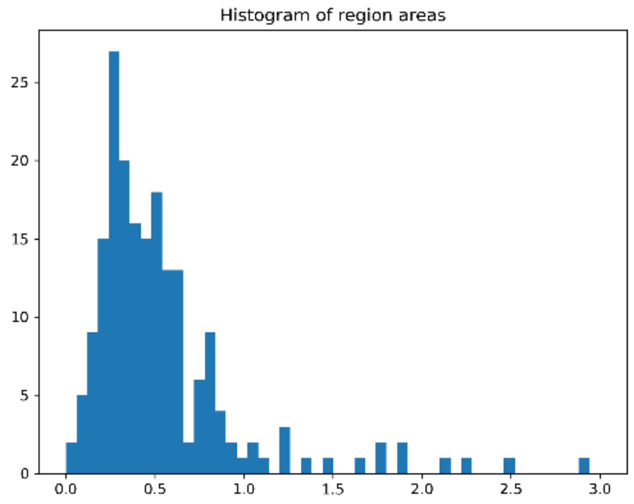


Voronoi tesselation

Voronoi tesselation is a method to compute the support area of regions in a point cloud. The support area is defined as the region where the distance to the dot is closer than any other dot in the cloud. This can be done either using the coordinated of the items or from images with dots. The distance map of the background can be used to compute the Voronoi tesselation.

The distribution of the region areas can be used to characterize the structure of the point cloud. Like in the histogram to the right.





0.1.8 Destructive Measurements

With most imaging techniques and sample types, the task of measurement itself impacts the sample.

- X-ray tomography which *claim* to be non-destructive still impart significant to lethal doses of X-ray radition for high resolution imaging
- Electron microscopy, auto-tome-based methods, histology are all markedly more destructive and make longitudinal studies impossible

Even when such measurements are possible registration can be a difficult task and introduce artifacts

Why is this important?

- techniques which allow us to compare different samples of the same type.
- are sensitive to common transformations
- Sample B after the treatment looks like Sample A stretched to be 2x larger
- The volume fraction at the center is higher than the edges but organization remains the same

0.2 Structure analysis

One main objective in scientific imaging is to describe and quantify shapes.

We have alread looked into several metrics to describe items in the image:

- Area
- Perimeter
- Orientation
- Position
- etc.

Today, we will look into two further techniques used for structure analysis:

- Skeletons
- Advanced object labeling

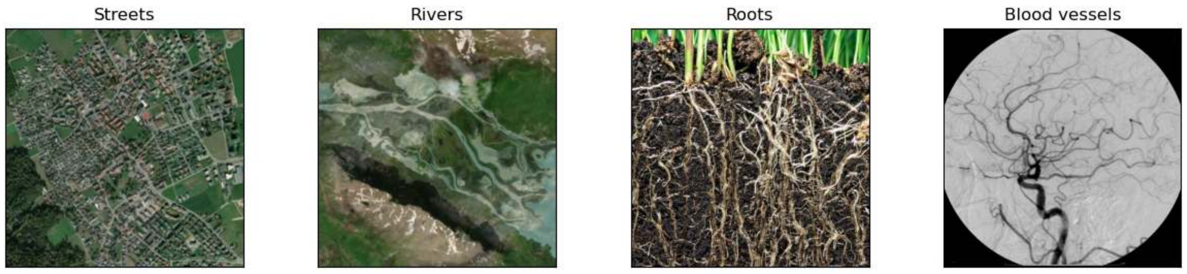
0.3 Skeletonization / Networks

Thin structures and networks often appears in images.

0.3.1 Thin structures to analyze

The world is full of thin structures we want to analyze. Below you see some examples.

```python
fig,axs = plt.subplots(1,4,figsize=(15,3))
imgs=['figures/map_roads.png',"figures/riverdelta.png",'figures/roots_picture.jpg',
 ↪'figures/Cerebral_Angiogram_Lateral.jpg']
titles=['Streets','Rivers','Roots','Blood vessels']
for ax,img,title in zip(axs,imgs,titles) :
    ax.imshow(plt.imread(img))
    ax.set(title=title,xticks=[],yticks=[])
```



0.3.2 What we want to know about networks

In many cases we want to describe the topology of the network

- which structures are connected
- how they are connected
  - Are there loops
- express the network in a simple manner
- quantify tortuosity
- branching

We start with a simpler example from the EPFL Dataset: EPFL CVLab's Library of Tree-Reconstruction Examples (http://cvlab.epfl.ch/data/delin)

0.3.3 To describe network topology we need

The minimal structure that spans the structure topology - i.e. a skeleton

0.4 Network analysis explained in real images

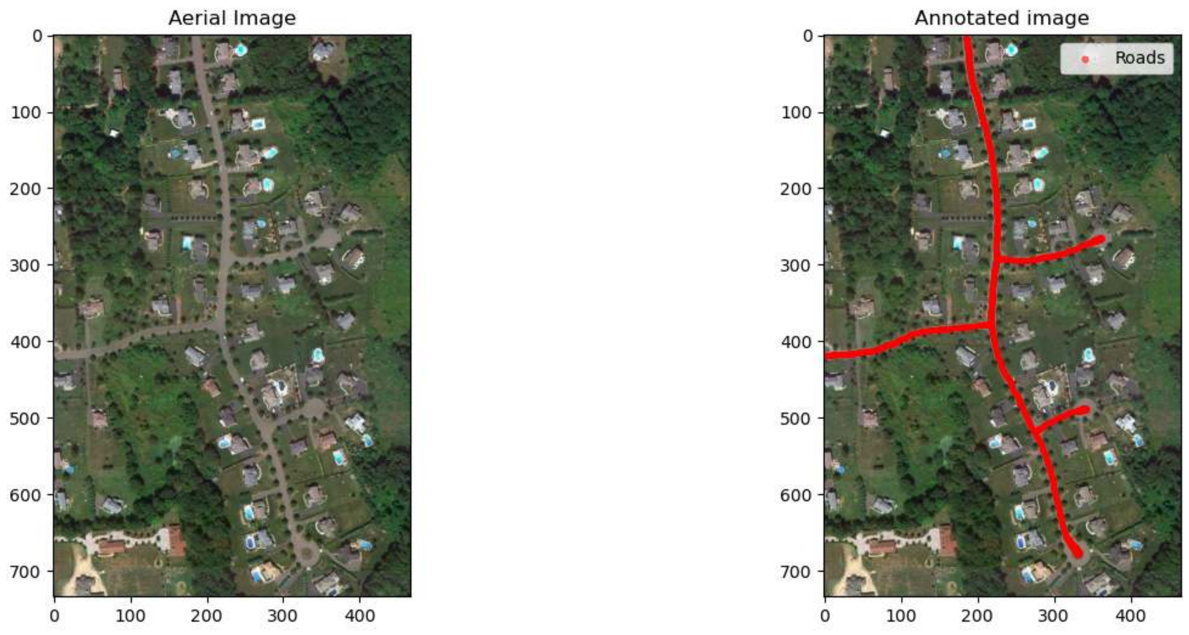
We will use an aerial picture of a street as test image for exploring how to build skeleton and to improve the performance of the skeletonization. We also have markup data as ground truth for the position of the street.

```python
def read_swc(in_path):
    swc_df = pd.read_csv(in_path, sep=' ', comment='#', header=None)
    # a pure guess here
    swc_df.columns = ['id', 'junk1', 'x', 'y', 'junk2', 'width', 'next_idx']
```

(continues on next page)

(continued from previous page)

```
    return swc_df[['x', 'y', 'width']]

im_data = imread('figures/ny_7.tif')
mk_data = read_swc('figures/ny_7.swc')

fig, (ax1, ax3) = plt.subplots(1, 2, figsize=(15, 6))
ax1.imshow(im_data);ax1.set_title('Aerial Image')
ax3.imshow(im_data, cmap='bone') ;ax3.scatter(mk_data['x'], mk_data['y'], s=mk_data[
↪'width'], alpha=0.5,color='red',label="Roads") ;ax3.set_title('Annotated image'),␣
↪ax3.legend();
```



0.4.1 A close-up of the street

The full street image is too large to see all details. Therefore, we crop a piece in middle of the picture.

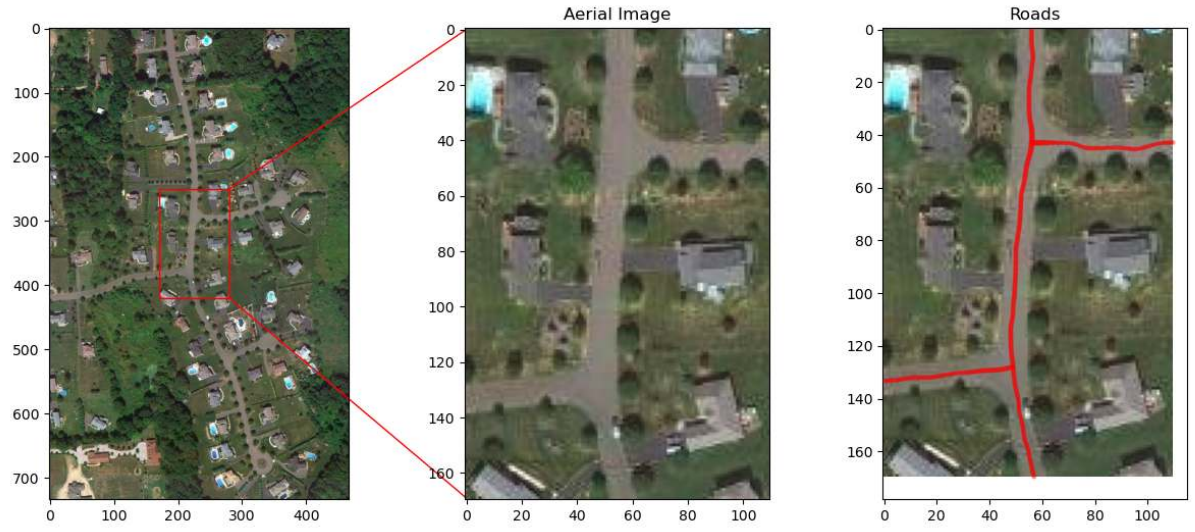
```
# Crop image
im_crop = im_data[250:420:1, 170:280:1]
# Select street point in cropped region
mk_crop = mk_data.query('y>=250').query('y<420').query('x>170').query('x<=280').copy()
# Adjust position. bias
mk_crop.x = (mk_crop.x-170)
mk_crop.y = (mk_crop.y-250)
```

```
fig, (ax1,ax2, ax3) = plt.subplots(1, 3, figsize=(15, 6))
ax1.imshow(im_data)
rect=Rectangle((170,250),height=170,width=110,color='r',fc='none')
ax1.add_patch(rect)
ax2.imshow(im_crop)
ax2.set_title('Aerial Image')
con2 = ConnectionPatch(xyA=(280,250), xyB=(0,0), coordsA="data", coordsB="data",
                      axesA=ax1, axesB=ax2, color="red", lw=1)
```

(continues on next page)

(continued from previous page)

```
ax1.add_artist(con2)
con1 = ConnectionPatch(xyA=(280,420), xyB=(0,169), coordsA="data", coordsB="data",
                       axesA=ax1, axesB=ax2, color="red", lw=1)
ax1.add_artist(con1);

ax3.imshow(im_crop, cmap='bone')
ax3.scatter(mk_crop['x'], mk_crop['y'], s=mk_crop['width'],color='red', alpha=0.25)
ax3.set_title('Roads');
```



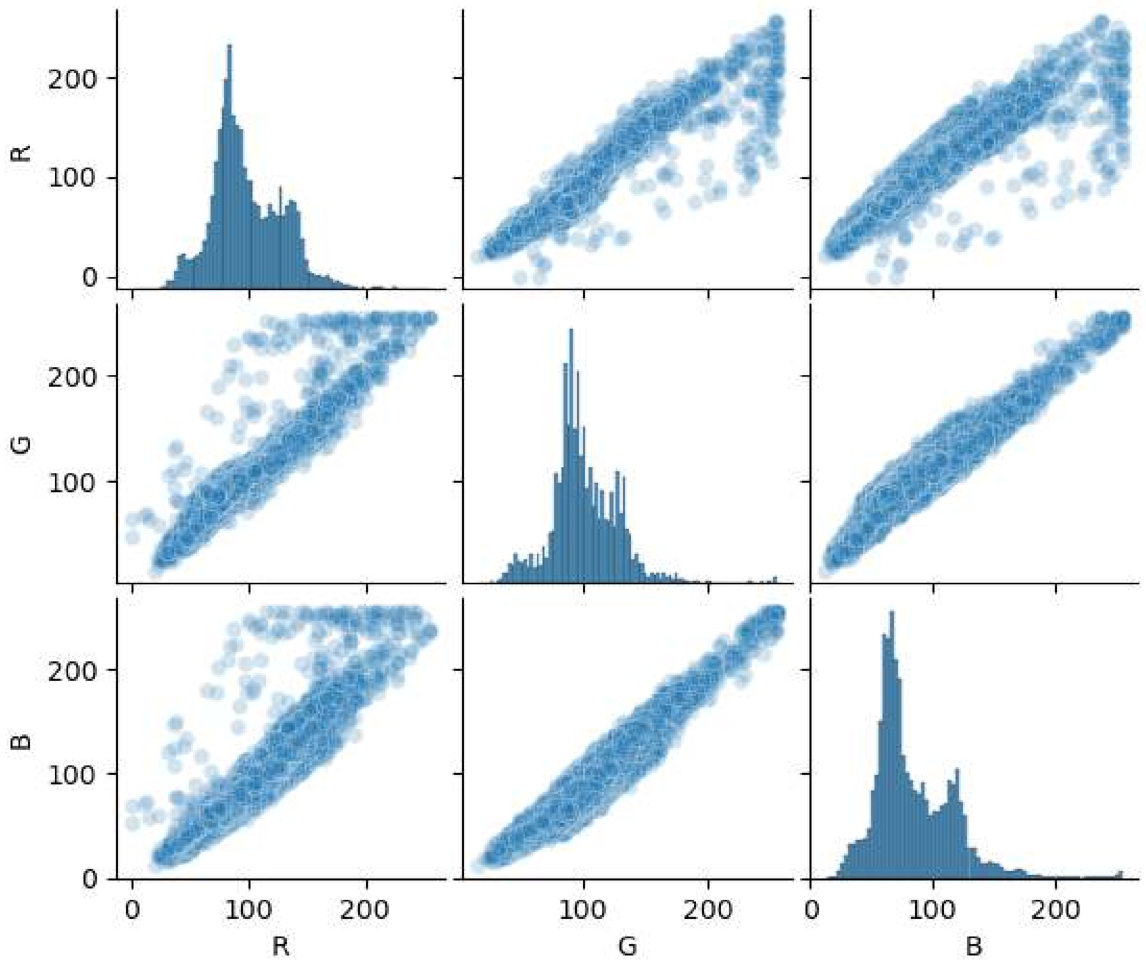
0.4.2 Let's try finding the roads

The first thing we have to for our street analysis example is to identify the street among all other features in the picture.

Step 1: Inspecting the data

Let's first get acquainted with the data using a pair plot for the color chanels RGB in the image

```
from sklearn.cluster import KMeans
from seaborn import pairplot
import pandas as pd
img = im_crop
lst = {'R': img[:,:,0].ravel(),'G': img[:,:,1].ravel(),'B': img[:,:,2].ravel()}
dfRGB = pd.DataFrame(lst)

pp=pairplot(dfRGB,plot_kws={'alpha': 0.2}); pp.fig.set_size_inches(6, 5)
```



Step 2: Segmentation

Thresholding

The picture is a color image that uses the RGB color model. In this case we will convert the color model to HSV and use the V parameter for the thresholding. An empirical value for the threshold is v>0.4. This threshold results in many structure besides the street.

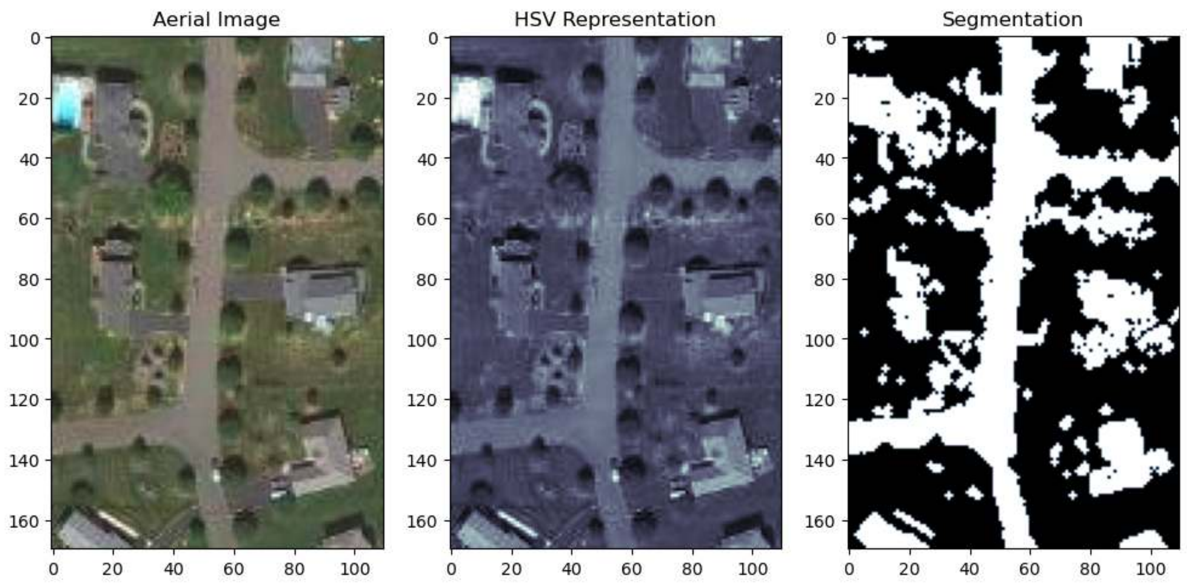
```
def thresh_image(in_img):
    v_img = rgb2hsv(in_img)[:, :, 2]
    th_img = v_img > 0.4
    op_img = opening(th_img, disk(1))
    return op_img

seg_img = thresh_image(im_crop)
```

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 6))
ax1.imshow(im_crop); ax1.set_title('Aerial Image')
ax2.imshow(rgb2hsv(im_crop)[:, :, 2], cmap='bone')
ax2.set_title('HSV Representation')
```

(continues on next page)

(continued from previous page)

```
ax3.imshow(seg_img, cmap='bone')
ax3.set_title('Segmentation');
```

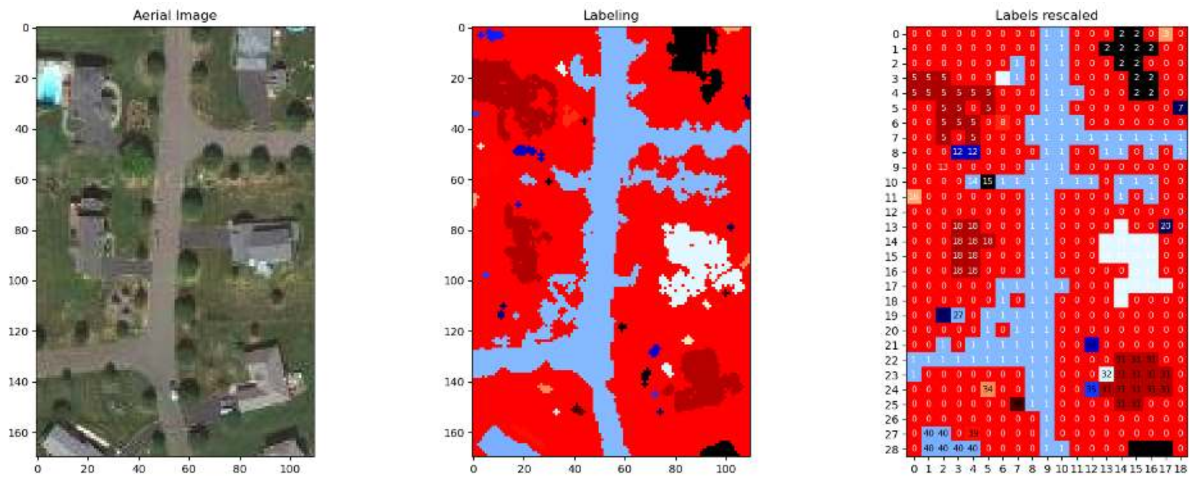


Step 3: Identify structures

Using connected component labeling (lecture 6)

```
lab_img0 = label(seg_img)
```

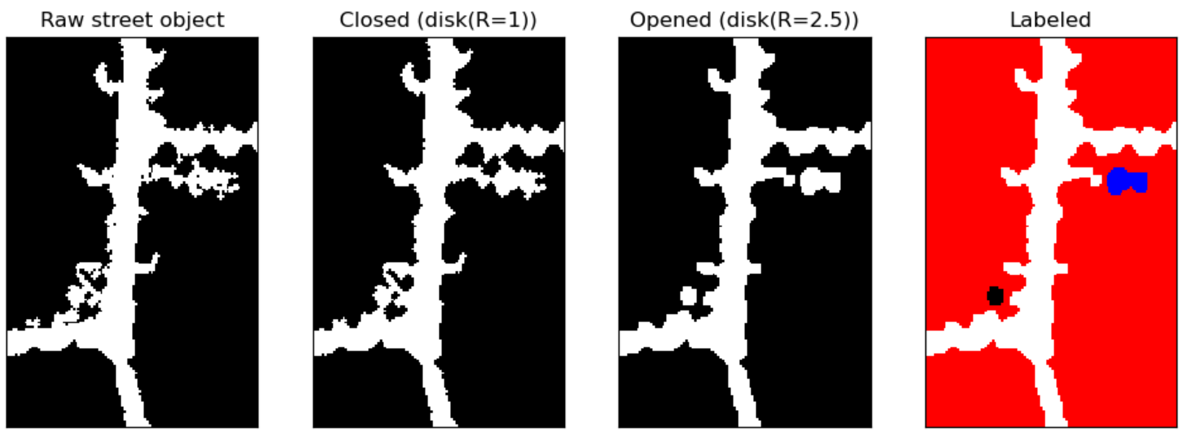
```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 7))
ax1.imshow(im_crop)
ax1.set_title('Aerial Image')
ax2.imshow(lab_img0, cmap='flag')
ax2.set_title('Labeling')
ps.heatmap(lab_img0[::6, ::6],cmap='flag',ax=ax3,bgbox=False,precision=0,fontsize=8)
 ↪# Show every 6th pixel)

ax3.set_title('Labels rescaled');
```



Step 4: Cleaning up

```python
from skimage.morphology import opening, closing,disk

street = (lab_img0==1)
clean1 = closing(street,footprint=disk(1))
clean2 = opening(clean1,footprint=disk(2.5))
# clean1 =  opening(street,footprint=disk(3))
# clean2 = closing(clean1,footprint=disk(1))
lab_img = label(clean2)
```

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

ax[0].imshow(street,cmap='gray',interpolation='none')
ax[0].set(xticks=[],yticks=[],title='Raw street object')
ax[1].imshow(clean1,cmap='gray',interpolation='none')
ax[1].set(xticks=[],yticks=[],title='Closed (disk(R=1))')
ax[2].imshow(clean2,cmap='gray',interpolation='none')
ax[2].set(xticks=[],yticks=[],title='Opened (disk(R=2.5))')
ax[3].imshow(lab_img,cmap='flag',interpolation='none')
ax[3].set(xticks=[],yticks=[],title='Labeled');
```



0.5 Skeletonization - take one

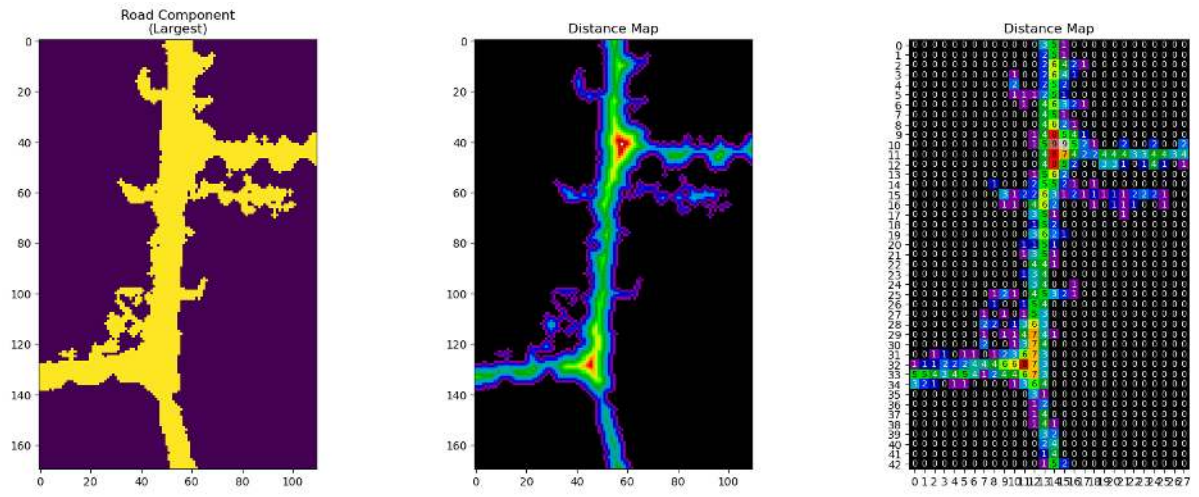
The first step is to take the distance transform the structure

$$I_d(x,y) = \text{dist}(I(x,y))$$

We can see in this image there are already local maxima that form a sort of backbone which closely maps to what we are interested in.

```python
from scipy import ndimage
keep_lab_img = (lab_img0 == 1) # Create an image with pixels belonging to sement i
dist_map     = ndimage.distance_transform_edt(keep_lab_img.astype(float))
```

```python
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 7))
ax1.imshow(keep_lab_img,interpolation='none')
ax1.set_title('Road Component\n(Largest)')

ax2.imshow(dist_map, cmap='nipy_spectral')
ax2.set_title('Distance Map')

ps.heatmap(dist_map[::4, ::4],cmap='nipy_spectral',ax=ax3,bgbox=False,precision=0,
 ↪fontsize=8) # Show every 6th pixel)
ax3.set_title('Distance Map');
```



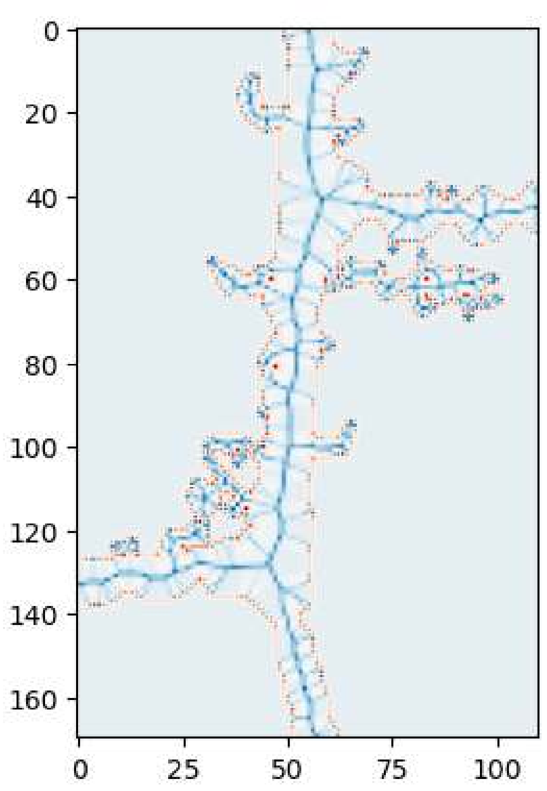
0.5.1 Skeletonization: Ridges

By using the Laplacian filter as an approximate for the derivative operator which finds the values which high local gradients.

$$\nabla I_d(x,y) = \left(\frac{\delta^2}{\delta x^2} + \frac{\delta^2}{\delta y^2}\right) I_d \approx \underbrace{\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}}_{\text{Laplacian Kernel}} * I_d(x,y)$$

```python
lapskel = laplace(dist_map)
```

```
plt.imshow(lapskel, cmap='RdBu');
ax2.set_title('Laplacian of Distance');
```



0.5.2 Creating the skeleton

We can locate the local maxima of the structure by setting a minimum surface distance

$$I_d(x,y) > MIN_{DIST}$$

and combining it with a minimum slope value

$$\nabla I_d(x,y) > MIN_{SLOPE}$$

Thresholds on the distance map

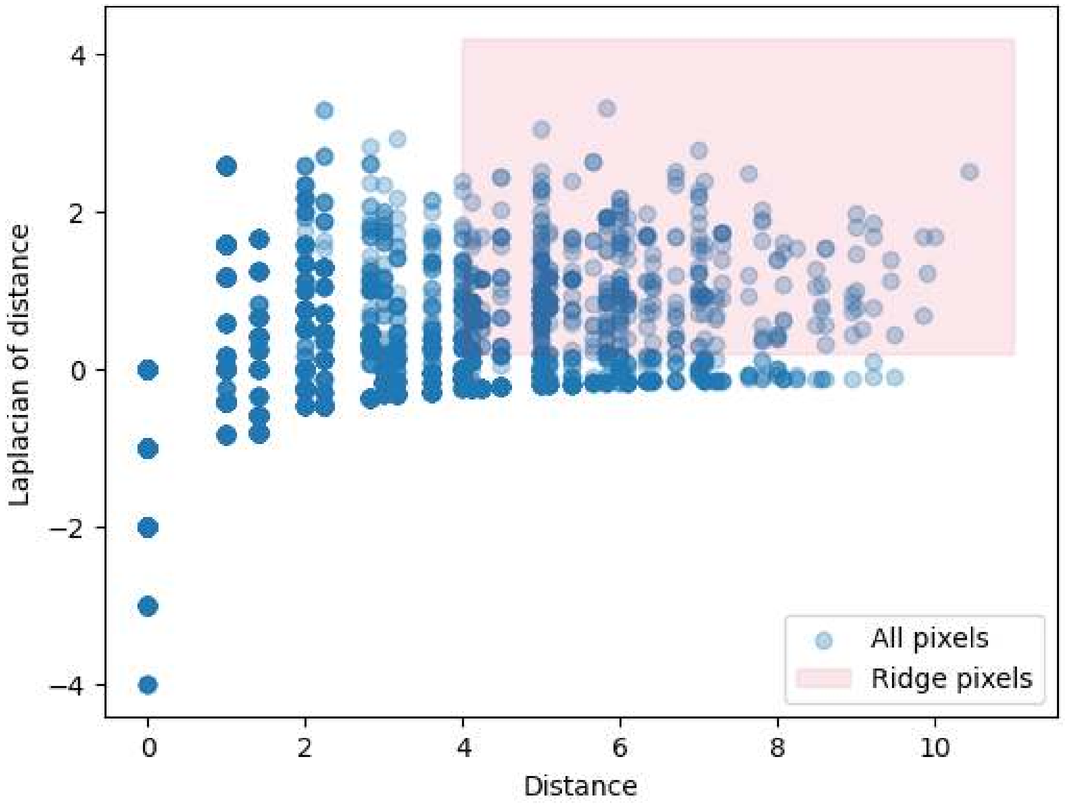
Harking back to our earlier lectures, this can be seen as a threshold on a feature vector representation of the entire dataset.

- We first make the dataset into a tuple

$$\text{cImg}(x,y) = \langle \underbrace{I_d(x,y)}_{1}, \underbrace{\nabla I_d(x,y)}_{2} \rangle$$

$$\text{skelImage}(x,y) = \begin{cases} 1, & \text{cImg}_1(x,y) \geq MIN_{DIST} \text{ and } \text{cImg}_2(x,y) \geq MIN_{SLOPE} \\ 0, & \text{otherwise} \end{cases}$$

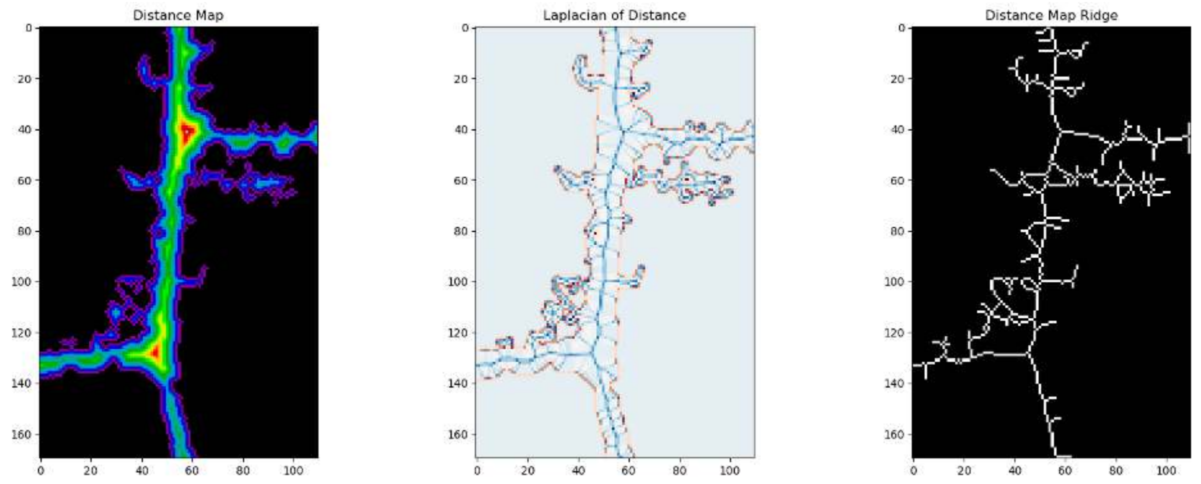
```
d = {'Distance' : dist_map.ravel(), 'Laplacian': lapskel.ravel()}
df = pd.DataFrame(d,)
#sns.pairplot(df)
fig, ax = plt.subplots()
ax.scatter(dist_map.ravel(),lapskel.ravel(),alpha=0.3,label='All pixels')
ax.add_patch(Rectangle((4,0.2),7,4,color='crimson',alpha=0.1,label='Ridge pixels'))
ax.set_xlabel('Distance')
ax.set_ylabel('Laplacian of distance')
ax.legend(loc='lower right');
```



Resulting ridge skeleton

```
lapskel = laplace(dist_map)
skel    = medial_axis(keep_lab_img, return_distance=False) ; # we use medial axis␣
↪since it is cleaner
```

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 7))

ax1.imshow(dist_map, cmap='nipy_spectral'); ax1.set_title('Distance Map')
ax2.imshow(lapskel, cmap='RdBu'); ax2.set_title('Laplacian of Distance')


ax3.imshow(skel, cmap='gray'); ax3.set_title('Distance Map Ridge');
```



0.6 Skeletonization with Morphological thinning

From scikit-image documentation (http://scikit-image.org/docs/dev/auto_examples/edges/plot_skeleton.html)

Morphological thinning, implemented in the `thin` function, works on the same principle as `skeletonize`:

- remove pixels from the borders at each iteration until none can be removed without altering the connectivity.
- The different rules of removal can speed up skeletonization and result in different final skeletons.

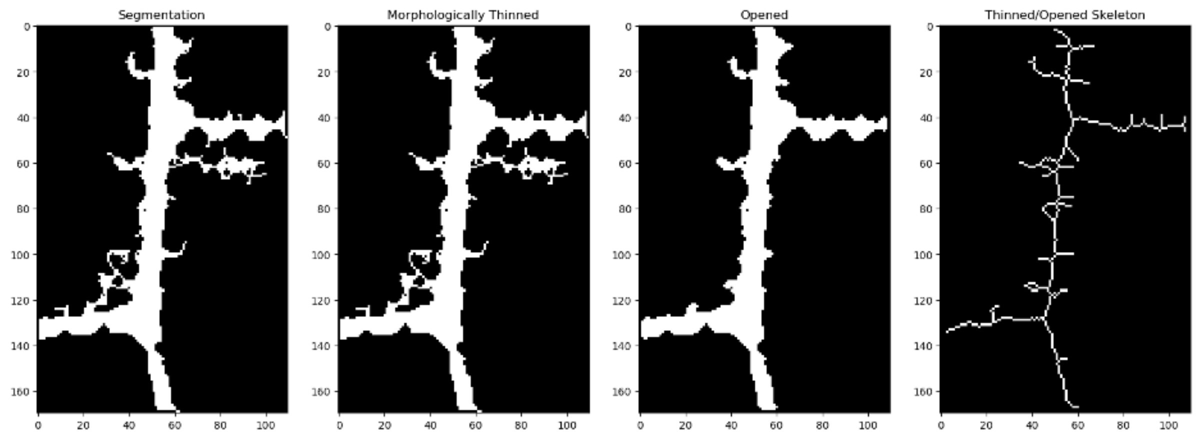
The `thin` function also takes an optional `max_iter` keyword argument to limit the number of thinning iterations, and thus produce a relatively thicker skeleton.

We can use this to thin the tiny junk elements first then erode, then perform the full skeletonization

0.6.1 Try morphological thinning

```
from skimage.morphology import thin, erosion
thin_image       = thin(keep_lab_img, max_num_iter=1)
er_thin_image    = opening(thin_image, disk(1))
er_thin_image    = label(er_thin_image) == 1
opened_skeleton = medial_axis(er_thin_image, return_distance=False)
```

```
fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(20, 7))
ax1.imshow(keep_lab_img, cmap="gray"); ax1.set_title('Segmentation')
ax2.imshow(thin_image, cmap="gray"); ax2.set_title('Morphologically Thinned')
ax3.imshow(er_thin_image, cmap="gray"); ax3.set_title('Opened')
ax4.imshow(opened_skeleton, cmap="gray"); ax4.set_title('Thinned/Opened Skeleton');
```



Still overgrown

The skeleton is still problematic for us and so we require some additional improvements to get a perfect skeleton.

There are a lot spurious branches and even loops on the thinned skeleton. These need to be pruned to get a skeleton that correspond to our expectations.

0.7 Skeleton: Junction Overview

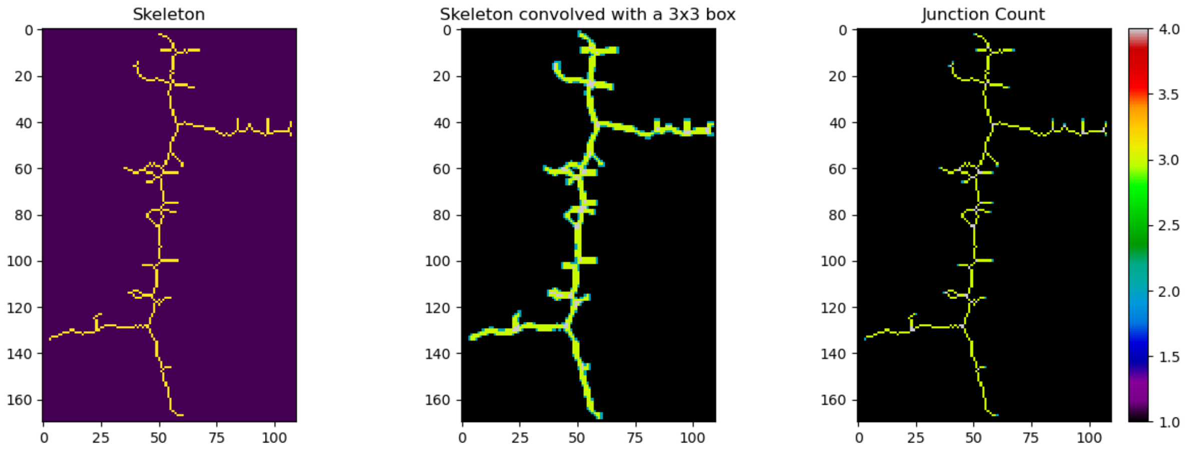
With the skeleton which is ideally one voxel thick, we can characterize the junctions in the system by looking at the neighborhood of each voxel.

Junctions are the pixels were more than two branches intersect. In 2D, there can be at most four branches meeting up at a single pixel. There can however be clusters of junctions that correspond to more complex junction topology.

Here, we will use the convolution with a 3x3 box kernel to identify the junctions. It essentially sums all pixels in the neighborhood. The convolution widens the skeleton by one pixel in all directions. This wider skeleton can be masked with the original skeleton.

```
neighbor_conv = convolve(opened_skeleton.astype(int), np.ones((3, 3)))
# Masking skeleton
neighbor_conv_masked = neighbor_conv.copy()
neighbor_conv_masked[~opened_skeleton] = 0
```

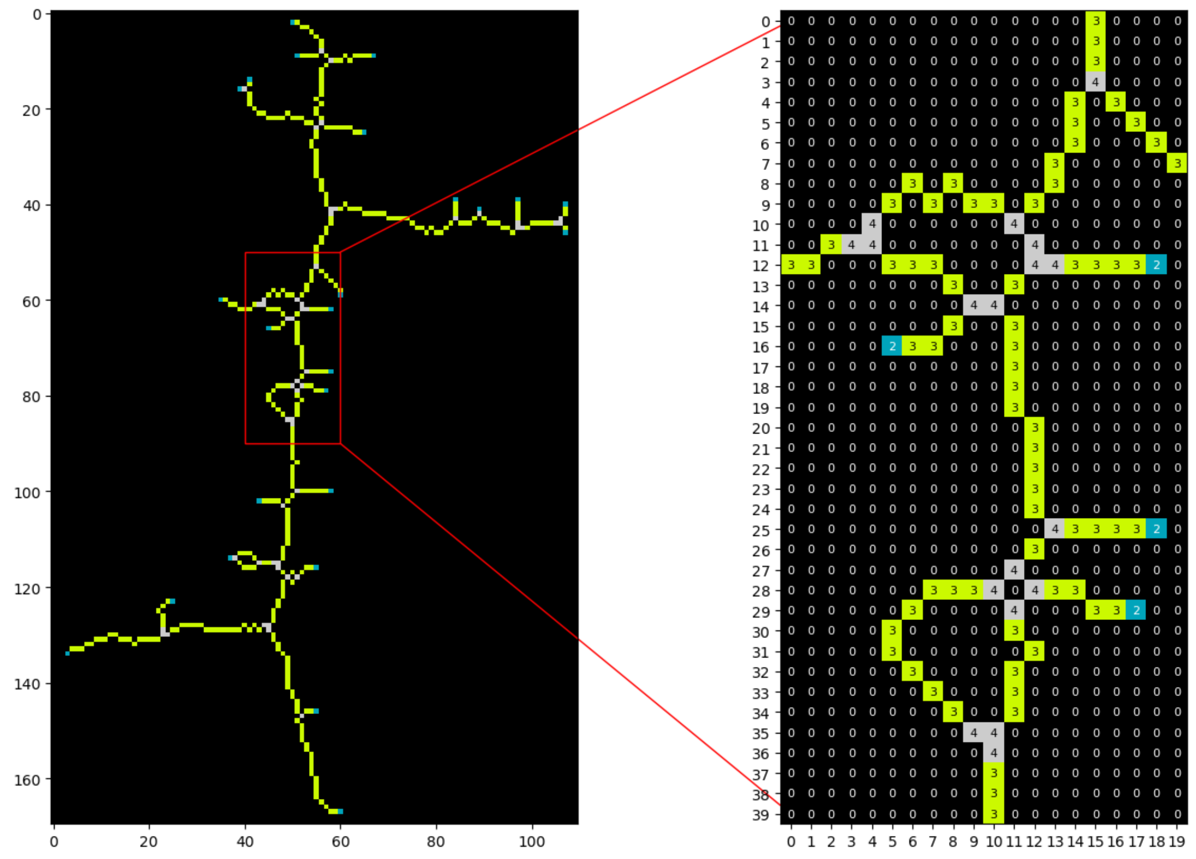
```
fig, ( ax1, ax3,ax2) = plt.subplots(1, 3, figsize=(15, 5))
ax1.imshow(opened_skeleton,interpolation='none')
ax1.set_title('Skeleton')
ax3.imshow(neighbor_conv, cmap='nipy_spectral', vmin=1, vmax=4, interpolation='none' )
ax3.set_title('Skeleton convolved with a 3x3 box')
j_img = ax2.imshow(neighbor_conv_masked, cmap='nipy_spectral', vmin=1, vmax=4,␣
↪interpolation='none')
plt.colorbar(j_img)
ax2.set_title('Junction Count');
```



0.7.1 Close-up on the skeleton

The skeleton pixels of the filteres and masked skeleton has different values. These values tell how many neighbor pixels each pixel has.

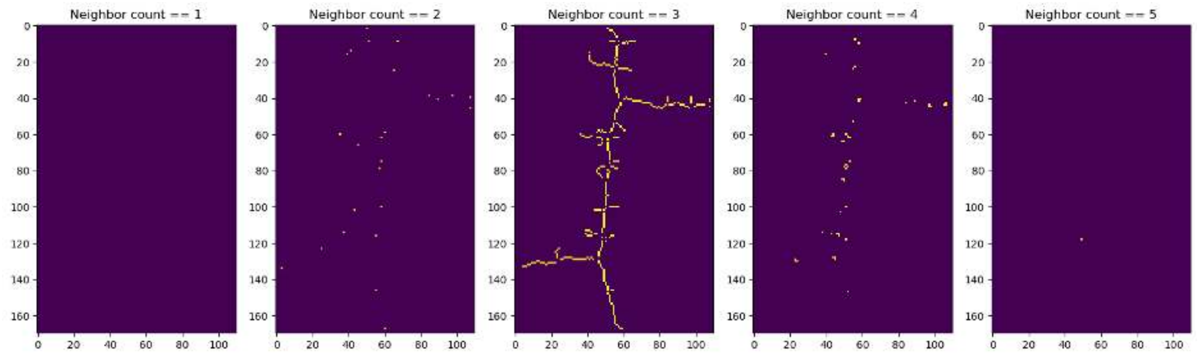
```
from matplotlib.patches import Rectangle
from matplotlib.patches import ConnectionPatch
fig, ax = plt.subplots(1, 2, figsize=(15, 10))
ax[0].imshow(neighbor_conv_masked,cmap='nipy_spectral', vmin=1, vmax=4, interpolation=
↪'none' );
rect = Rectangle(xy=(40.0, 50.0), height=40, width=20, linewidth=1, edgecolor='r',␣
↪facecolor='none')
ax[0].add_patch(rect)
n_crop = neighbor_conv_masked[50:90, 40:60]

ps.heatmap(n_crop,cmap='nipy_spectral',ax=ax[1],bgbox=False,precision=0,fontsize=8,
↪vmin=1,vmax=n_crop.max()) # Show every 6th pixel)
# sns.heatmap(n_crop, annot=True, fmt="d",cmap='nipy_spectral',
#             ax=ax[1], cbar=True,
#             vmin=1, vmax=n_crop.max(), annot_kws={"size": 10});

con2 = ConnectionPatch(xyA=(60,50), xyB=(0,0), coordsA="data", coordsB="data",
                      axesA=ax[0], axesB=ax[1], color="red", lw=1)
ax[0].add_artist(con2)
con1 = ConnectionPatch(xyA=(60,90), xyB=(0,39), coordsA="data", coordsB="data",
                      axesA=ax[0], axesB=ax[1], color="red", lw=1)
ax[0].add_artist(con1);
```



0.7.2 Skeleton pixel neighborhood classes

Now that we have seen that the convolution gives information about the neigborhood constellation, we can start to look for different characteristic combinations.

```python
junc_types = np.unique(neighbor_conv[neighbor_conv > 0])
fig, m_axs = plt.subplots(1, len(junc_types), figsize=(20, 7))
for i, c_ax in zip(junc_types, m_axs):
    c_ax.imshow(neighbor_conv_masked == i, interpolation='none')
    c_ax.set_title('Neighbor count == {}'.format(i))
```



In the table below we make a rough categorization of the neighbor counts.

Pixel counts	Interpretation
0	Background
1	Isolated point
2	End point
3	Line segment
4	Three-way Junction
5	Four-way junction

0.7.3 Smarter kernel coding

The uniform kernel was able to guid us to right positions on the skeleton. There are however ambiguos cases that will be interpreted in the wrong way. This can be handled by using a kernel that keeps track of the exact configuration of the pixel. The idea is to use neighbor weight from the positions in a binary number i.e. 2^N with $N \in [0 \ldots 9]$

Using a kernel like this: $\$j_4 = \begin{array}{|c|c|c|}\hline & 1 & \\ \hline 8 & 256 & 2 \\ \hline & 4 & \\ \hline \end{array} \qquad j_8 = \begin{array}{|c|c|c|}\hline 1 & 2 & 4 \\ \hline 128 & 256 & 8 \\ \hline 64 & 32 & 16 \\ \hline \end{array}\$$

The neighborhood is uniquely coded, less ambiguos than summing pixels.

- Coding pixels with values x:
  - $x = 0$ are background
  - $x < 256$ touch the skeleton
  - $x > 256$ are on the skeleton
- The number of branches and their orientation can be encoded by counting bit flips or using a LUT.

This coding is orientation sensitive, which may be to detailed for some applications.

Compare different skeleton analysis kernels

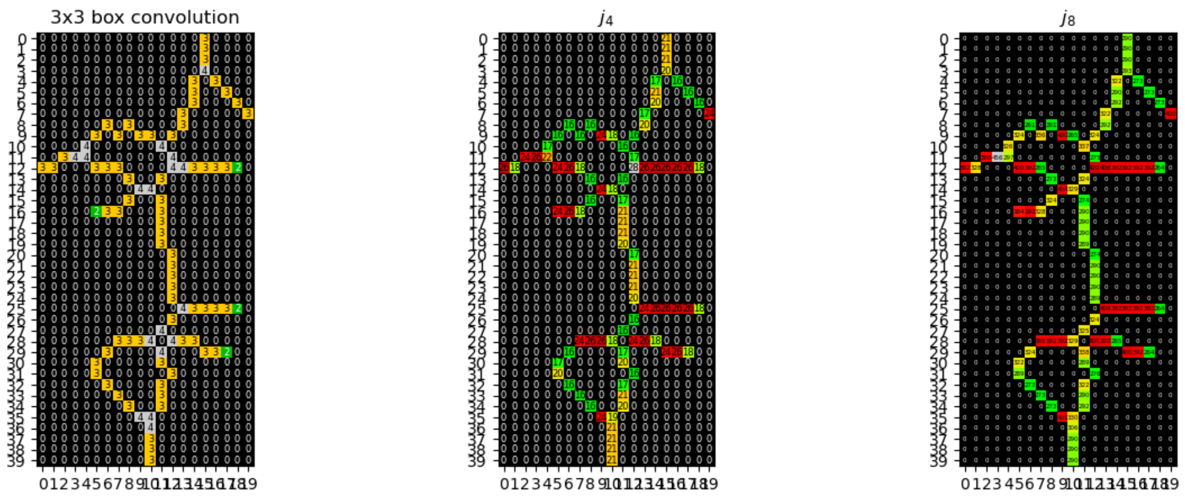
```
n_crop = neighbor_conv_masked[50:90, 40:60]

neighbor_j4 = convolve(opened_skeleton[50:90, 40:60].astype(int), np.array([[0,1,0],
 ↪[8,16,2],[0,4,0]]))
neighbor_j4[opened_skeleton[50:90, 40:60]==0]=0
neighbor_j8 = convolve(opened_skeleton[50:90, 40:60].astype(int), np.array([[1,2,4],
 ↪[128,256,8],[64,32,16]]))
neighbor_j8[opened_skeleton[50:90, 40:60]==0]=0
```

```
fig, (ax1,ax2,ax3) = plt.subplots(1, 3, figsize=(15, 5))
ps.heatmap(n_crop,cmap='nipy_spectral',ax=ax1,bgbox=False,precision=0,fontsize=6,
 ↪vmin=0, vmax=n_crop.max())

ax1.set_title('3x3 box convolution')
ps.heatmap(neighbor_j4,cmap='nipy_spectral',ax=ax2,bgbox=False,precision=0,fontsize=6,
 ↪vmin=0, vmax=neighbor_j4.max())

ax2.set_title(r'$j_4$')
ps.heatmap(neighbor_j8,cmap='nipy_spectral',ax=ax3,bgbox=False,precision=0,fontsize=4,
 ↪vmin=0, vmax=neighbor_j8.max())
```

(continues on next page)

(continued from previous page)

```
ax3.set_title(r'$j_8$');
```



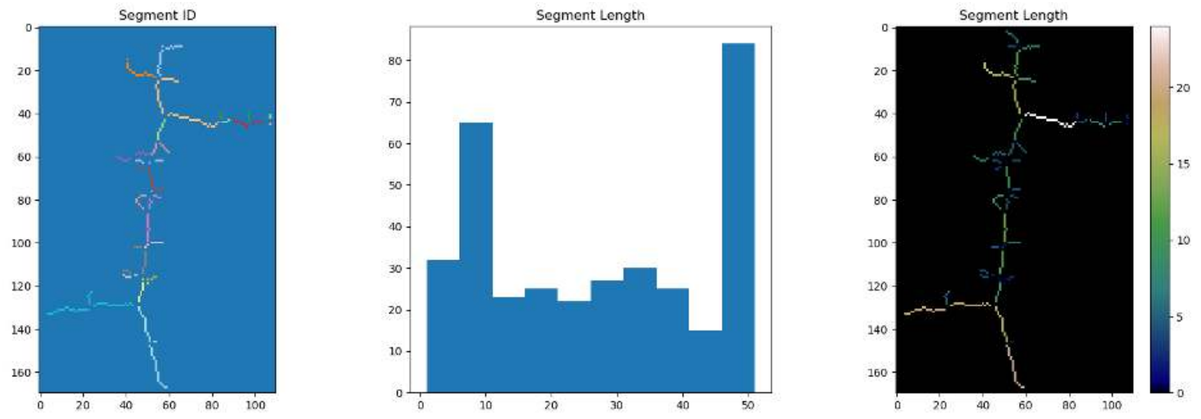
0.8 Dedicated pruning algorithms

- Ideally model-based
- Minimum branch length (using component labeling on the Count==3)
- Minimum branch width (using the distance map values)

0.8.1 Analyzing segment length

```
lab_seg = label(neighbor_conv_masked == 3)

label_length_img = np.zeros_like(lab_seg)
for i in np.unique(lab_seg[lab_seg > 0]):
    label_length_img[lab_seg == i] = np.sum(lab_seg == i)
```

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 6))
ax1.imshow(lab_seg, cmap='tab20',interpolation='none')
ax1.set_title('Segment ID')
ax2.hist(lab_seg[lab_seg > 0])
ax2.set_title('Segment Length')
ll_ax = ax3.imshow(label_length_img, cmap='gist_earth',interpolation='none')
ax3.set_title('Segment Length'); plt.colorbar(ll_ax);
```

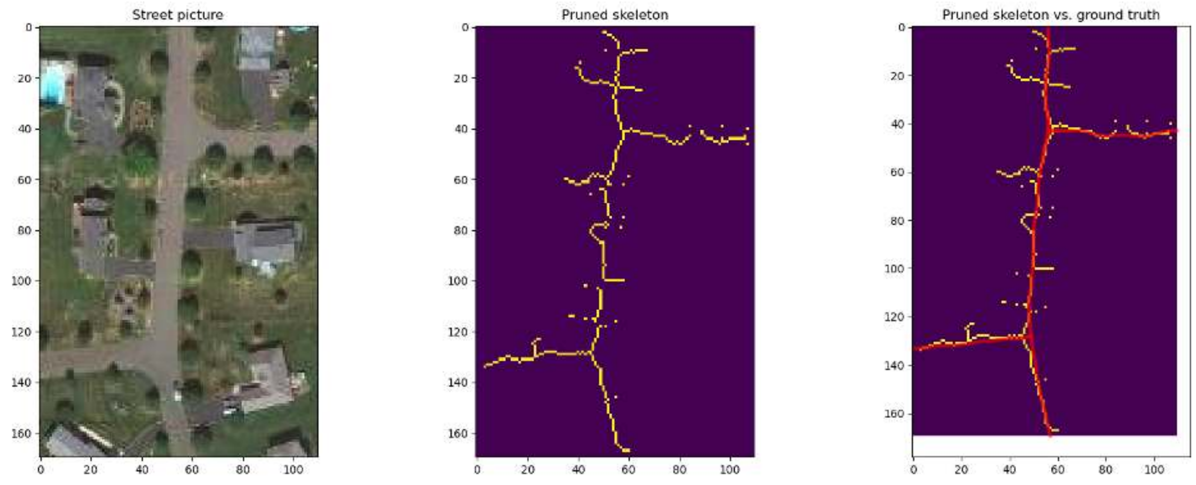


Looking at the pruned skeleton

The skeleton should be pruned to only contain line segments with more than five pixels. This is done by thresholding the `length_skeleton` variable. The end-points and junctions are added to get a complete skeleton again. These pixels are picked from the convolved image, value '2' for end-points and any value >3 is for the endpoints.

```
length_skeleton =   (label_length_img > 5) + \
                    (neighbor_conv_masked == 2) + \
                    (neighbor_conv_masked > 3)
```

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 7))
ax1.imshow(im_crop); ax1.set_title('Street picture')
ax2.imshow(length_skeleton, interpolation='none'); ax2.set_title('Pruned skeleton')
ax3.imshow(length_skeleton, interpolation='none'); ax3.set_title('Pruned skeleton vs.␣
↪ground truth')
ax3.scatter(mk_crop['x'], mk_crop['y'], s=mk_crop['width'],
            alpha=0.25, color='red', label='Ground Truth',);
```

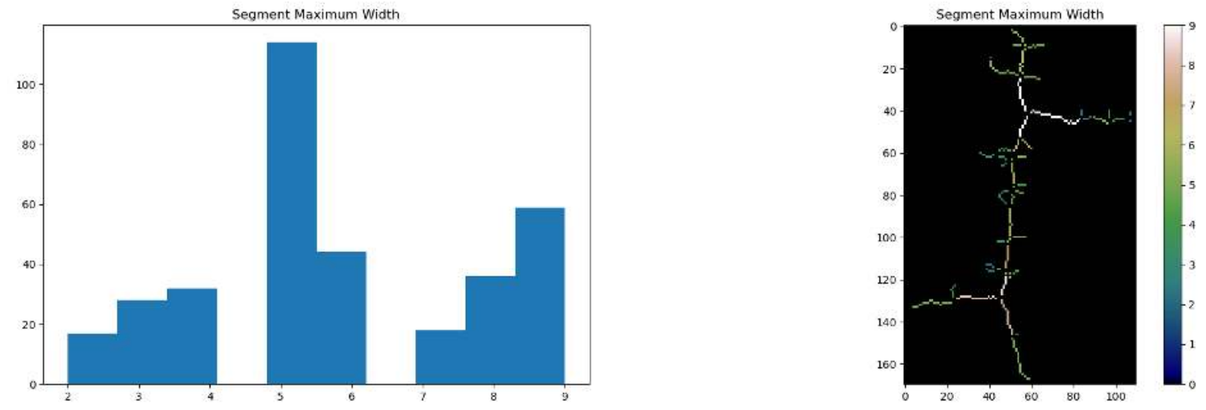


0.8.2 Analyzing maximum segment width

The segment width is a different metric for the skeleton pruning. The width is provided by computing the distance map of the original structure. Each segment can now be assigned the greatest distance fund at the pixels belonging to that segment.

```python
label_width_img = np.zeros_like(lab_seg)
for i in np.unique(lab_seg[lab_seg > 0]):
    label_width_img[lab_seg == i] = np.max(dist_map[lab_seg == i])
```

```python
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 6))
ax1.hist(label_width_img[label_width_img > 0])
ax1.set_title('Segment Maximum Width')

ll_ax = ax2.imshow(label_width_img, cmap='gist_earth',interpolation='none')
ax2.set_title('Segment Maximum Width'); plt.colorbar(ll_ax);
```

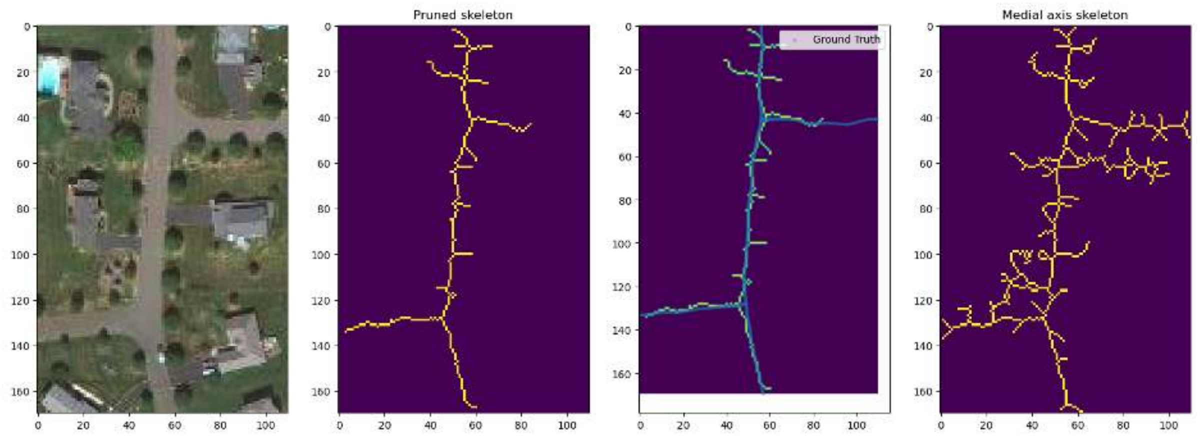


Pruning using structure width

Pruning for the structure width, we use a threshold on the segment maximum width skeleton. Like for the lenght pruning, we again add the junctions and the end points.

```python
width_skeleton =   (label_width_img > 4.5) \
                 + (neighbor_conv_masked == 2) \
                 + (neighbor_conv_masked > 3)
width_skeleton = label(width_skeleton) == 1
```

```python
fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(20, 7))
ax1.imshow(im_crop)
ax2.imshow(width_skeleton), ax2.set_title('Pruned skeleton')
ax3.imshow(width_skeleton)
ax3.scatter(mk_crop['x'], mk_crop['y'], s=mk_crop['width'],
            alpha=0.25, label='Ground Truth')
ax3.legend();
ax4.imshow(medial_axis(keep_lab_img, return_distance=False),interpolation='none'),
ax4.set_title('Medial axis skeleton');
```



0.8.3 An alternative

We know clean data is easier to work with…

Let's try to improve the input before creating the skeleton:

- Change segmentation method
- Use morphological filters

Other ways to do the segmentation…

We used a color space transformation

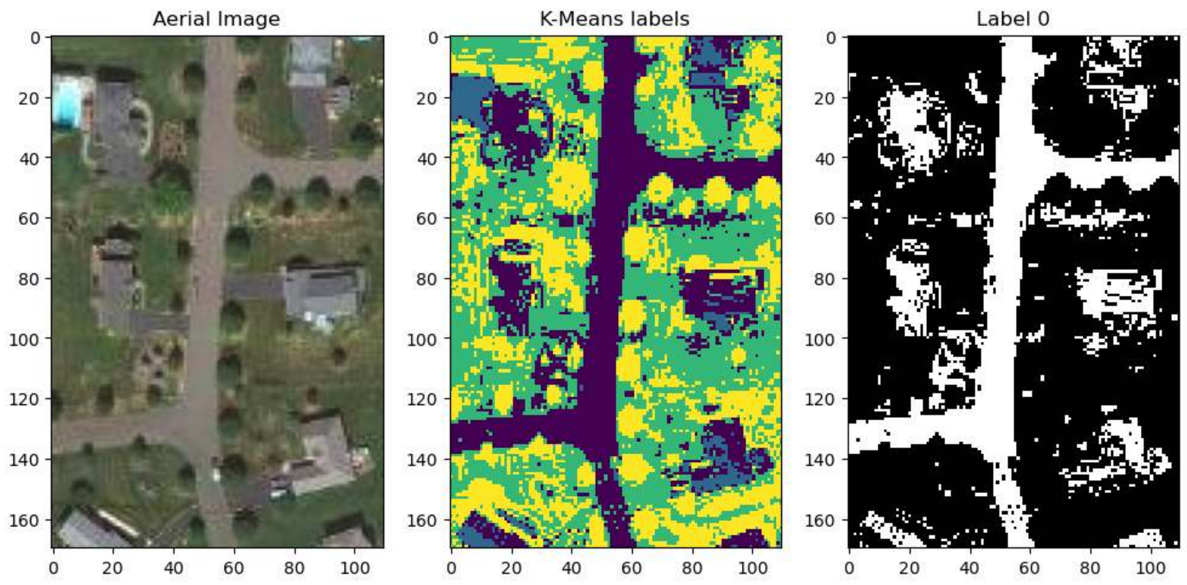
We could also use:

- Unsupervised segmentation like k-means
- Supervised segmentation like k-Nearest Neighbors (requires training)

Let's try k-Means clustering on the RGB channels

```
km=KMeans(n_clusters=4, random_state=42)
km.fit(dfRGB[['R','B']])
km_img = (km.labels_.reshape(im_crop.shape[:2])==0)
```

```
fig,ax=plt.subplots(1,3,figsize=(12,6))
ax[0].imshow(im_crop)
ax[0].set_title('Aerial Image')
ax[1].imshow(km.labels_.reshape(im_crop.shape[:2]),interpolation='none')
ax[1].set_title('K-Means labels')
ax[2].imshow(km.labels_.reshape(im_crop.shape[:2])==0,interpolation='none',cmap='gray
↪')
ax[2].set_title('Label 0');
```



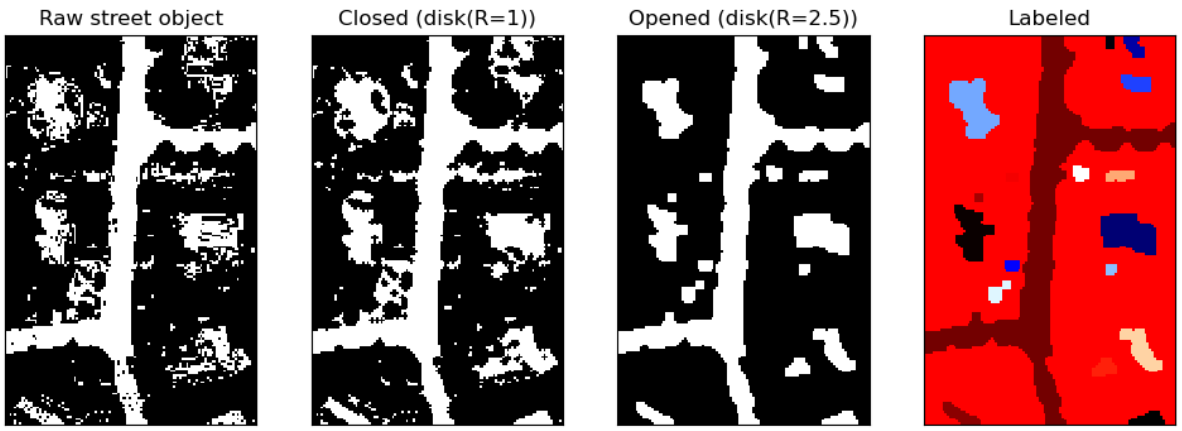
Cleaning up

We know from lecture 4 that morphological filtering can clean up messy segmentations with many misclassified pixels. In this case, we will use a sequence of closing and opening to improve the image before labelling to select the street object.

Note that the structure elements are of different sizes here. This is because the size of the false negatives is different from the false positives. Also the order of the opening and closing operations matter. In this particular case it made sense to apply closing before opening. Which depends if you create or destroy structures or connections and must be tested.

```
from skimage.morphology import opening, closing,disk

street = (km_img==1)
clean1 = closing(street,footprint=disk(1))
clean2 = opening(clean1,footprint=disk(2.5))

lab_img = label(clean2)
```

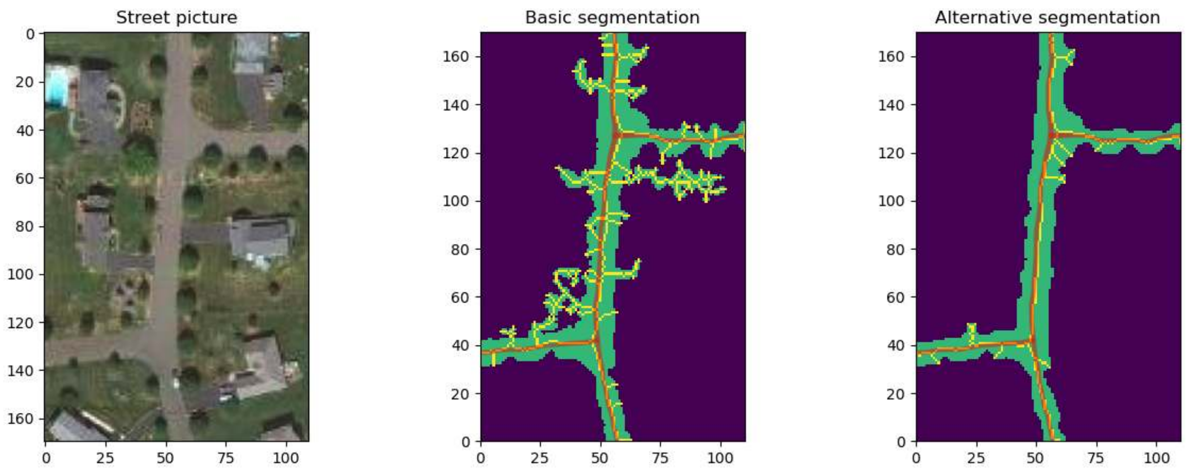
```
fig, ax = plt.subplots(1,4,figsize=(12,4))

ax[0].imshow(street,cmap='gray',interpolation='none')
ax[0].set(xticks=[],yticks=[],title='Raw street object')
ax[1].imshow(clean1,cmap='gray',interpolation='none')
ax[1].set(xticks=[],yticks=[],title='Closed (disk(R=1))')
ax[2].imshow(clean2,cmap='gray',interpolation='none')
ax[2].set(xticks=[],yticks=[],title='Opened (disk(R=2.5))')
ax[3].imshow(lab_img,cmap='flag',interpolation='none')
ax[3].set(xticks=[],yticks=[],title='Labeled');
```



Make skeleton

```
new_skeleton = medial_axis(lab_img==1, return_distance=False)
```

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 5))
ax1.imshow(im_crop); ax1.set_title('Street picture')
ax3.imshow(new_skeleton+(lab_img==1)*2, interpolation='none',extent=[0,skel.shape[1],
↪0,skel.shape[0]]);

ax3.set_title('Alternative segmentation')
ax3.scatter(mk_crop['x'], skel.shape[0]-mk_crop['y'], s=mk_crop['width'],
            alpha=0.1, color='red', label='Ground Truth',);
ax2.imshow(skel+(lab_img0==1)*2, interpolation='none',extent=[0,skel.shape[1],0,skel.
↪shape[0]]);
ax2.set_title('Basic segmentation')
ax2.scatter(mk_crop['x'], skel.shape[0]-mk_crop['y'], s=mk_crop['width'],
            alpha=0.1, color='red', label='Ground Truth',);
```



You can clearly see that the alternative approach creates a less ambiguous structure and the following skeleton is much leaner. The leaner skeleton requires less post processing. Please note that the basic segementation woulad also lead to a much leaner skeleton. This was not done in the lecture because we wanted to demonstrate the skeleton analysis which many loops and spurious branches.

0.9 Establish Topology

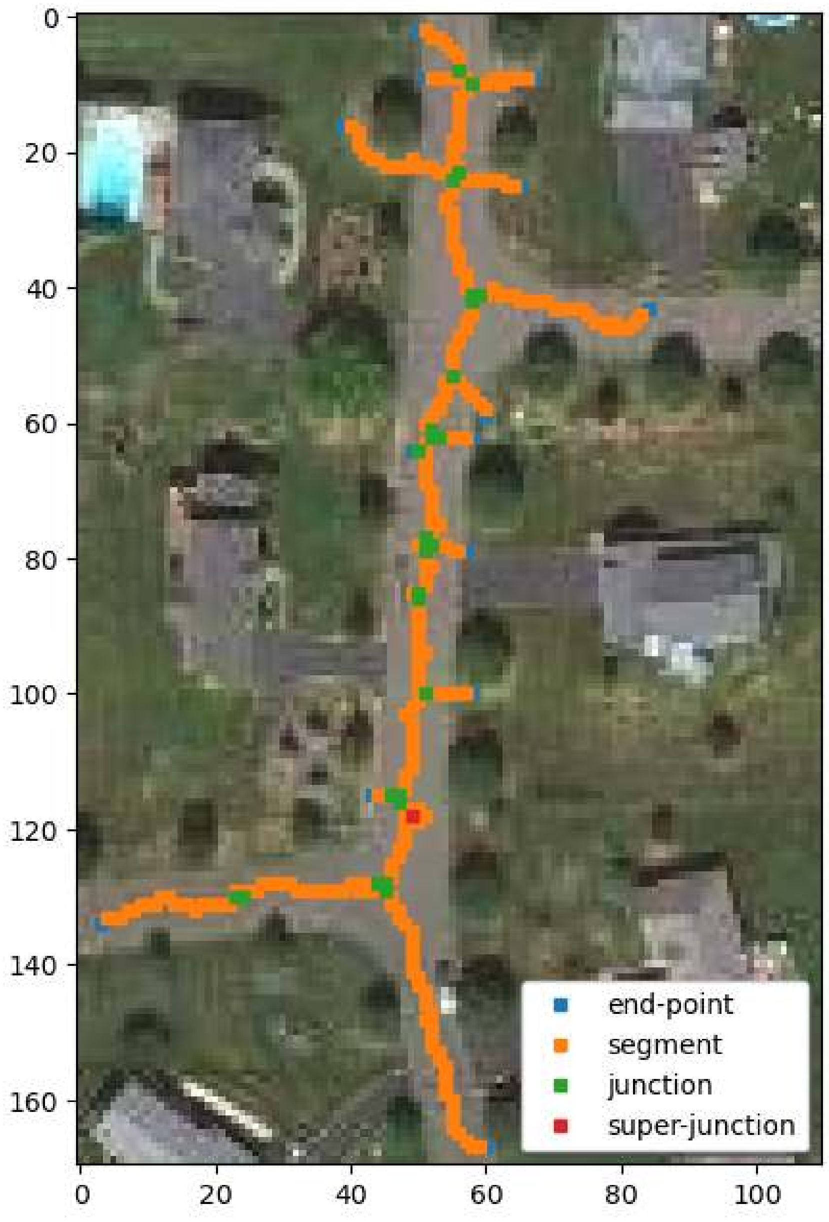
From the cleaned, pruned skeleton we can start to establish topology.

Using the same criteria as before we can break down the image into

- segments,
- junctions,
- and end-points

0.9.1 Topology of the street network

```
ws_neighbors = convolve(width_skeleton.astype(
    int), np.ones((3, 3)), mode='constant', cval=0)
ws_neighbors[~width_skeleton] = 0
fig, (ax1) = plt.subplots(1, 1, figsize=(5,10), dpi=100)
ax1.imshow(im_crop)
j_name = {1: 'dangling point', 2: 'end-point',
          3: 'segment', 4: 'junction', 5: 'super-junction'}
for j_count in np.unique(ws_neighbors[ws_neighbors > 0]):
    y_c, x_c = np.where(ws_neighbors == j_count)
    ax1.plot(x_c, y_c, 's',
             label=j_name.get(j_count, 'unknown'),
             markersize=5)

leg = ax1.legend(shadow=True, fancybox=True, frameon=True)
```



0.9.2 Getting Topology in Image Space

We want to determine which nodes are directly connected in this image so we can extract a graph. If we take a simple case of two nodes connected by one edge and the bottom node connected to another edge going nowhere.

We have an image with edges and nodes: \$ $\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & e & 0 & 0 \\ 0 & 0 & n & e \end{bmatrix}$ \$

We can use component labeling to identify each node and each edge uniquely

Node Labels

$$N_{lab} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix}$$

Edge Labels

$$E_{lab} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

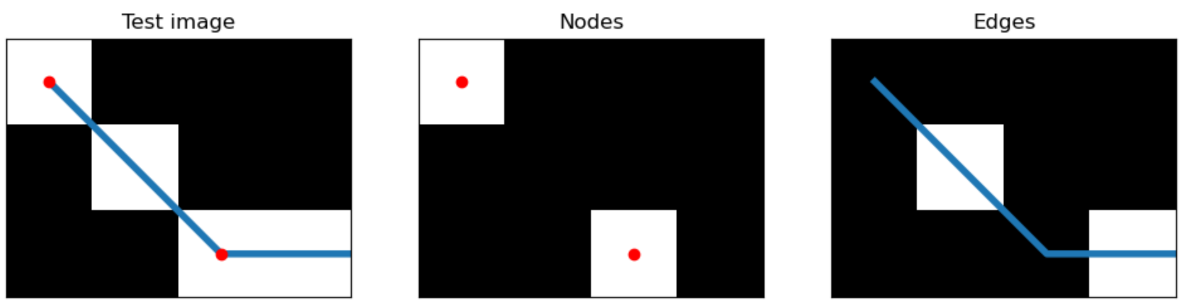
We can then use a dilation operation on the nodes and the edges to see which overlap

Small topology analysis example

In this example we create images with nodes and edges respectively. The items in the images are labelled to tell which analyze. The next step is to dilate each node in the image to tell which edges it is connected to. An edge belongs to a node if there is an overlap between the dilated node (its neighborhood) and any pixel of the edge. The edges in this example are very short, only one pixel. This is normally not the case.

```python
from skimage.morphology import dilation
n_img = np.zeros((3, 4))
e_img = np.zeros_like(n_img)
n_img[0, 0] = 1
e_img[1, 1] = 1
n_img[2, 2] = 1
e_img[2, 3] = 1
```

```python
fig,ax=plt.subplots(1,3,figsize=(12,4))

ax[0].imshow(n_img+e_img,cmap='gray')
ax[0].set(title='Test image',xticks=[],yticks=[])
ax[0].plot([0,1,2,3.5],[0,1,2,2],linewidth=4)
ax[0].plot([0,2],[0,2],'or')
ax[1].imshow(n_img,cmap='gray')
ax[1].set(title='Nodes',xticks=[],yticks=[])
ax[1].plot([0,2],[0,2],'or')
ax[2].imshow(e_img,cmap='gray')
ax[2].set(title='Edges',xticks=[],yticks=[]);
ax[2].plot([0,1,2,3.5],[0,1,2,2],linewidth=4);
```

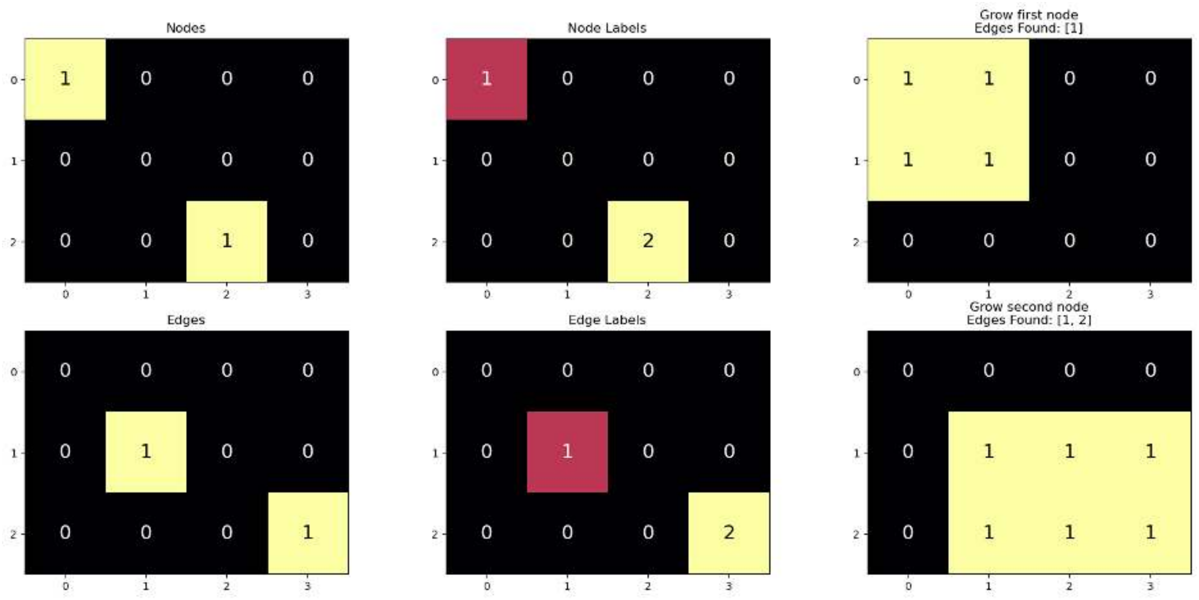


The resulting analysis

Here, you can see the result of the edge-node connection code.

```
n_labs = label(n_img)
e_labs = label(e_img)

n_grow_1 = dilation(n_labs == 1, np.ones((3, 3)))
n_grow_2 = dilation(n_labs == 2, np.ones((3, 3)))
```

```
fig, ((ax1, ax3, ax5), (ax2, ax4, ax6)) = plt.subplots(2, 3, figsize=(20,9))


ps.heatmap(n_img,bgbox=False, ax=ax1,precision=0,fontsize=18)
ax1.set_title('Nodes')

ps.heatmap(e_img,bgbox=False, ax=ax2,precision=0,fontsize=18)
ax2.set_title('Edges')

# labeling

ps.heatmap(n_labs,bgbox=False, ax=ax3,precision=0,fontsize=18); ax3.set_title('Node␣
↪Labels')

ps.heatmap(e_labs, ax=ax4, bgbox=False,precision=0, fontsize=18); ax4.set_title('Edge␣
↪Labels')

# growing

ps.heatmap(n_grow_1, ax=ax5, bgbox=False,precision=0, fontsize=18);
ax5.set_title('Grow first node\n{} {}'.format('Edges Found:', [x for x in np.unique(e_
↪labs[n_grow_1 > 0]) if x > 0]))


ps.heatmap(n_grow_2, ax=ax6, bgbox=False,precision=0, fontsize=18)
ax6.set_title('Grow second node\n{} {}'.format('Edges Found:', [x for x in np.
↪unique(e_labs[n_grow_2 > 0]) if x > 0]));
```



Here, you see that node 1 only overlaps edge 1 while node 2 overlaps both edges 1 and 2 as it is between them.

Analysing the topology of the street network

Using skeleton pixel neightborhood analysis with a 3x3 kernel as before we have

Pixel counts	Interpretation	Category
0	Background	Background/Don't care
1	Isolated point	Don't care
2	End point	Node
3	Line segment	Edge
4	Three-way Junction	Node
5	Four-way junction	Node

Identify edges and nodes

The first step of the analysis is to label nodes and edges. This is done by using the connected components labeling of the neighborhood filtered image.

1. The nodes are identified by neigborhood counts greater than 3.
2. The edges are identified as neighborhood counts equal to 3.

These two images will be labelled to give the segments id numbers.

The next step is to create a dictionary of all nodes containing the CoG and the average width at the node.

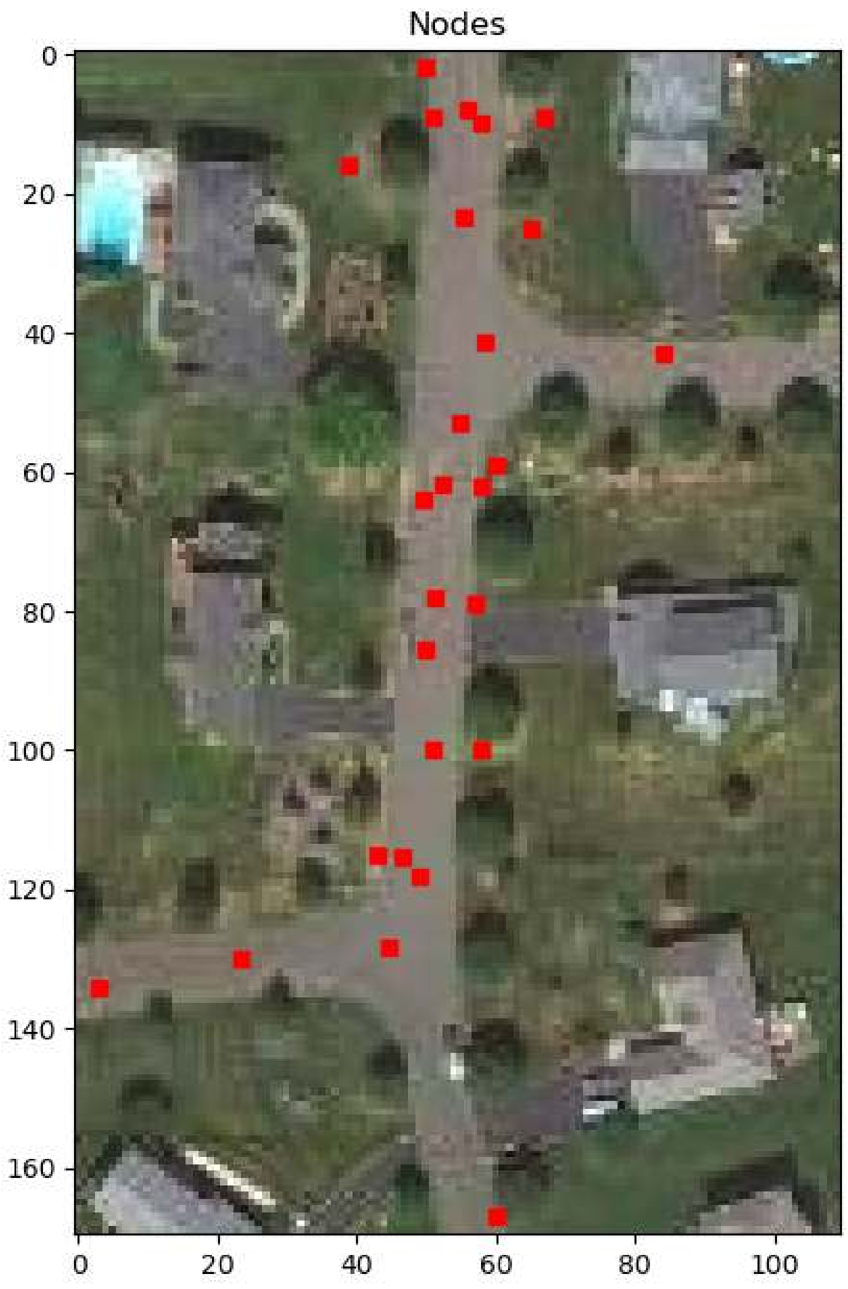
```
node_id_image = label((ws_neighbors > 3) | (ws_neighbors == 2))
edge_id_image = label(ws_neighbors == 3)

node_dict = {}
for c_node in np.unique(node_id_image[node_id_image > 0]):
    y_n, x_n = np.where(node_id_image == c_node)
```

(continues on next page)

(continued from previous page)

```
    node_dict[c_node] = {'x': np.mean(x_n),
                         'y': np.mean(y_n),
                         'width': np.mean(dist_map[node_id_image == c_node])}
```

```
fig,ax1 = plt.subplots(1,figsize=(12,8))
ax1.imshow(im_crop)
for node in node_dict :
    ax1.plot(node_dict[node]['x'], node_dict[node]['y'], 'rs')

ax1.set_title("Nodes");
```



Analyze edges and connections

We now want to analyze the network by connecting nodes with edges and create a dictioary with more information about the connected segments.

The connection is done in the same way as the toy example with two node and two edges above, i.e.:

1. Dilate each edge id the edge image with a 3x3 structure element
2. Find the nodes that are covered by the dilated edges (there should be only two)
3. Create a dictionary with measured information like
    1. connected nodes
    2. length in pixels
    3. Euclidean distance between the end points
    4. Max width of the edge
    5. Average width of the edge
4. Compute the edge matrix that tells which nodes are connected and how many pixels the segment contains, i.e. the length.

the widths are given by the distance map value of the original structure at the skeleton line.

```
edge_dict = {}
edge_matrix = np.eye(len(node_dict)+1)

s_nodes = []
e_nodes = []
for c_edge in np.unique(edge_id_image[edge_id_image > 0]):
    edge_grow_mask = dilation(edge_id_image == c_edge, np.ones((3, 3)))
    v_nodes = np.unique(node_id_image[edge_grow_mask > 0])
    v_nodes = [v for v in v_nodes if v > 0]
    if c_edge<6 :
        print('Edge', c_edge, 'connects', v_nodes)
    if len(v_nodes) == 2:
        edge_dict[c_edge] = {'start': v_nodes[0],
                             'end': v_nodes[-1],
                             'length': np.sum(edge_id_image == c_edge),
                             'euclidean_distance': np.sqrt(np.square(node_dict[v_
↪nodes[0]]['x'] -
                                                                     node_dict[v_
↪nodes[-1]]['x']) +
                                                           np.square(node_dict[v_
↪nodes[0]]['y'] -
                                                                     node_dict[v_
↪nodes[-1]]['y'])
                                                          ),
                             'max_width': np.max(dist_map[edge_id_image == c_edge]),
                             'mean_width': np.mean(dist_map[edge_id_image == c_edge])}
        edge_matrix[v_nodes[0], v_nodes[-1]] = np.sum(edge_id_image == c_edge)
        edge_matrix[v_nodes[-1], v_nodes[0]] = np.sum(edge_id_image == c_edge)
        s_nodes.append(node_dict[v_nodes[0]])
        e_nodes.append(node_dict[v_nodes[-1]])
```

```
Edge 1 connects [1, 2]
Edge 2 connects [2, 3]
Edge 3 connects [2, 5]
Edge 4 connects [4, 5]
Edge 5 connects [5, 7]
```

Looking at the segment measurements

```
skel_df = pd.DataFrame(edge_dict).transpose()
skel_df.sample(10)
```

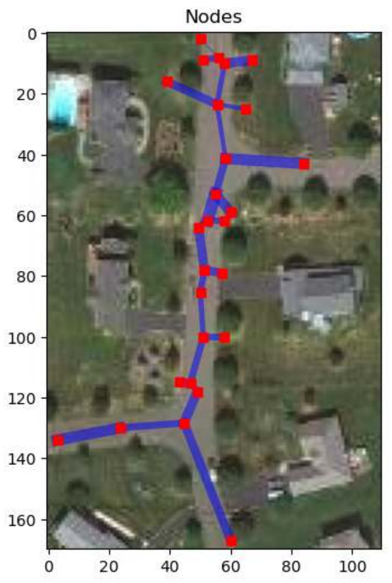
```
    start   end  length  euclidean_distance  max_width  mean_width
15   15.0  16.0    12.0           14.119529   6.708204    5.520520
22   19.0  22.0    14.0           15.933891   7.000000    6.068130
10    9.0  11.0    10.0           12.133516   9.433981    8.043840
5     5.0   7.0    12.0           13.729530   6.403124    5.525122
8     7.0   9.0    16.0           18.057008   9.848858    7.229723
14   13.0  15.0     1.0            3.670453   5.830952    5.830952
23   21.0  22.0     2.0            3.681787   5.000000    4.500000
12   11.0  12.0     5.0            7.810250   6.000000    4.047214
13   13.0  14.0     4.0            5.676462   5.000000    3.630363
27   24.0  25.0    19.0           21.232182   8.062258    5.510590
```

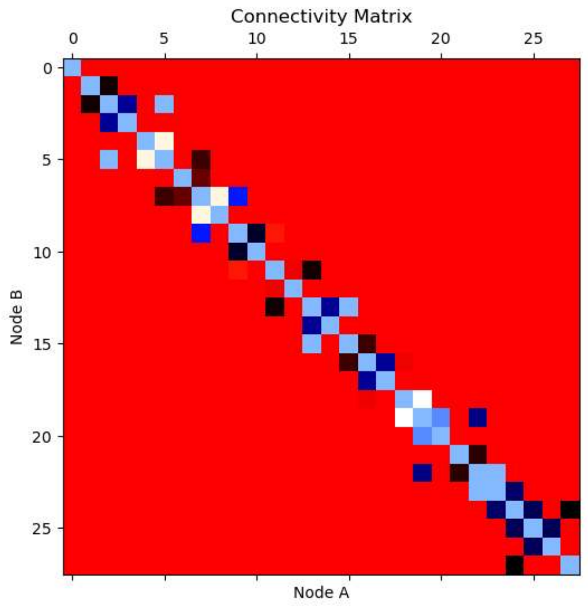
Display the analysis

Here we show the network with the node connected by the edges. The edge line width is controlled by the segment with from the analysis.

The connectivity matrix shows that the node labels are relatively local in this case. The exception is at the branches where there the node numbering go a bit apart.

```
fig, (ax1,ax2) = plt.subplots(1,2,figsize=(15,6))
ax1.imshow(im_crop)

ax1.set_title("Nodes");
for c_edge,(s_node,e_node) in enumerate(zip(s_nodes,e_nodes)) :
    ax1.plot([s_node['x'], e_node['x']],
          [s_node['y'], e_node['y']], 'b-', linewidth=np.mean(dist_map[edge_id_image␣
 ↪== c_edge]), alpha=0.5)

for node in node_dict :
    ax1.plot(node_dict[node]['x'], node_dict[node]['y'], 'rs')

ax2.matshow(edge_matrix, cmap='flag')
ax2.set(title='Connectivity Matrix',xlabel='Node A',ylabel='Node B');
```





0.10 Skeleton analysis - Tortuosity

One of the more interesting ones in material science is called tortuosity and it is defined as the ratio between the arc-length of a *segment* and the distance between its starting and ending points.

$$\tau = \frac{C}{L}$$

- L - Distance between start and end point
- C - Length of the curve segment

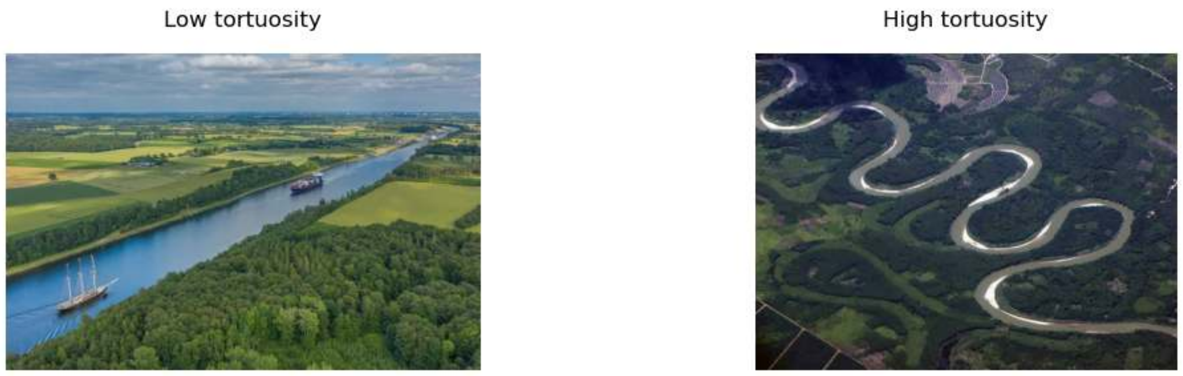
0.10.1 Interpreting tortuosity

A high degree of tortuosity indicates that the network is convoluted and is important when estimating or predicting flow rates.

Specifically

- in geology it is an indication that diffusion and fluid transport will occur more slowly
- in analytical chemistry it is utilized to perform size exclusion chromatography
- in vascular tissue it can be a sign of pathology.

```
fig,axs = plt.subplots(1,2,figsize=(12,3))
imgs=['figures/canal.jpg',"figures/meander.jpg"]
titles=['Low tortuosity','High tortuosity']
for ax,img,title in zip(axs,imgs,titles) :
    ax.imshow(plt.imread(img))
    ax.set(xticks=[],yticks=[])
    ax.set_title(title,y=1.05)
    ax.axis('off')
plt.tight_layout()
```

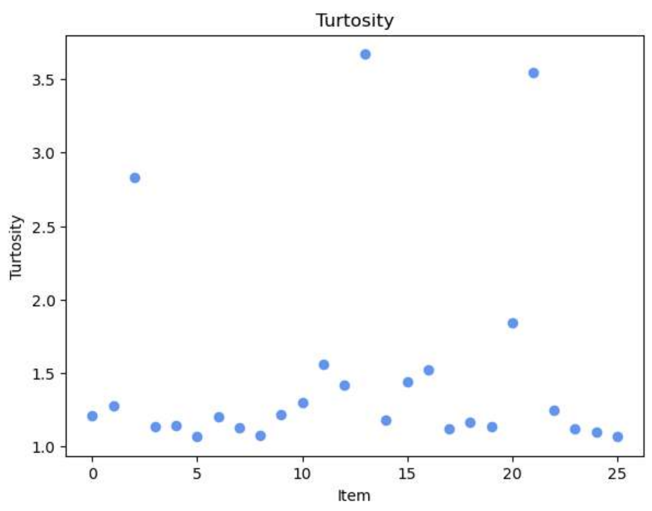


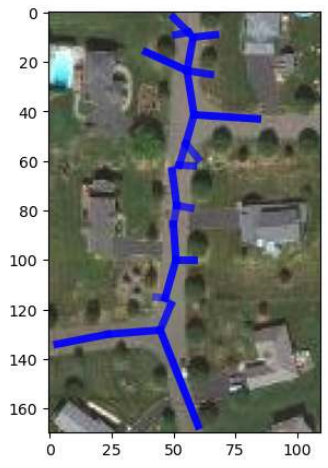
0.10.2 Tortuosity of the street

We see that the tortuosity is relatively low in the street example. This is because most elements are rather straight here.

The opacity is controlled in the network shown in the image to the right. This shows that most elements are opaque, the transparent segmenents are relatively short. From this observation we can draw the conclusion that the tortuosity is less precise for short segments. This makes sense a we can only use the discrete steps given by the pixels.

```
fig, (ax0,ax1) = plt.subplots(1, 2, figsize=(15, 5))
ax1.imshow(im_crop)
t=[]
for _, d_values in edge_dict.items():
    v_nodes = [d_values['start'], d_values['end']]
    t = t + [(d_values['length'])/d_values['euclidean_distance']]
    s_node = node_dict[v_nodes[0]]
    e_node = node_dict[v_nodes[-1]]
    tt=t[-1]

    if tt>1 :
        tt=1.0
    ax1.plot([s_node['x'], e_node['x']],
             [s_node['y'], e_node['y']], 'b-',
             linewidth=5, alpha=tt)
ax0.plot(1.0/np.array(t),'o', color='cornflowerblue');
ax0.set(title='Turtosity', xlabel='Item', ylabel='Turtosity');
```





0.10.3 Further ways to visualize the network

1. Randomly organized graph
   - Nodes colored by the width
   - Edges colored by the length and width set by segment width
2. Add width and length information in the original picture

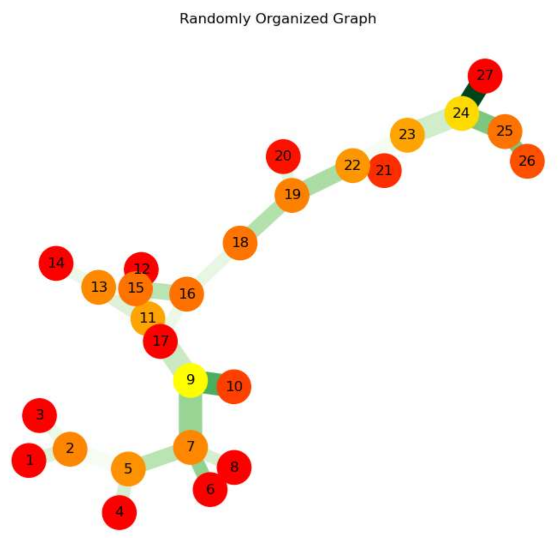
```
import networkx as nx

G = nx.Graph()
for k, v in node_dict.items():
    G.add_node(k, weight=v['width'])
for k, v in edge_dict.items():
    G.add_edge(v['start'], v['end'], **v)
```

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 8))
nx.draw_spring(G, ax=ax1, with_labels=True,
               node_color=[node_dict[k]['width']
                           for k in sorted(node_dict.keys())],
               node_size=800,
               cmap=plt.cm.autumn,
               edge_color=[G.edges[k]['length'] for k in list(G.edges.keys())],
               width=[2*G.edges[k]['max_width'] for k in list(G.edges.keys())],
               edge_cmap=plt.cm.Greens
              )
ax1.set_title('Randomly Organized Graph')
ax2.imshow(im_crop)
nx.draw(G,
        pos={k: (v['x'], v['y']) for k, v in node_dict.items()},
        ax=ax2,
        node_color=[node_dict[k]['width'] for k in sorted(node_dict.keys())],
        node_size=50,
        cmap=plt.cm.autumn,
        edge_color=[G.edges[k]['length'] for k in list(G.edges.keys())],
```

(continues on next page)

(continued from previous page)

```
          width=[2*G.edges[k]['max_width'] for k in list(G.edges.keys())],
          edge_cmap=plt.cm.Blues,
          alpha=0.5,
          with_labels=False)
```



0.10.4 Graph Analysis

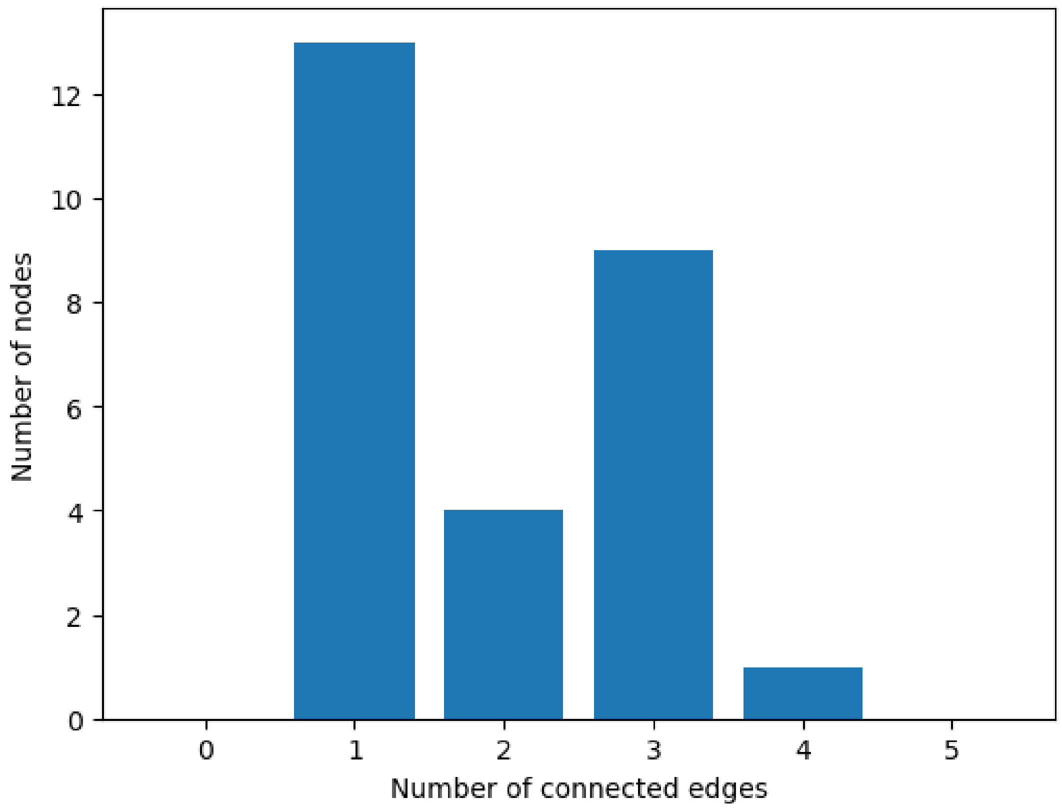
Once the data has been represented in a graph form, we can begin to analyze some of graph aspects of it, like the degree and connectivity plots.

In graph theory, the degree of a vertex in a graph is the number of edges incident to that vertex. For directed graphs, the degree can be further categorized into the in-degree (number of edges coming into the vertex) and out-degree (number of edges leaving the vertex).

The degree of a graph is the maximum degree among all its vertices. It provides insights into the connectivity of the graph and is often used to classify graphs into different types, such as regular graphs (where all vertices have the same degree) or irregular graphs (where vertices have varying degrees).

Additionally, the sum of degrees of all vertices in a graph is twice the number of edges, which is a consequence of the Handshaking Lemma in graph theory. This property is often useful in various graph-theoretical proofs and calculations.

```
degree_sequence = sorted([d for n, d in G.degree()],
                         reverse=True)   # degree sequence

hb,ha = np.histogram(degree_sequence,bins=np.arange(0,7))

plt.bar(ha[:-1],hb ,align="center"), plt.xlabel('Number of connected edges'), plt.
 ↪ylabel('Number of nodes');
```



0.10.5 Skeletons going 3D

- Object topology must still be preserved
- More complex neighborhoods to analyze

A skeletonization algorithm survey

Application of 3D skeletons: Root networks

Motivation

Why analyzing root networks

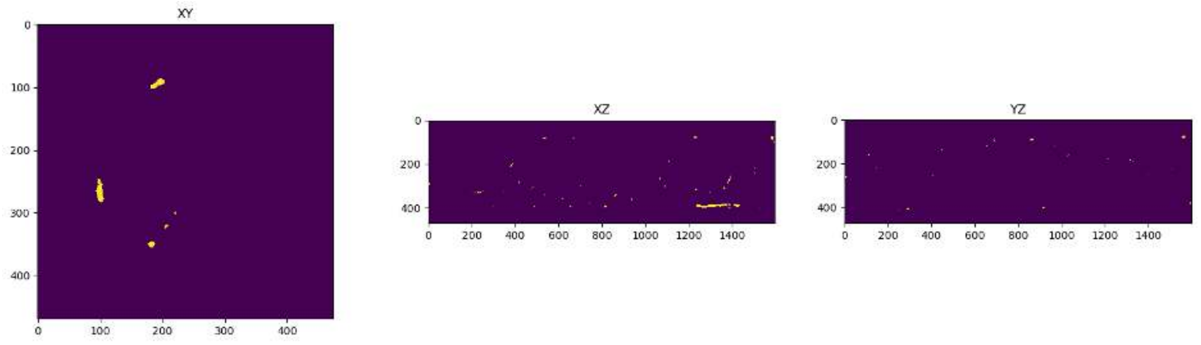
- Soil stability
- Water uptake
- Growth under different contitions

Way to analyze the network

- Root network morphology
  - Segment length
  - Branching
  - Branch orientation
  - Turtosity
- Influence volumes

Load volume with roots

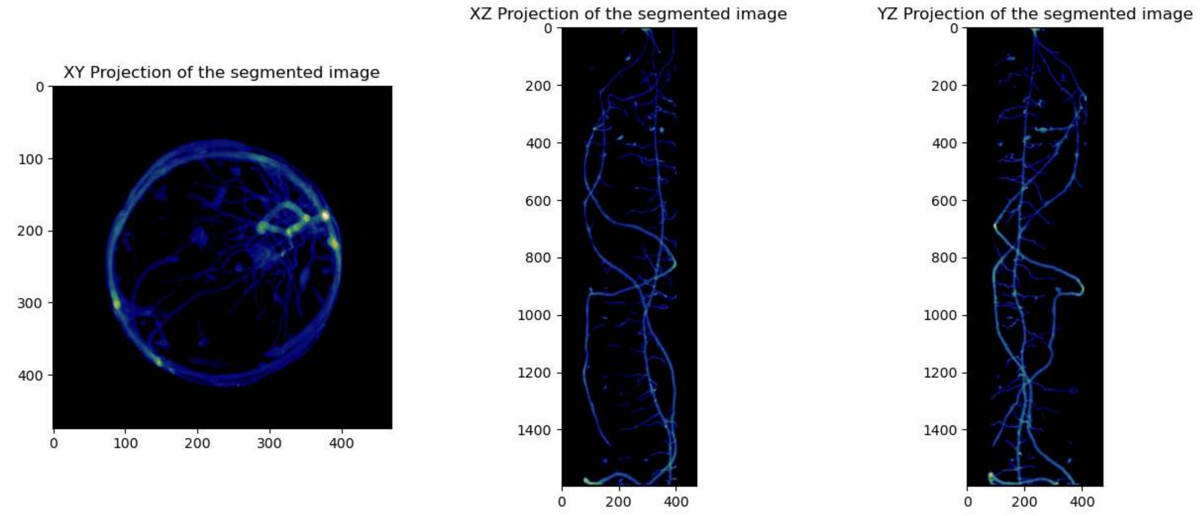
```
from pathlib import Path
import zipfile

# Define the target file and zip source
target_file = Path("data/Cropped_prediction_8bit.npy")
zip_source = Path("data/Cropped_prediction_8bit.npy.zip")

# Check if the file exists
if not target_file.exists():
    print(f"{target_file} not found. Extracting from {zip_source}...")

    with zipfile.ZipFile(zip_source, 'r') as zip_ref:
        zip_ref.extractall(target_file.parent)  # Extract to the same directory as␣
↪the target file

root = np.load('data/Cropped_prediction_8bit.npy')
fig, (ax1,ax2,ax3) = plt.subplots(1,3,figsize=(20,5))
ax1.imshow(root[:,:,700],interpolation='none'); ax1.set_title('XY')
ax2.imshow(root[:,220,:],interpolation='none'); ax2.set_title('XZ')
ax3.imshow(root[220,:,:],interpolation='none'); ax3.set_title('YZ');
```



Let's look at the projection instead

The slice planes did not reveal the structure of the volume very well. Here is an alternative. Sum all pixels along one axis to obtain a 2D projection of the structure. This is a useful approach for sparse structures.

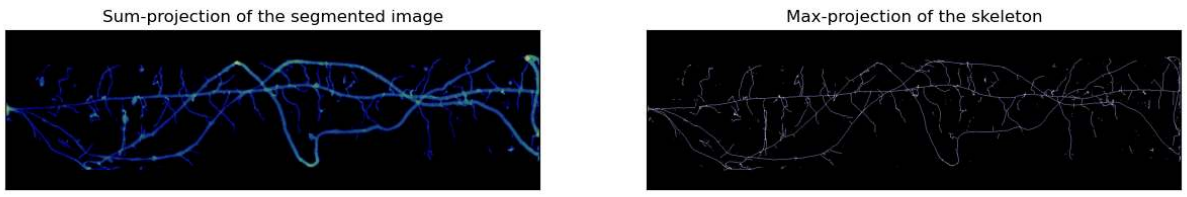
```
fig,ax=plt.subplots(1,3,figsize=(15,6))
planes = ['XY','XZ','YZ']
axes = [2,1,0]
for idx,(axis,plane) in enumerate(zip(axes,planes)):
    ax[idx].imshow(root.mean(axis=axis).transpose(),cmap='gist_earth'),
    ax[idx].set_title(f'{plane} Projection of the segmented image');
```


XY Projection of the segmented image
XZ Projection of the segmented image
YZ Projection of the segmented image
0
100
200
300
400
600
800
1000
1200
1400

Create the 3D skeleton

```
skel = skeletonize_3d(root)
```

```
fig,ax=plt.subplots(1,2,figsize=(15,4))
ax[0].imshow(root.mean(axis=axis),cmap='gist_earth'),
ax[0].set(title='Sum-projection of the segmented image',xticks=[],yticks=[]);
ax[1].imshow(skel.max(axis=0), cmap="bone"),
ax[1].set(title='Max-projection of the skeleton',xticks=[],yticks=[]);
```


Sum-projection of the segmented image
Max-projection of the skeleton

Detailed 3D view of skeleton

0.11 Segmenting touching items

Watershed is a method for segmenting objects without using component labeling.

- It utilizes the shape of structures to find objects
- From the distance map we can make out substructures with our eyes
- But how to we find them?!

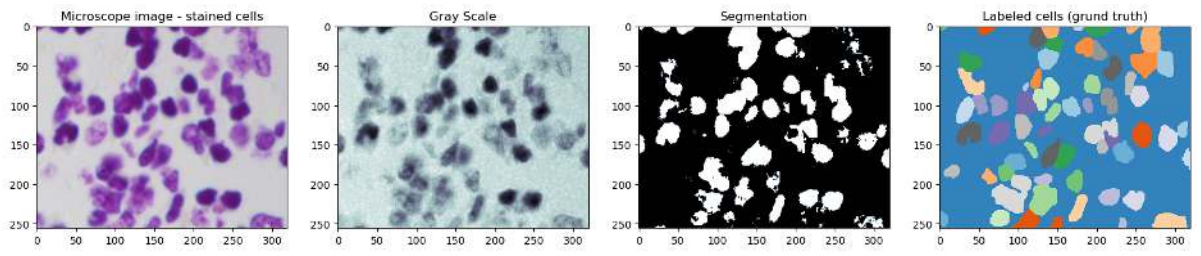
0.11.1 Watershed

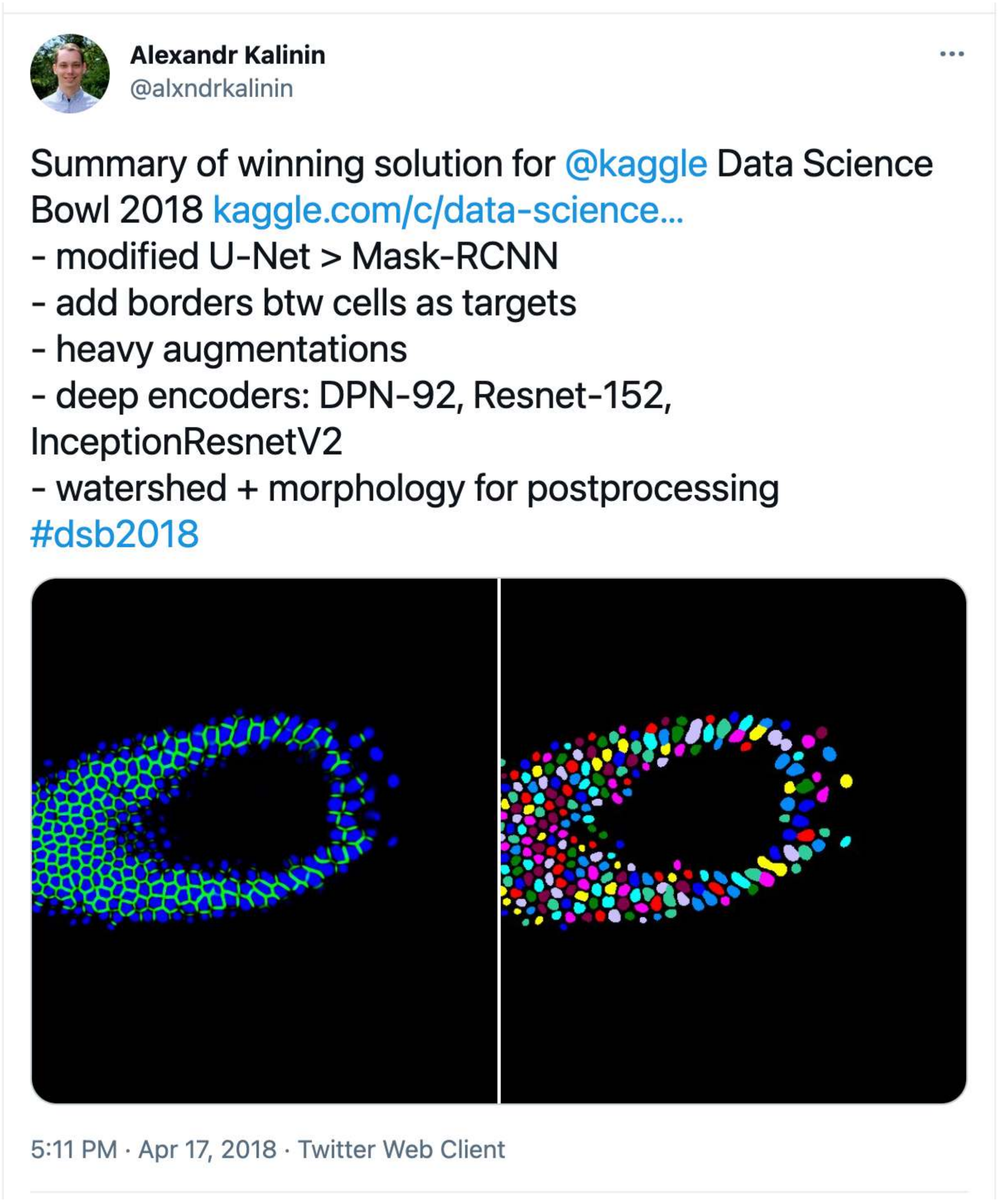
We use a sample image now from the Datascience Bowl 2018 from Kaggle. The challenge is to identify nuclei in histology images to eventually find cancer better. The winner tweeted about the solution here

0.11.2 Let's load a cell image

The test image for this part is a microscope image of stained cells. The original is a color image, but we will convert it to grayscale for the segmentation. The thresholded image has quite many misclassifications as you can see in the provided ground truth images to the very right.

```
from skimage.filters import threshold_otsu
from skimage.color import rgb2hsv
import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imread
%matplotlib inline

rgb_img = imread("../Lecture-02/figures/dsb_sample/slide.png")[:, :, :3]
gt_labs = imread("../Lecture-02/figures/dsb_sample/labels.png")
bw_img = rgb2hsv(rgb_img)[:, :, 2]

fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(20, 6), dpi=100)
ax1.imshow(rgb_img, cmap='bone'),ax1.set_title('Microscope image - stained cells')
ax2.imshow(bw_img, cmap='bone'),ax2.set_title('Gray Scale')
ax3.imshow(bw_img < threshold_otsu(bw_img), cmap='bone'), ax3.set_title('Segmentation
↪')
ax4.imshow(gt_labs, cmap='tab20c',interpolation='None'),ax4.set_title('Labeled cells␣
↪(grund truth)');
```





Alexandr Kalinin
@alxndrkalinin

Summary of winning solution for @kaggle Data Science Bowl 2018 kaggle.com/c/data-science...
- modified U-Net > Mask-RCNN
- add borders btw cells as targets
- heavy augmentations
- deep encoders: DPN-92, Resnet-152, InceptionResnetV2
- watershed + morphology for postprocessing
#dsb2018

5:11 PM · Apr 17, 2018 · Twitter Web Client

Fig. 1: Tweet by Alexandr Kalinin

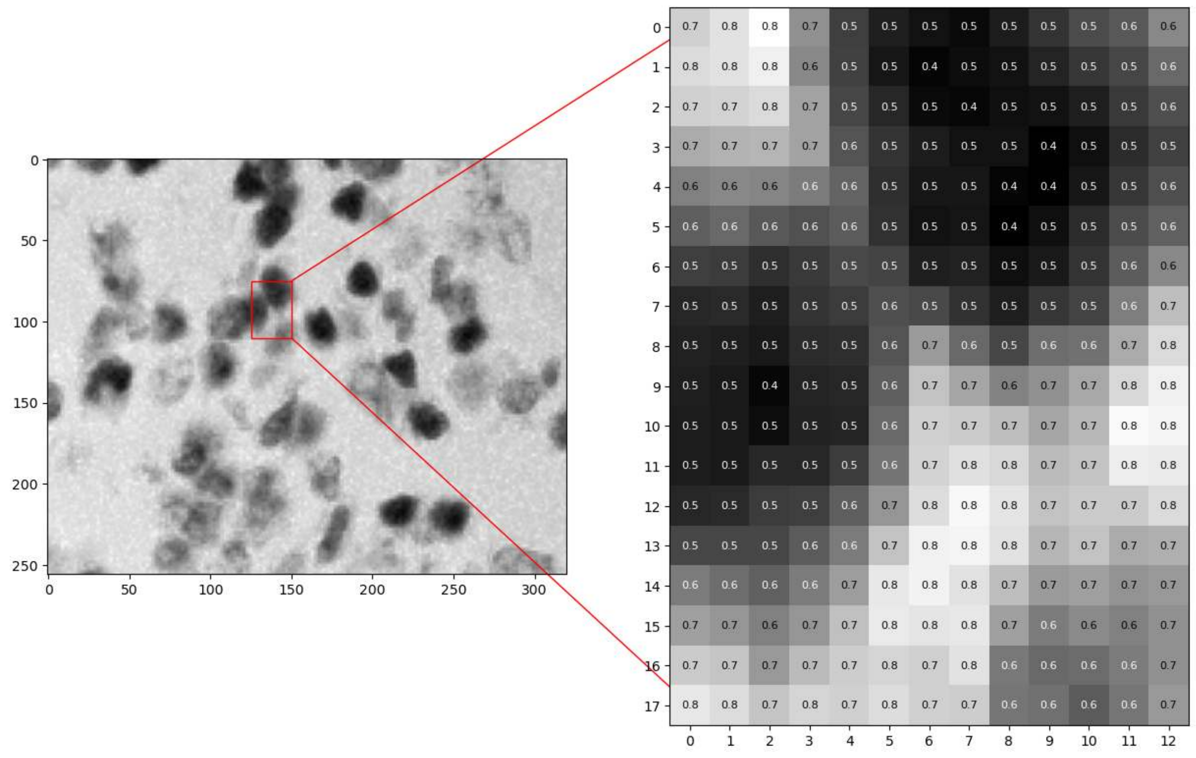
0.11.3 Try component labeling

First reduce the data

The full image has too much detail. We crop the image to only contain few cells.

```python
from skimage.morphology import label
bw_roi        = bw_img[75:110:2, 125:150:2]
```

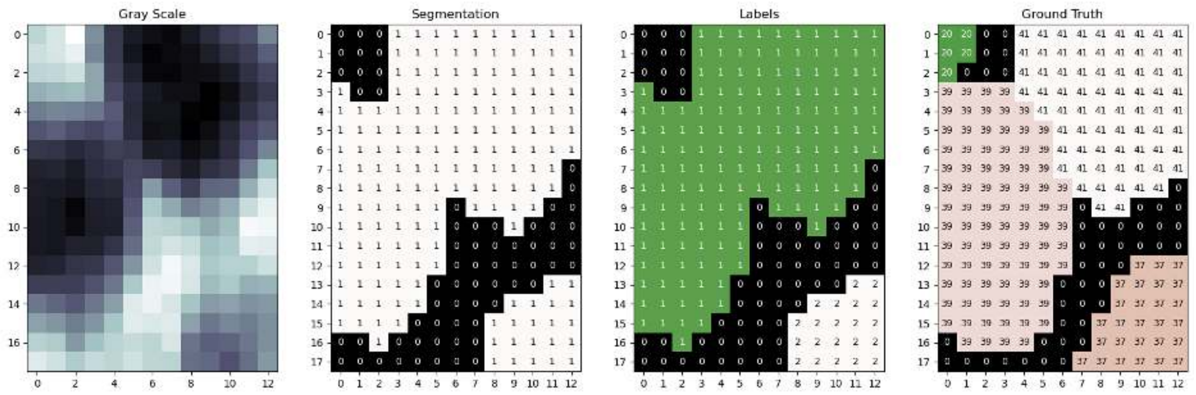
```python
from matplotlib.patches import Rectangle
from matplotlib.patches import ConnectionPatch
fig, ax = plt.subplots(1, 2, figsize=(15, 10))
ax[0].imshow(bw_img,cmap='gray');
rect = Rectangle(xy=(125.0, 75.0), height=35, width=25, linewidth=1, edgecolor='r',␣
↪facecolor='none')
ax[0].add_patch(rect)
ps.heatmap(bw_roi,cmap='gray',ax=ax[1],bgbox=False,precision=1,fontsize=8) # Show␣
↪every 6th pixel)

con2 = ConnectionPatch(xyA=(150,75), xyB=(0,0), coordsA="data", coordsB="data",
                       axesA=ax[0], axesB=ax[1], color="red", lw=1)
ax[0].add_artist(con2)
con1 = ConnectionPatch(xyA=(150,110), xyB=(0,17), coordsA="data", coordsB="data",
                       axesA=ax[0], axesB=ax[1], color="red", lw=1)
ax[0].add_artist(con1);
```



Trying component labelling on the cells

```
bw_roi_seg   = bw_roi < threshold_otsu(bw_img)
bw_roi_label = label(bw_roi_seg)
```

```
import seaborn as sns
fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(20, 6), dpi=100)
ax1.imshow(bw_roi, cmap='bone'); ax1.set_title('Gray Scale')
ps.heatmap(bw_roi_seg, bgbox=False, precision=0, fontsize=9,
            ax=ax2, cmap='gist_earth'); ax2.set_title('Segmentation');
ps.heatmap(bw_roi_label, bgbox=False, precision=0, fontsize=9,
            ax=ax3, cmap='gist_earth'); ax3.set_title('Labels')
ps.heatmap(gt_labs[75:110:2, 125:150:2], bgbox=False, precision=0, fontsize=9, ax=ax4,
 ↪ cmap='gist_earth'); ax4.set_title('Ground Truth');
```



When we compare the ground truth with the connected component labeling, we see that it fails to identify the cells. This demonstrates the weakness of the labeling algorithm - it is not able to label items that touch. We need something else to handle this use case.

0.12 Watershed: Flowing Downhill

Watershed segmentation is a well-known labeling method for complicated scenes where the items are connected or overlapping. The name comes from the analogy in nature; if you pour a bucket of water in a hilly landscape, it will flow downhill and at the end reach the bottom of the valley. If there is a thunderstorm the water will form small lakes in the landscape, these lakes will continue to grow until they reach the ridge of the hill where they will spill over into the next valley.

The watershed algorithm works in the same manner, where the lakes are the segmented items. However, in the case of segmentation, we imagine building dams at the ridges to prevent spillover. These dams form the item boundaries.

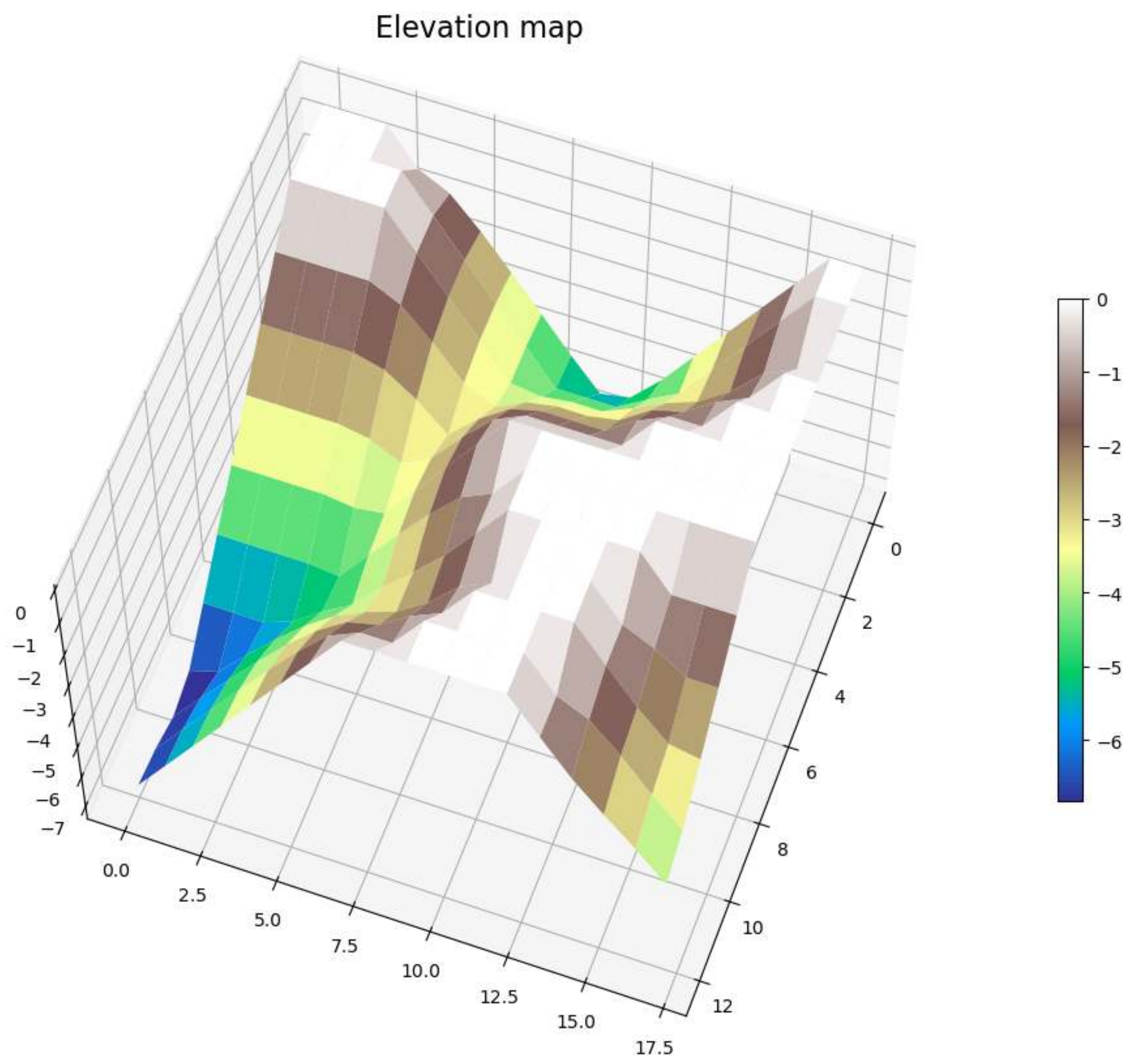
We can imagine watershed as waterflowing down hill into basins. The topology in this case is given by the distance map

```
from scipy.ndimage import distance_transform_edt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(15, 10))
ax = fig.add_subplot(projection='3d')
bw_roi_dmap = distance_transform_edt(bw_roi_seg) # The distance map needed for the␣
 ↪segmentation
```

(continues on next page)

(continued from previous page)

```
# Plot the surface.
t_xx, t_yy = np.meshgrid(np.arange(bw_roi_dmap.shape[1]),np.arange(bw_roi_dmap.
 ↪shape[0]))
surf = ax.plot_surface(t_xx, t_yy,-1*bw_roi_dmap, cmap="terrain",linewidth=0.25,␣
 ↪antialiased=True)

# Customize the z axis.
ax.view_init(60, 20)
# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5); ax.set_title('Elevation map',fontsize=16);
```



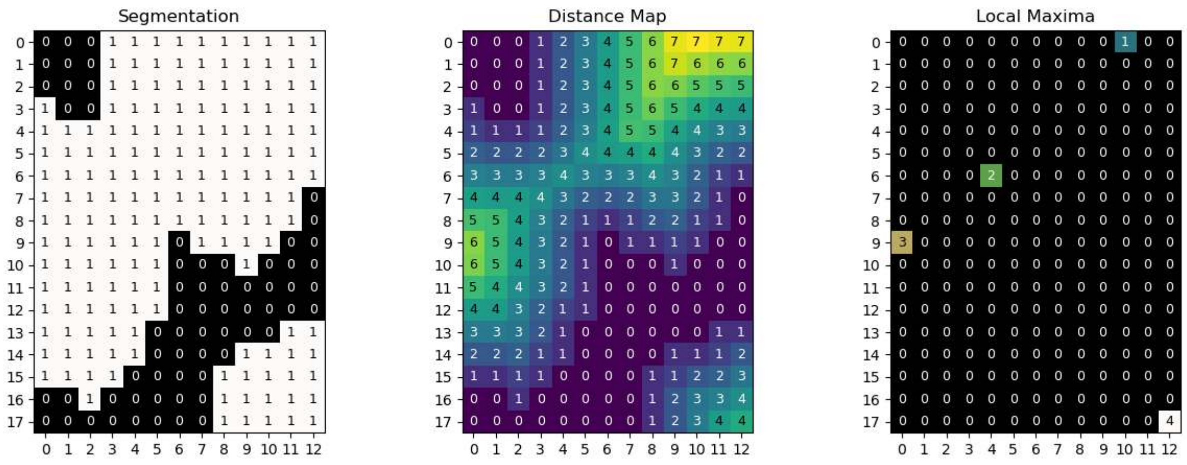
0.12.1 Preparations for watershed segmentation

We need to create a seed image

1. Segment image to bi-level
2. Compute distance map (and flip upside down)
3. Identify local maxima

```python
from skimage.feature import peak_local_max
from scipy.ndimage   import distance_transform_edt

coords = peak_local_max(distance_transform_edt(bw_roi_seg), footprint=np.ones((3, 3)),
                        labels=bw_roi_seg, exclude_border=False)
roi_local_maxi = np.zeros(bw_roi_dmap.shape, dtype=bool)
roi_local_maxi[tuple(coords.T)] = True

# Seed labels
labeled_maxi    = label(roi_local_maxi)
```

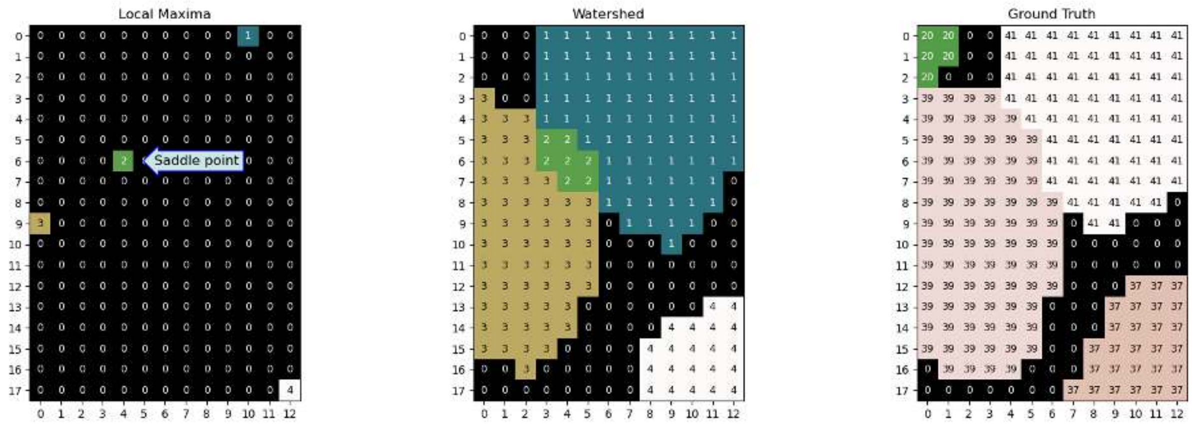
```python
fig, ax= plt.subplots(1, 3, figsize=(15,5))
ps.heatmap(bw_roi_seg,  bgbox=False,  ax=ax[0], precision=0, cmap='gist_earth',
 ↪fontsize=9)
ax[0].set_title('Segmentation')
ps.heatmap(bw_roi_dmap,  bgbox=False, ax=ax[1], precision=0, cmap='viridis',
 ↪fontsize=9)
ax[1].set_title('Distance Map');
ps.heatmap(labeled_maxi, bgbox=False, ax=ax[2], precision=0, cmap='gist_earth',
 ↪fontsize=9)
ax[2].set_title('Local Maxima');
```



0.12.2 Run watershed

```
from skimage.segmentation import watershed
ws_labels = watershed(-bw_roi_dmap, labeled_maxi, mask=bw_roi_seg)
```

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 6), dpi=100)
ps.heatmap(labeled_maxi, bgbox=False,precision=0,fontsize=9,
            ax=ax1,  cmap='gist_earth');
ax1.set_title('Local Maxima')
bbox_props = dict(boxstyle="Larrow", fc=(0.8, 0.9, 0.9), ec="b", lw=1)
t = ax1.text(5.5,6, "Saddle point", ha="left", va="center", rotation=0,
            size=12,
            bbox=bbox_props)
ps.heatmap(ws_labels, bgbox=False,precision=0,fontsize=9,
            ax=ax2,  cmap='gist_earth');
ax2.set_title('Watershed')

ps.heatmap(gt_labs[75:110:2, 125:150:2], bgbox=False,precision=0,fontsize=9, ax=ax3, ↪cmap='gist_earth');
ax3.set_title('Ground Truth');
```



0.12.3 Removing too small elements - Method 1

One of the components (Label=2) is too small!

We can remove it by deleting unwanted seeds.

- seed belonging to the bottom 10 percentile of areas
- rerunning watershed

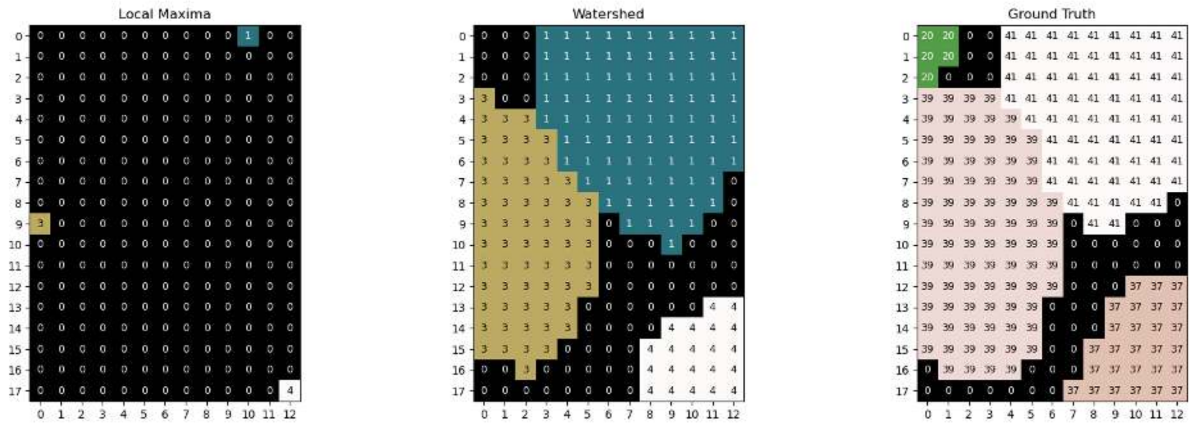
```
label_area_dict = {i: np.sum(ws_labels == i)
                   for i in np.unique(ws_labels[ws_labels > 0])}
clean_label_maxi = labeled_maxi.copy()
area_cutoff = np.percentile(list(label_area_dict.values()), 10)
print('Cutoff at {0:0.2f}'.format(area_cutoff))
for i, k in label_area_dict.items():
    print('Label: ', i, 'Area:', k, 'Keep:', k > area_cutoff)
    if k <= area_cutoff:
        clean_label_maxi[clean_label_maxi == i] = 0
```

```
Cutoff at 11.20
Label:  1 Area: 82 Keep: True
Label:  2 Area: 7 Keep: False
Label:  3 Area: 59 Keep: True
Label:  4 Area: 21 Keep: True
```

Watershed after removing labels

```
ws_labels = watershed(-bw_roi_dmap, clean_label_maxi, mask=bw_roi_seg)
```

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 6), dpi=100)
ps.heatmap(clean_label_maxi, bgbox=False,precision=0,fontsize=9,
            ax=ax1,  cmap='gist_earth');
ax1.set_title('Local Maxima')

ps.heatmap(ws_labels, bgbox=False,precision=0,fontsize=9,
            ax=ax2, cmap='gist_earth');
ax2.set_title('Watershed')

ps.heatmap(gt_labs[75:110:2, 125:150:2], bgbox=False,precision=0,fontsize=9, ax=ax3, ␣
↪cmap='gist_earth');
ax3.set_title('Ground Truth');
```



This result looks much more convincing than the component labeling before. We do, however, still fail to segment one region compared to the ground truth. This is because out thresholding didn't manage to mark the weaker intensity cell as foreground. So, to get this one, we need to revise our cell identification segmentation.

0.12.4 Scaling back up

Now we can perform the operation on the whole image and see how the results look

```
bw_seg_img = opening(bw_img < threshold_otsu(bw_img), disk(3))

bw_dmap  = distance_transform_edt(bw_seg_img)
bw_peak_coords = peak_local_max(bw_dmap.copy(), footprint=np.ones((3, 3)),
                                labels=bw_seg_img, exclude_border=True)
bw_dmap[bw_dmap<0]=0
```

(continues on next page)

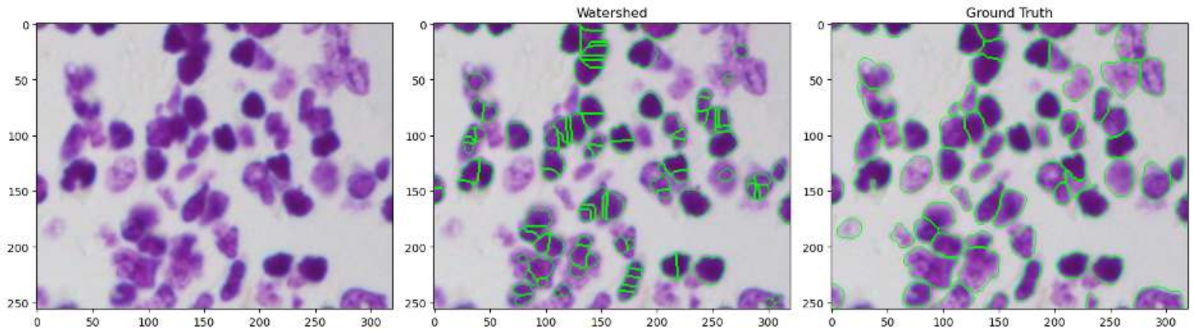
(continued from previous page)

```
bw_peaks = np.zeros_like(bw_dmap)
bw_peaks[tuple(bw_peak_coords.T)]=1
bw_peaks = label(bw_peaks)
ws_labels = watershed(-bw_dmap, bw_peaks, mask=bw_seg_img)

label_area_dict = {i: np.sum(ws_labels == i)
                   for i in np.unique(ws_labels[ws_labels > 0])}

clean_label_maxi = bw_peaks.copy()
lab_areas = list(label_area_dict.values())
area_cutoff = np.percentile(lab_areas, 20)
print('10% cutoff', area_cutoff, '. Removed', np.sum(
    np.array(lab_areas) < area_cutoff), 'components')
for i, k in label_area_dict.items():
    if k <= area_cutoff:
        clean_label_maxi[clean_label_maxi == i] = 0

ws_labels = watershed(-bw_dmap, clean_label_maxi, mask=bw_seg_img)
```

```
10% cutoff 35.0 . Removed 25 components
```

The resulting segmentaction

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 6))
ax1.imshow(rgb_img, cmap='bone')

ax2.imshow(mark_boundaries(label_img=ws_labels,
                           image=rgb_img, color=(0, 1, 0))); ax2.set_title('Watershed
↪')

ax3.imshow(mark_boundaries(label_img=gt_labs, image=rgb_img, color=(0, 1, 0)));
ax3.set_title('Ground Truth');
plt.tight_layout();
fig.savefig('ws_full.png',dpi=300);
```



No fantastic performance :-(

We have:

- many over-segmented cells
- missing cells

Why bad performance?

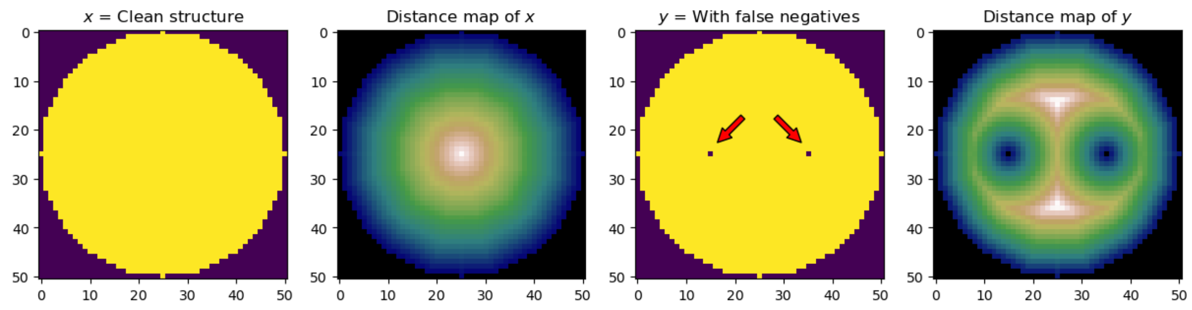
Missing objects

- Too simple thresholding method was used (Otsu)

Over segmentation

- Irregular shapes and misclassified pixels produce too many local maxima in the elevation map

```
import matplotlib.patches as mpatches
fig,ax=plt.subplots(1,4, figsize=(15,4)); ax=ax.ravel()
d=disk(25)
ax[0].imshow(d,interpolation='none')
ax[0].set_title('$x$ = Clean structure')
ax[1].imshow(distance_transform_edt(d), cmap= 'gist_earth' )
ax[1].set_title('Distance map of $x$')
d[25,15]=0; d[25,35]=0
ax[2].imshow(d,interpolation='none')

arrow = mpatches.FancyArrowPatch((22, 17), (16, 23),
                                 mutation_scale=20,fc='red')
ax[2].add_patch(arrow)
arrow = mpatches.FancyArrowPatch((28, 17), (34, 23),
                                 mutation_scale=20,fc='red')
ax[2].add_patch(arrow)
ax[2].set_title('$y$ = With false negatives')

ax[3].imshow(distance_transform_edt(d), cmap= 'gist_earth' );
ax[3].set_title('Distance map of $y$');
```



Solutions

- Find better segmentation approach than only histogram
- Use morphological algorithms like h-max and min-impose Soille, 1999

0.12.5 Removing too small elements - method 2

The over segmentation is a know problem of the watershed segmentation.

A method proposed by Soille, 1999 is to

1. Analyze the distance map to reject low amplitude peaks.
2. Strenghten the local minima.

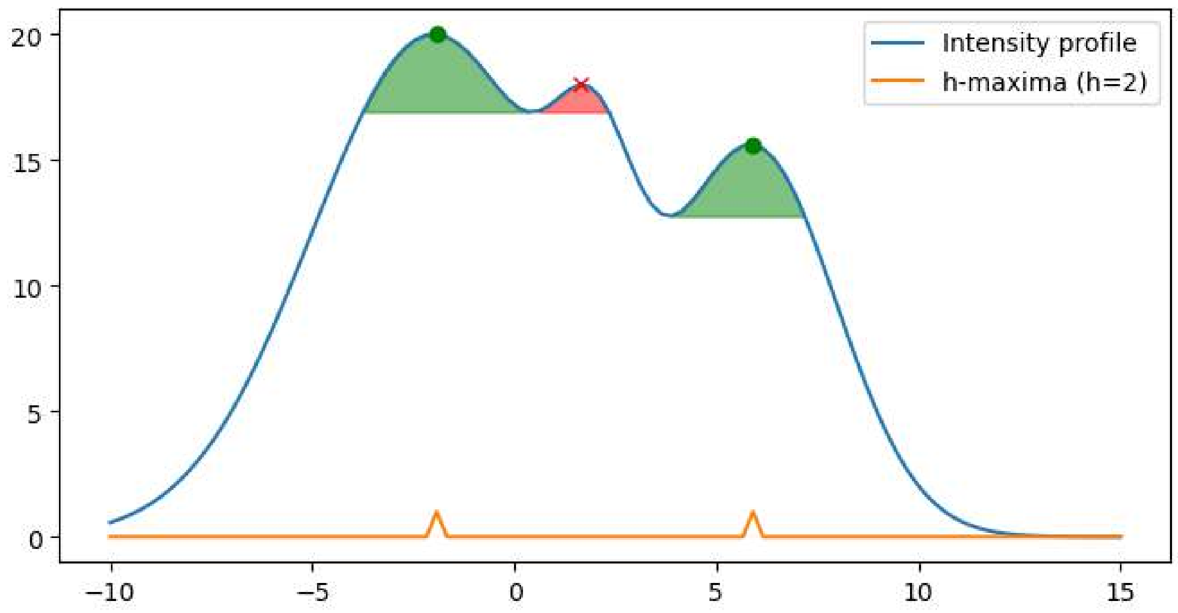
This can be done using methods called reconstruction by dilation $R^{\delta}_{f}(g)$ and erosion $R^{\varepsilon}_{f}(g)$. These are iterative algorithms that reconstructs a mutual minimum or maximum between a mask image and the reconstructed image.

Soille, 1999

Use h-max to find peaks

- The find peaks method finds any peak in the image.
- h-Max add the criterion that the peak must be at least h greylevels higher than the surroundings to produce a marker.

```python
from scipy.signal import find_peaks
from skimage.morphology import h_maxima
def gaussian(x,A,m,s) :
    return A*np.exp(-(x-m)**2/(2*s*s))

x=np.linspace(-10,15,100)
g=gaussian(x,20,-2,3)+gaussian(x,7.5,2,1)+gaussian(x,15,6,2)
peaks,_ = find_peaks(g)
valleys,_ = find_peaks(-g)
# print(peaks,valleys);
fig,ax=plt.subplots(1,figsize=(8,4))
ax.plot(x,g,label='Intensity profile')
ax.fill_between(x[25:42],g[25:42],y2=g[41],color='g',alpha=0.5)
ax.fill_between(x[42:50],g[42:50],y2=g[41],color='r',alpha=0.5)
ax.fill_between(x[55:69],g[55:69],y2=g[55],color='g',alpha=0.5)
ax.plot([x[32],x[63]],[g[32],g[63]],'og')
ax.plot(x[46],g[46],'xr');
# bbox_props = dict(boxstyle="Larrow", fc=(0.8, 0.9, 0.9), ec="b", lw=1)
# t = ax.text(x[23],g[41], "h>2", ha="left", va="top", rotation=-90,
#             size=6,
#             bbox=bbox_props)

# arrow = mpatches.FancyArrowPatch(x[23],g[41], (x[23], g[32]),
#                                  mutation_scale=20,fc='red')
# ax[2].add_patch(arrow)
ax.plot(x,h_maxima(g,h=2),label='h-maxima (h=2)')
ax.legend()
ax.set_yticks([0,5,10,15,20]);
```

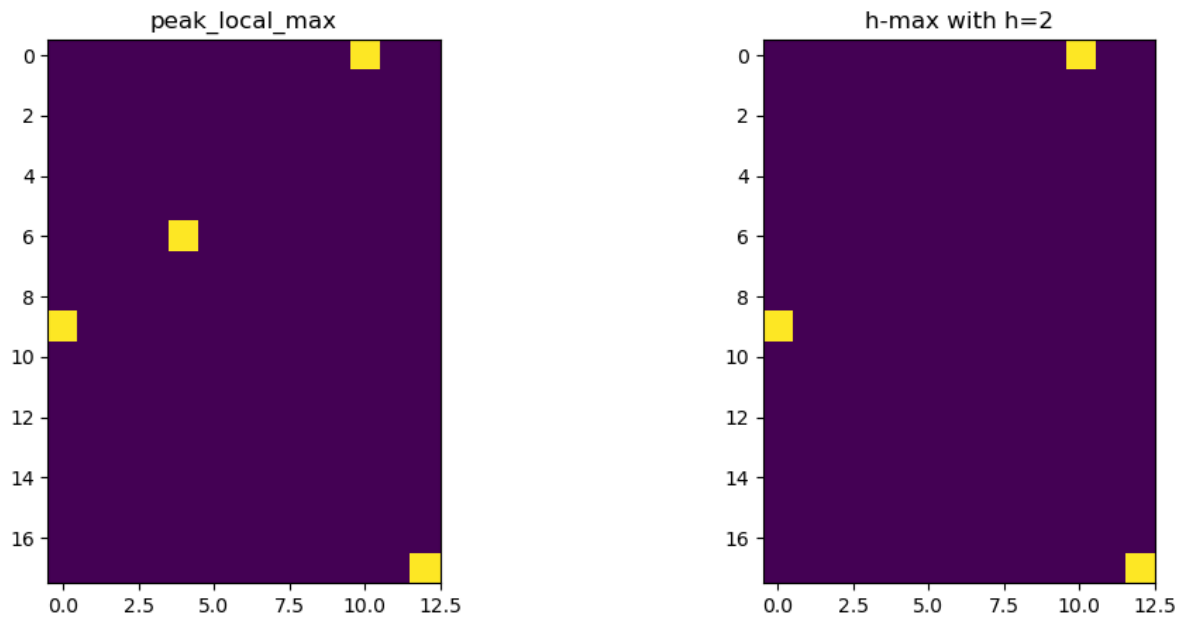


The h-max finds all peaks with a local height of h. The green peaks in the curve above will be kept.

Applying h-max

```python
from skimage.morphology import h_maxima

roi_local_maxi = peak_local_max(bw_roi_dmap.copy(), footprint=np.ones((3, 3)),
                                labels=bw_roi_seg, exclude_border=False)
bw_roi_dmap[bw_roi_dmap<0]=0
bw_peaks = np.zeros_like(bw_roi_dmap)
bw_peaks[tuple(roi_local_maxi.T)]=1

h=2
hmax=h_maxima(bw_roi_dmap,h=h)
```

```python
fig, ax = plt.subplots(1,2,figsize=(12,5))
ax[0].imshow(bw_peaks); ax[0].set_title('peak_local_max')
ax[1].imshow(hmax); ax[1].set_title("h-max with h={}".format(h));
```



Use min-impose to strengthen the peaks

$$R^{\varepsilon}_{(f+1)\hat{}f_m}(f_m)$$

with

$$f_m(p) = \begin{cases} 0 & \text{if } p \text{ belongs to a marker} \\ t_{max} & \text{otherwise} \end{cases}$$

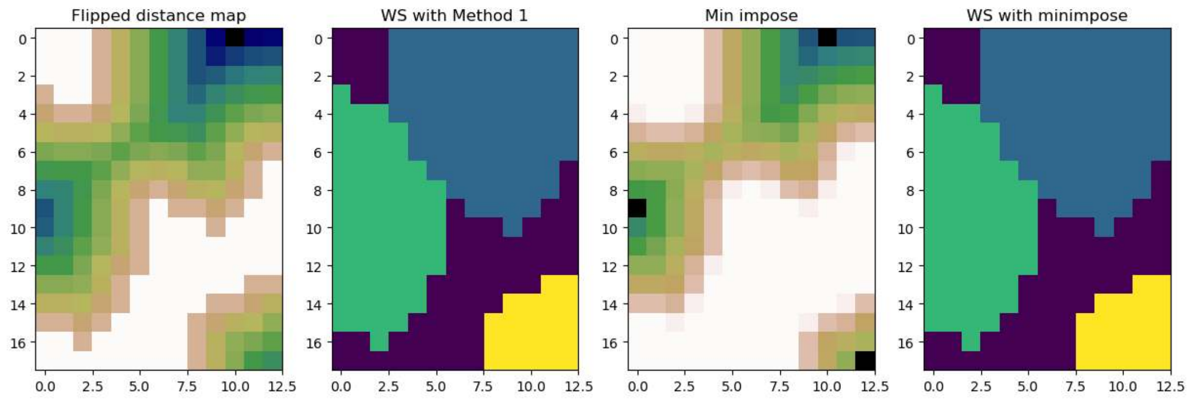
Compare to pulling a wrinkled membrane.

```python
from skimage.morphology import grayreconstruct as gr

def min_impose(dimg,markers) :
    fm=markers.copy()
    fm[markers != 0] = 0
    fm[markers == 0] = dimg.max()
    dimg2 = np.minimum(fm,dimg+1)
    res   = gr.reconstruction(fm,dimg2,method='erosion')

    return res
```

Testing minimpose and h-min - cropped image

```python
d = bw_roi_dmap.max()-bw_roi_dmap

ws0 = watershed(-bw_roi_dmap, label(hmax), mask=bw_roi_seg)
m3=label(hmax)
minimp_h=min_impose(d,hmax)

ws  = watershed(minimp_h,m3,mask=bw_roi_seg)
```

```
fig,ax = plt.subplots(1,4,figsize=(15,5))
ax[0].imshow(d, cmap='gist_earth'); ax[0].set_title('Flipped distance map')
ax[1].imshow(ws0);
ax[1].set_title('WS with Method 1')

ax[2].imshow(minimp_h,cmap='gist_earth');
ax[2].set_title('Min impose')

ax[3].imshow(ws);
ax[3].set_title('WS with minimpose');
```

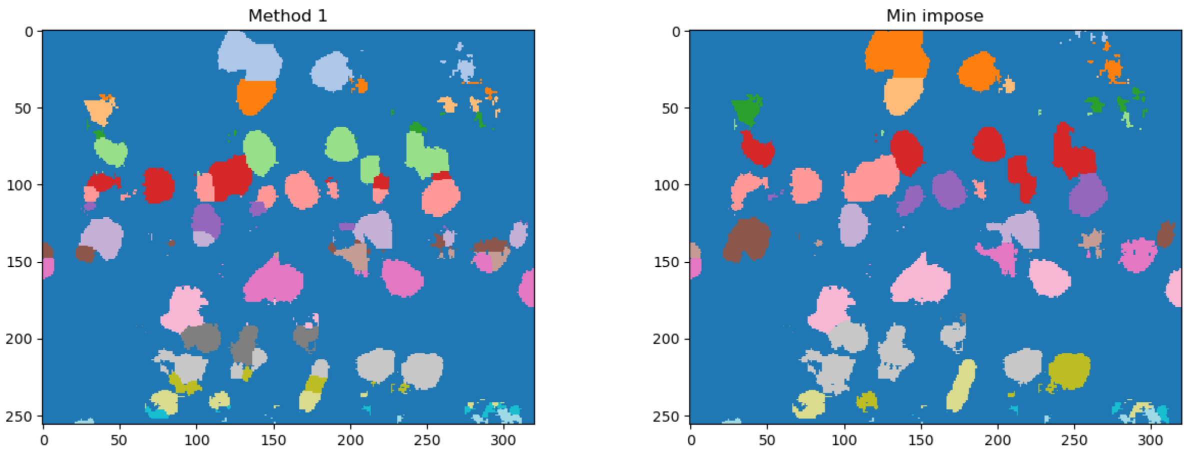


… not much difference here!

Testing minimpose and h-min - full image

```
seg=bw_img < threshold_otsu(bw_img)
dmap = distance_transform_edt(seg)

peak_idx  = peak_local_max(dmap, footprint=np.ones((3, 3)), labels=seg, exclude_
↪border=False)
peak_mask = np.zeros_like(dmap, dtype=bool)
peak_mask[tuple(peak_idx.T)] = True
m0 = label(peak_mask)

ws0 = watershed(-dmap, m0, mask=seg)

h=1
localmax = h_maxima(dmap,h)
rdmap    = dmap.max()-dmap
marker   = label(localmax)
ws1      = watershed(min_impose(rdmap,marker),marker,mask=seg)
```

```
fig,ax=plt.subplots(1,2,figsize=(15,5))
ax[0].imshow(ws0,cmap="tab20",interpolation='None'); ax[0].set_title('Method 1')
ax[1].imshow(ws1,cmap="tab20",interpolation='None'); ax[1].set_title('Min impose');
```

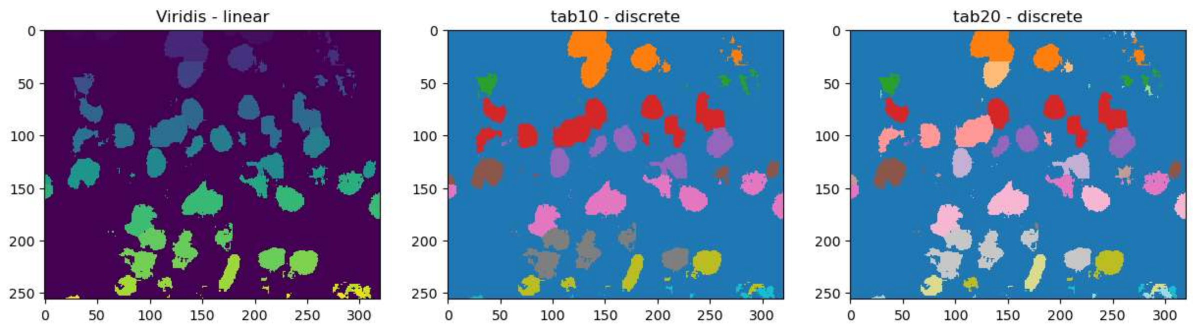


0.13 Color maps for many labels

0.13.1 Standard color maps

It is hard to identify all items in a watershed segmented image

```
fig, ax = plt.subplots(1,3,figsize=(15,4))
ws_labels = ws1
ax[0].imshow(ws_labels,cmap='viridis',interpolation='none');
ax[0].set_title('Viridis - linear')

ax[1].imshow(ws_labels,cmap='tab10',interpolation='none');
ax[1].set_title('tab10 - discrete')

ax[2].imshow(ws_labels,cmap='tab20',interpolation='none');
ax[2].set_title('tab20 - discrete');
```



- Linear colormaps like viridis appear like a gradient.
- Discrete colormaps mostly have too few categories, multiple item have the same color.

0.13.2 Custom colormaps

Create custom colormaps with the same number of entries as labels

Random colors Randomize the RGB channels and orthogonalize the vectors

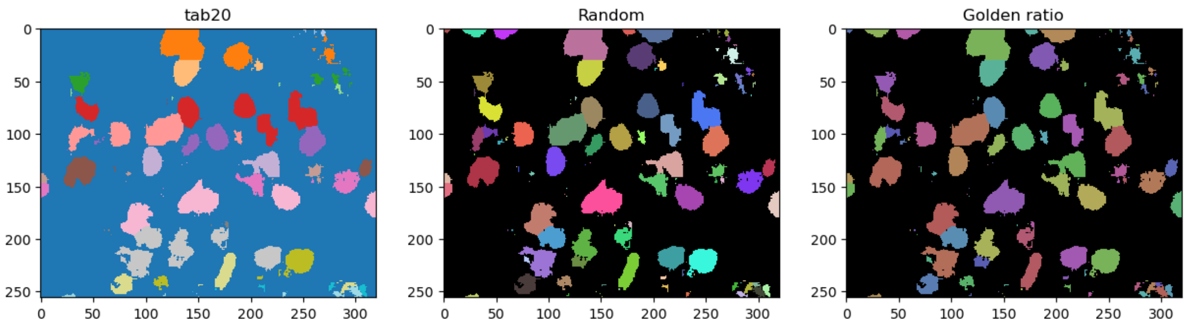
```
from scipy.stats import ortho_group
from matplotlib.colors import ListedColormap

def randomCM(N, low=0.2, high=1.0,seed=42, bg=0) :
    np.random.seed(seed=seed)
    clist=np.random.uniform(low=low,high=high,size=[N,3]);
    m = ortho_group.rvs(dim=3)
    if bg is not None : clist[0,:]=bg;

    rmap = ListedColormap(clist)

    return rmap
```

Golden ratio Use the golden ratio to determine the next hue value

```
def goldenCM(N,increment=1.0,s=0.5,v=0.7,bg=0) :
    phi= 0.5*(np.sqrt(5)-1)

    hsv = np.zeros([N,3]);
    hsv[:, 0] = increment*phi*np.linspace(0,N-1,N)-np.floor(increment*phi*np.
↪linspace(0,N-1,N))
    hsv[:, 1] = s
    hsv[:, 2] = v
    rgb = hsv2rgb(hsv)
    if bg is not None : rgb[0,:]=bg
    cm = ListedColormap(rgb)
    return cm
```

Trying the custom colormaps

```
fig, ax =plt.subplots(1,3,figsize=(15,4))
ws_labels = ws1
ax[0].imshow(ws_labels,cmap='tab20',interpolation='none');    ax[0].set_title('tab20');
ax[1].imshow(ws_labels,cmap=randomCM(ws_labels.max()),interpolation='none'); ax[1].
↪set_title('Random')
ax[2].imshow(ws_labels,cmap=goldenCM(ws_labels.max()),interpolation='none'); ax[2].
↪set_title('Golden ratio');
```

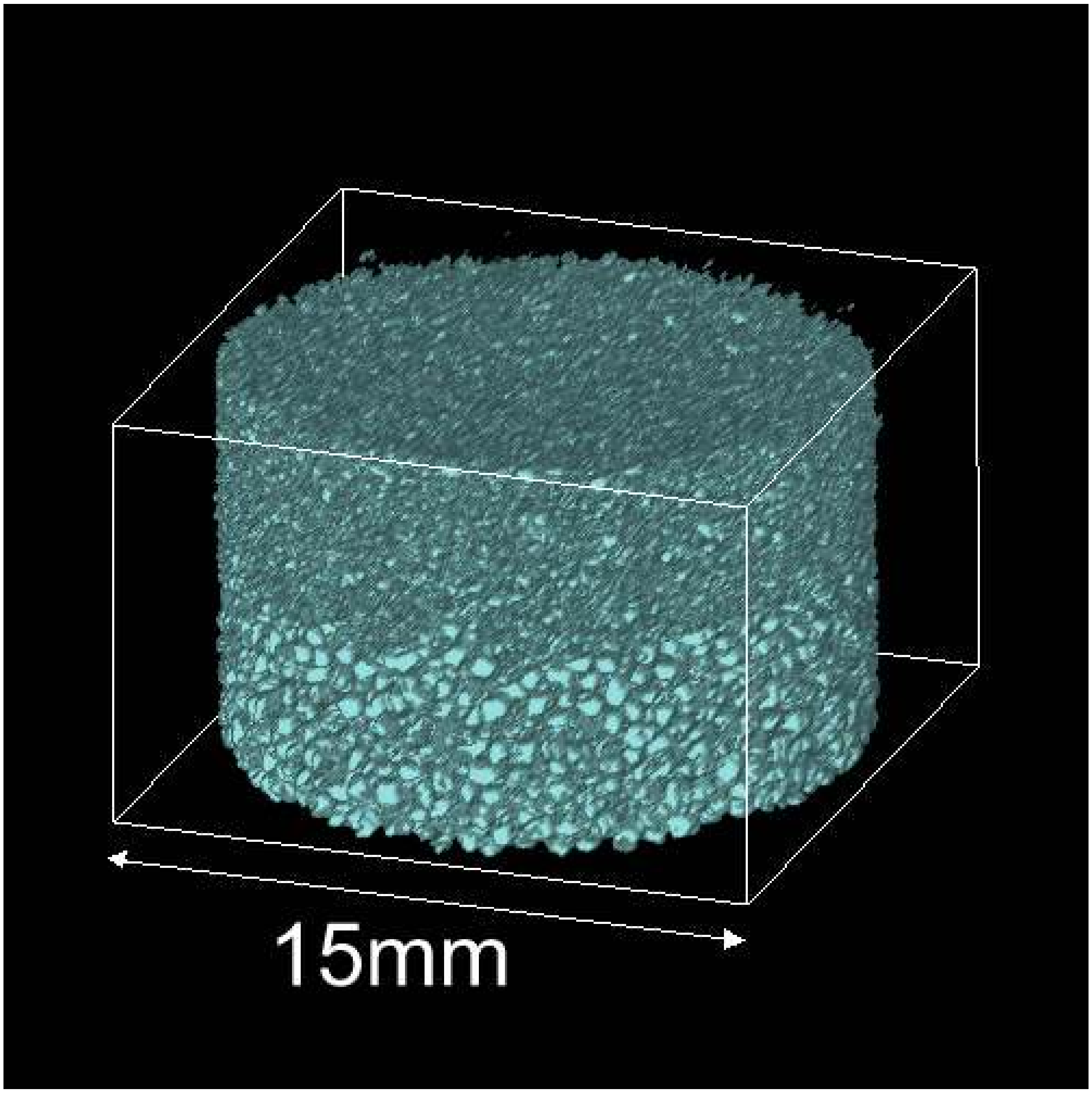


0.14 Example - Separating grain packings

It is very common to scan grain packings in different fields of materials science.

In this example,

- we have a volume with grains from two size fractions (500x500x300 voxels).



- The purpose of this packing is to study the water flow properties at the interface between the fractions.

Our task

- We want to separate the two fractions

Freely from Kaestner et al. 2005

0.14.1 Workflow to separate grain packings

0. Load the data
1. Image preparation
    - Noise reduction
    - Binarization
    - Clean misclassified voxels
2. Label the grains
    - Compute distance map
    - Identify and label grain peaks
    - Use watershed segmentation
3. Categorize the grains
    - Compute region information
    - Remove tiny and huge regions
    - Inspect the region properties
    - k-means clustering
4. Assign fraction class to each grain

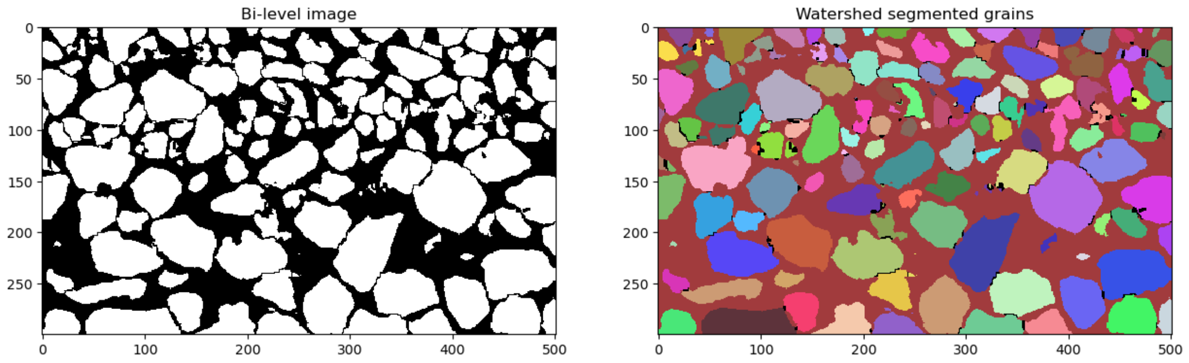
0.14.2 Load and inspect the data

We cheat here and skip the segmentation steps… (the data is zipped)

```
# Define the target file and zip source
def load_with_unzip(fname) :
    target_file = Path(fname)
    zip_source = Path(fname+".zip")

    # Check if the file exists
    if not target_file.exists():
        print(f"{target_file} not found. Extracting from {zip_source}...")

        with zipfile.ZipFile(zip_source, 'r') as zip_ref:
            zip_ref.extractall(target_file.parent)  # Extract to the same directory␣
↪as the target file

    return np.load(target_file)
```

```
bi = load_with_unzip('data/grains.npy')
ws = load_with_unzip('data/ws_grains.npy')
```

```
fig, ax = plt.subplots(1,2,figsize=(15,7))
ax[0].imshow(bi[250,:,:].transpose(),interpolation='none',cmap='gray');
ax[0].set_title('Bi-level image')
ax[1].imshow(ws[250,:,:].transpose(), randomCM(ws[250,:,:].max()-ws[250,:,:].min()),
↪interpolation='none');
ax[1].set_title('Watershed segmented grains');
```



0.14.3 Compute region properties

```
from skimage.measure import regionprops_table
import pandas as pd
import seaborn as sns

rp = pd.DataFrame.from_dict(regionprops_table(ws,properties=['label', 'area',
 ↪'centroid']))
rp.sample(5)
```

```
      label     area  centroid-0  centroid-1  centroid-2
1378   1379    780.0   74.916667    1.607692  179.276923
845     846  33108.0   12.254319  124.889996   81.973783
422     423  11271.0  288.379292  165.538816   39.873303
997     998   7883.0  244.648738  216.303945  100.990993
667     668  12217.0  329.069166   44.555128   57.523124
```

This is a great reduction in amount of data

```
steps=[{"Tomography": 2*1000**3, "Reconstruction": 4*1000**3, "Filtering": 4*1000**3,
 ↪"Thresholding": 1000**3, "Labeling": 2*1000**3, "Regionanalysis": 2000*5*4}]
steps
df = pd.DataFrame(steps)
df
```

```
   Tomography  Reconstruction   Filtering  Thresholding    Labeling  \
0  2000000000      4000000000  4000000000    1000000000  2000000000

   Regionanalysis
0           40000
```

- The image volume was 500x500x300 voxels
- The table has 1680 rows and 5 columns

Prune the table

```
rp.describe()
```

```
               label          area  centroid-0  centroid-1  centroid-2
count   1680.000000  1.680000e+03  1680.000000  1680.000000  1680.000000
mean     878.508929  4.435623e+04   251.003673   239.139842    99.054180
std     1651.137527  6.753043e+05   150.420221   153.932234    81.357521
min        1.000000  1.000000e+00     0.757225     0.000000     0.000000
25%      420.750000  5.785000e+03   118.284959   104.175254    36.202705
50%      840.500000  1.225150e+04   252.919980   232.282127    80.273448
75%     1260.250000  3.230125e+04   379.742401   365.654178   132.446053
max    65535.000000  2.766392e+07   500.000000   500.000000   299.000000
```

We see that some items in the image are either huge or very tiny.

0.14.4 Apply size filters on the table

Let's filter the region property table and extract the gains with relaistic size:

- The huge item is the pore space

```
rp['fg']    = rp['area']<1e7
```

- The small items are grain fractions at the boundary or over segmentations

```
rp['small'] = rp['area']<1000
```

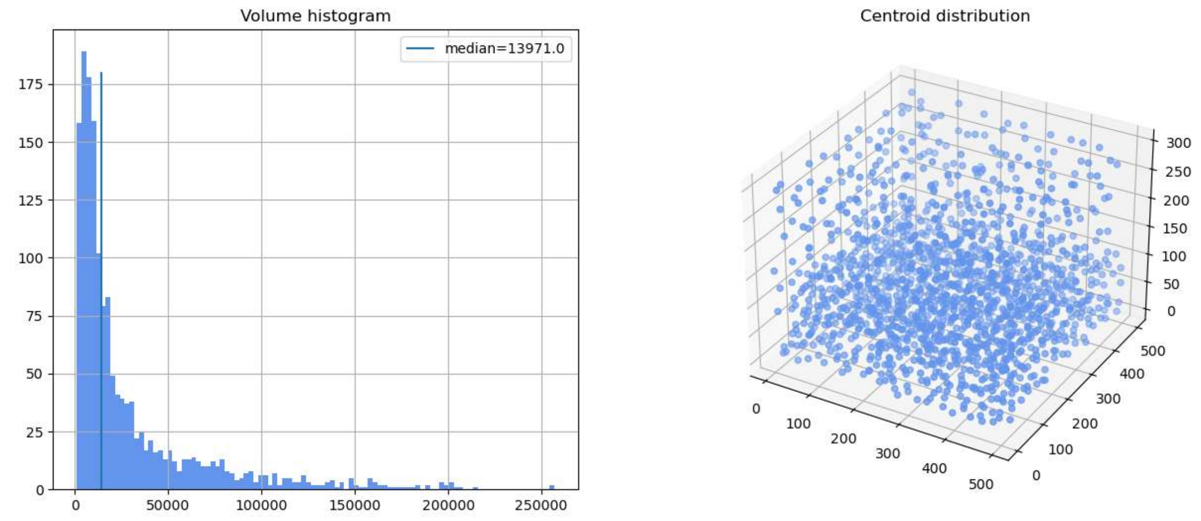
Finally, we make a new table with only realistic grains:

```
grains = rp[(rp['fg']==True) & (rp['small']==False)].copy()

grains.describe()
```

```
             label          area   centroid-0   centroid-1   centroid-2
count  1578.000000    1578.000000  1578.000000  1578.000000  1578.000000
mean    847.536122   29671.088086   248.952285   239.968275   100.152262
std     477.916546   38256.916013   149.588070   148.950250    80.473061
min       4.000000    1003.000000     1.279469     1.180109     1.675439
25%     438.250000    6796.250000   117.989229   110.726740    37.902962
50%     849.500000   13971.000000   251.955865   233.461264    81.492341
75%    1260.250000   35687.750000   377.036584   363.307207   133.888488
max    1679.000000  256746.000000   498.189711   498.680690   296.459704
```

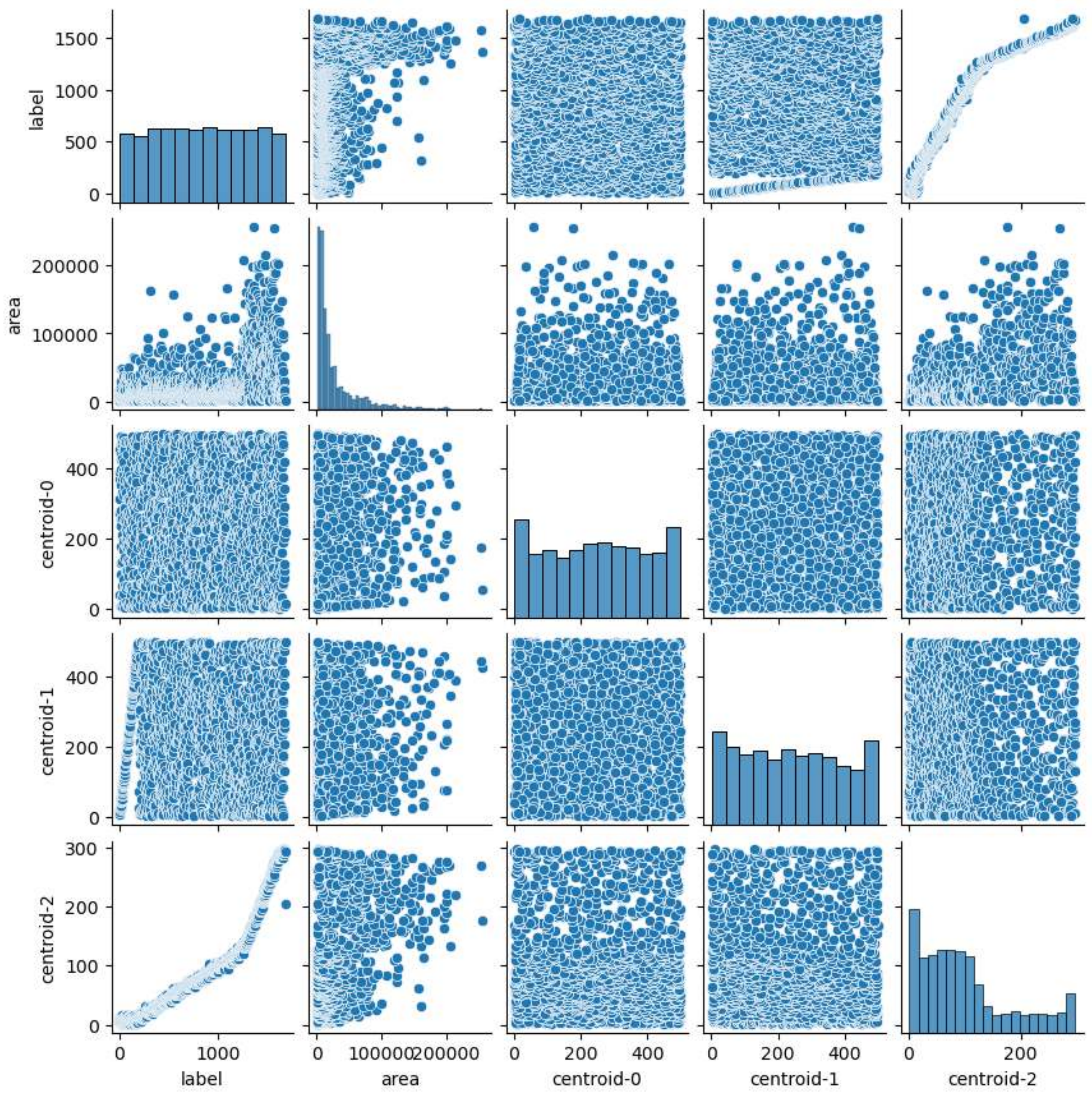
Visualize the pruned table

```
from mpl_toolkits.mplot3d.axes3d import Axes3D
fig = plt.figure(figsize=(15,6))
ax = fig.add_subplot(1, 2, 1)
grains['area'].hist(bins=100,ax=ax, color='cornflowerblue'); ax.set_title('Volume␣
↪histogram')
md=grains['area'].median();
ax.vlines([md],ymin=0,ymax=180,label='median={}'.format(md));
ax.legend()
ax3d = fig.add_subplot(1, 2, 2, projection='3d')
ax3d.scatter(grains['centroid-0'],grains['centroid-1'],grains['centroid-2'], color=
↪'cornflowerblue');
ax3d.set_title('Centroid distribution');
```



A pair plot of the properties

```
sns.pairplot(grains, vars=['label','area','centroid-0','centroid-1','centroid-2'],
↪height=1.75);
```

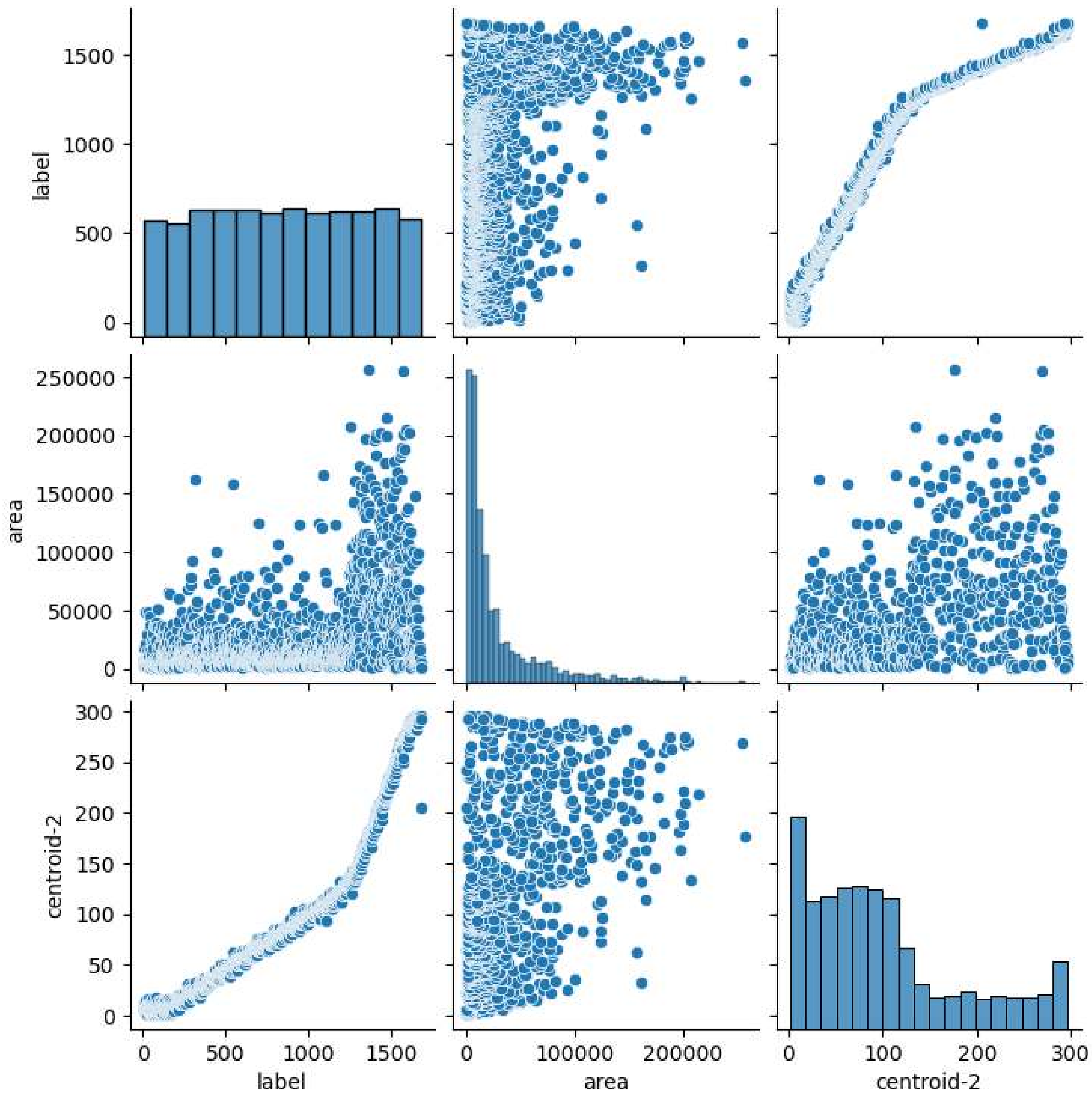


We can see clear property pairs that provide information to support the classification:

- The item label vs area or centroid-2.
- The item area vs centroid-2

Too much information - A closer look at the relevant pairs

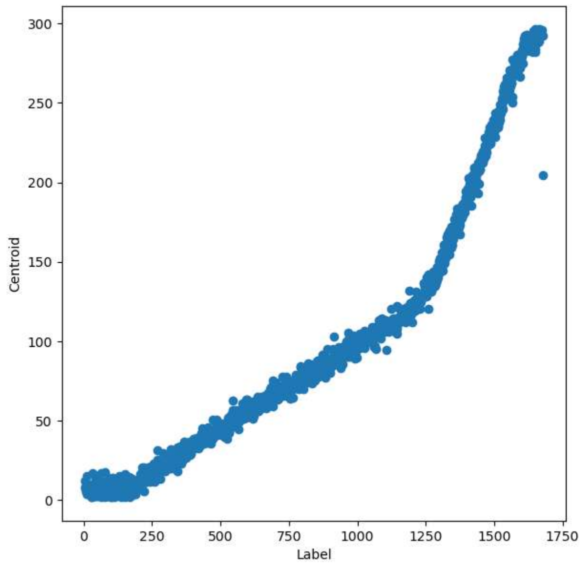
```
sns.pairplot(grains, vars=['label','area','centroid-2']);
```

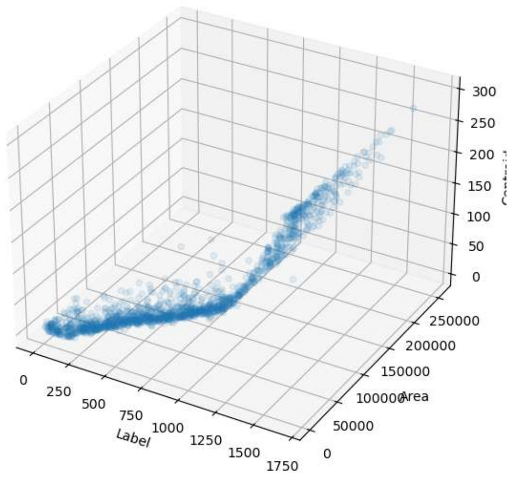


We see here that there is a clear connection between the label and area and centroid. This does normally not hold true, but is rather a coincidence because of the way the labelling algorithm is implemented. It increases the label numbers with the linear scan over the coordinates. Still, in our case we can used this to our advantage as it clearly helps us to find the cut line between the two groups.

Looking for clusters

```
fig = plt.figure(figsize=(15,7))
axlc = fig.add_subplot(1,2,1)
axlc.scatter(grains['label'], grains['centroid-2'])
axlc.set_ylabel('Centroid'); axlc.set_xlabel('Label')
ax3d = fig.add_subplot(1, 2, 2, projection='3d')
ax3d.scatter(grains['label'], grains['area'], grains['centroid-2'], alpha=0.1);
ax3d.set_xlabel('Label'); ax3d.set_ylabel('Area');  ax3d.set_zlabel('Centroid');
```





These are logical pairings - the grains fractions are

- packed in the vertical direction
- the watershed labels run in the vertical direction.

K-Means to cluster the grains

```
from sklearn.cluster import KMeans

kmeans= KMeans(2)
x=kmeans.fit_predict(grains[['label','area','centroid-2']].values)
grains['group']=x.tolist()
grains.sample(5,random_state=30)
```

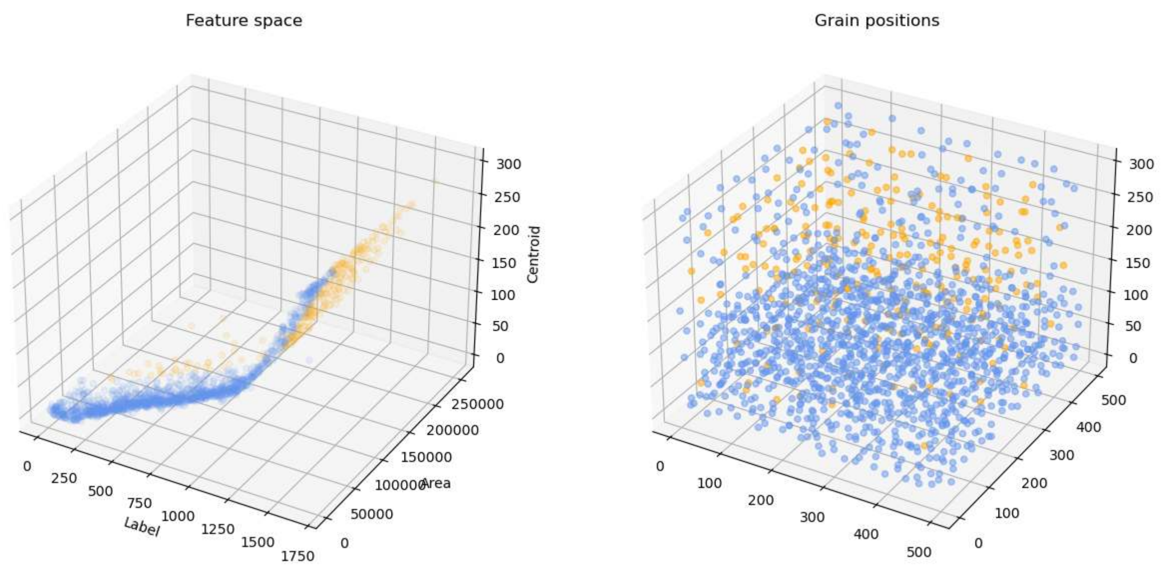
```
      label      area  centroid-0  centroid-1  centroid-2    fg  small  group
852     853    6719.0  255.218634  494.421789   84.453788  True  False      0
444     445  100690.0  435.006982   60.617301   36.261237  True  False      1
538     539    4143.0  120.373401  395.345643   53.542119  True  False      0
1340   1341  155857.0  447.615680  232.692969  162.332625  True  False      1
1605   1606   27592.0  108.840533  205.322014  287.128298  True  False      0
```

```
# Plotting
fig = plt.figure(figsize=(15,7))
```

(continues on next page)

(continued from previous page)

```
ax3d = fig.add_subplot(1, 2, 1, projection='3d')
axpos = fig.add_subplot(1, 2, 2, projection='3d')
c = ['cornflowerblue','orange', 'red']
for idx in range(3) :
    group = grains[grains['group']==idx]
    ax3d.scatter(group['label'], group['area'], group['centroid-2'], alpha=0.1,␣
↪color=c[idx]);
    axpos.scatter(group['centroid-0'], group['centroid-1'], group['centroid-2'],␣
↪alpha=0.5, color=c[idx])
ax3d.set_title('Feature space'); axpos.set_title('Grain positions');
ax3d.set_xlabel('Label'); ax3d.set_ylabel('Area');  ax3d.set_zlabel('Centroid');
```



This is not really what we wanted!

Looking closer at the ranges of each feature we see that the Area more than 1000-fold greater.

We need to scale the features

Try k-means to cluster the grains - normalized features

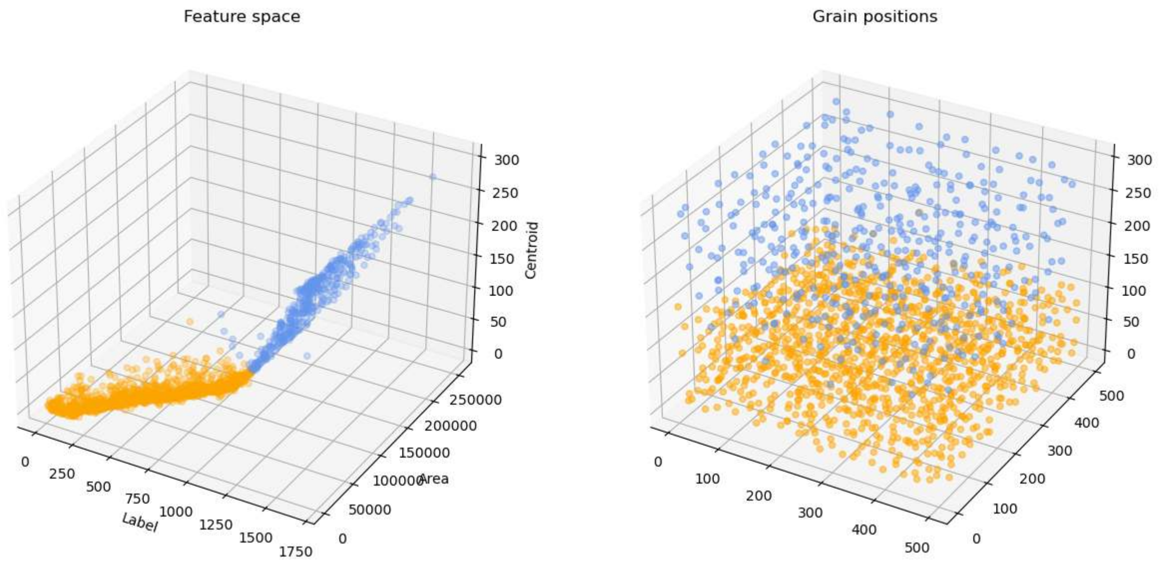
With each vector v \$$v_{normalized} = \frac{v-mean(v)}{std(v)}$\$

This kind of statistical normalization is very common in machine lerning applications. The models are often biased towards the variable with the greatest variance.

All variables have values in the same range after normalization and thus also equally treated.

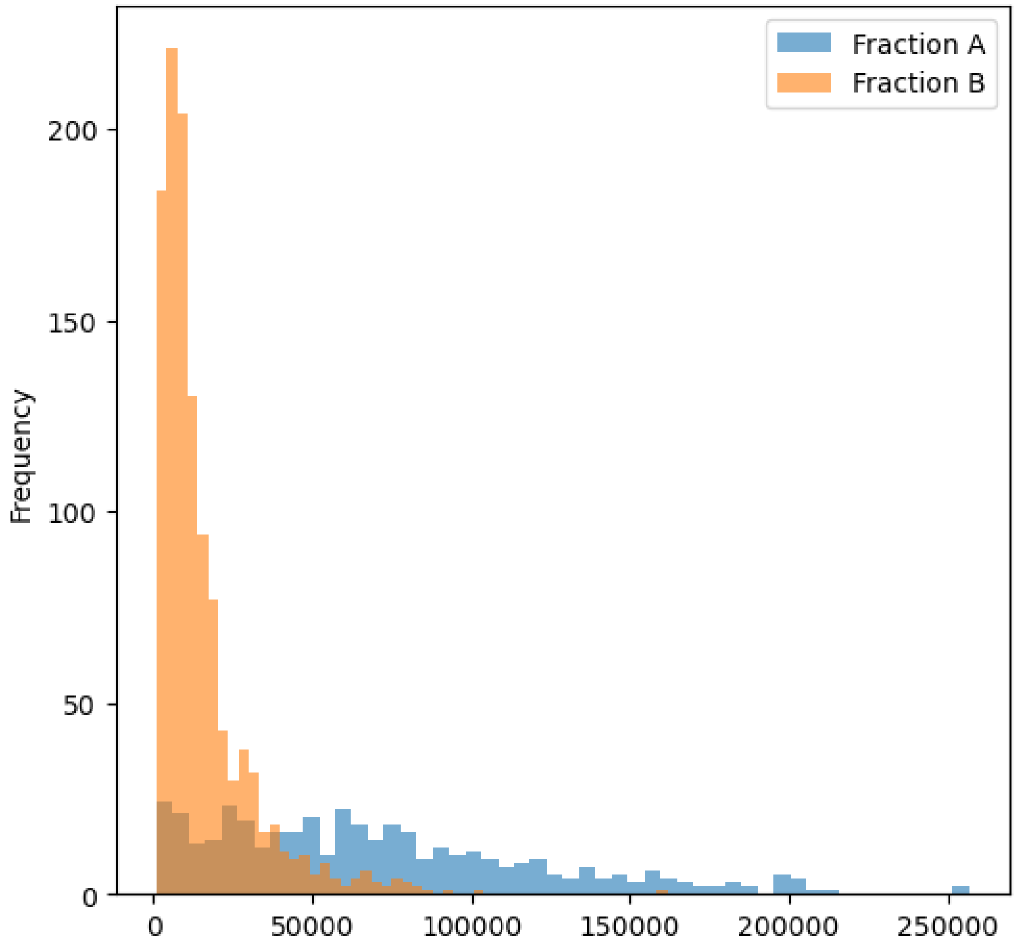
```
grains['nlabel'] = (grains['label']-grains['label'].mean())/grains['label'].std()
grains['narea']  = (grains['area']-grains['area'].mean())/grains['area'].std()
grains['nz']     = (grains['centroid-2']-grains['centroid-2'].mean())/grains[
↪'centroid-2'].std()

kmeans= KMeans(2)
x=kmeans.fit_predict(grains[['nlabel','narea','nz']].values)
grains['group']=x
```

```
# Plotting
fig = plt.figure(figsize=(15,7))
ax3d = fig.add_subplot(1, 2, 1, projection='3d')
axpos = fig.add_subplot(1, 2, 2, projection='3d')
c = ['cornflowerblue','orange', 'red']
for idx in range(3) :
    group = grains[grains['group']==idx]
    ax3d.scatter(group['label'], group['area'], group['centroid-2'], alpha=0.3,␣
↪color=c[idx]);
    axpos.scatter(group['centroid-0'], group['centroid-1'], group['centroid-2'],␣
↪alpha=0.5, color=c[idx])
ax3d.set_title('Feature space'); axpos.set_title('Grain positions');
ax3d.set_xlabel('Label'); ax3d.set_ylabel('Area');  ax3d.set_zlabel('Centroid');
```



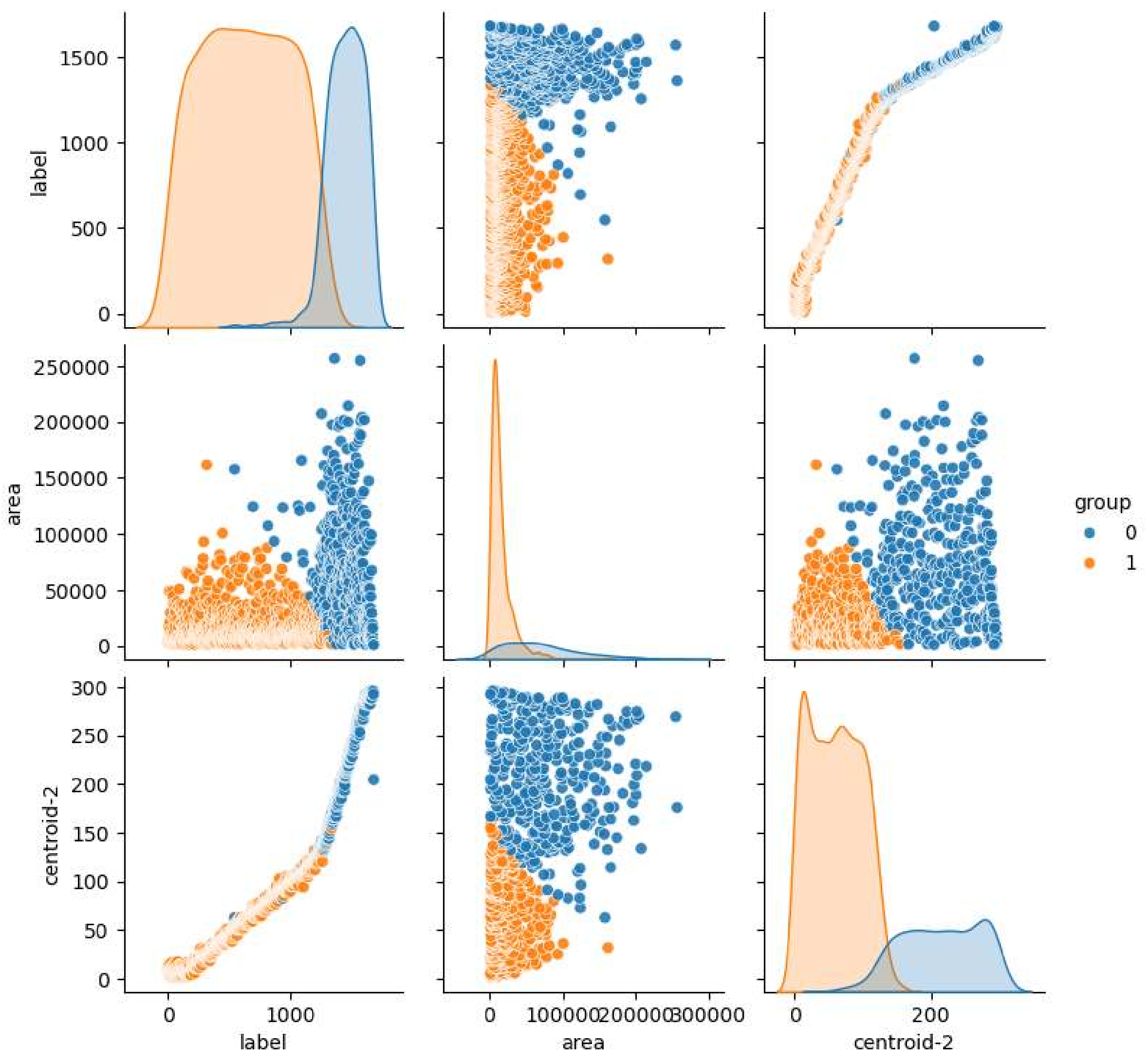
Looking at the grains size histograms

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

grains[grains['group']==0]['area'].plot.hist(ax=ax, bins=50, label='Fraction A',
↪alpha=0.6)
grains[grains['group']==1]['area'].plot.hist(ax=ax, bins=50, label='Fraction B',
↪alpha=0.6)
ax.legend();
```



Return to the pair plot

```
sns.pairplot(grains, vars=['label','area','centroid-2'],hue='group',plot_kws={'alpha
↪':0.9} );
```



Assign classes back to the image

Finally, we come to the last step assigning group labels to the grains.

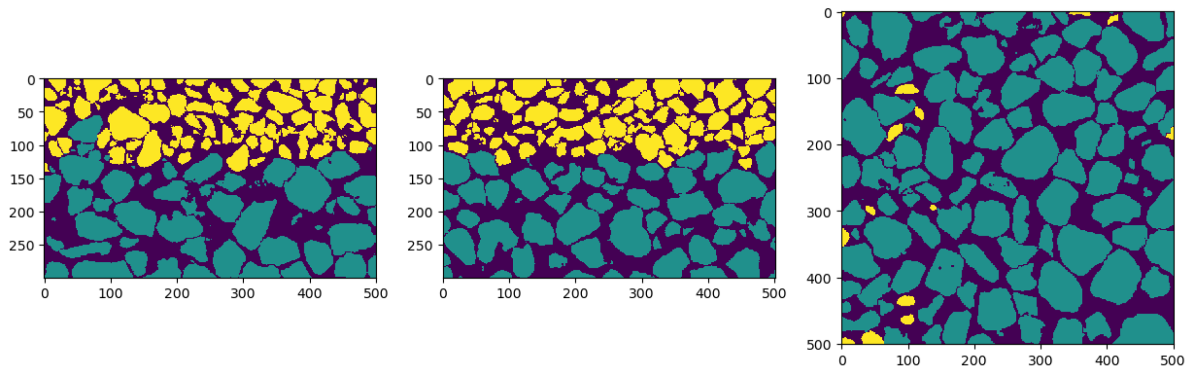
```
from skimage.morphology import reconstruction

def assignGroups(img, itemdf) :
    seeds = np.zeros_like(bi).astype('float')
    for i,row in itemdf.iterrows() :
        seeds[int(np.floor(row['centroid-0'])), int(np.floor(row['centroid-1'])),␣
↪int(np.floor(row['centroid-2']))]=2+row['group']

    a=img*seeds.max()+1.0
    seeds[a<seeds] = 0
    glbl = reconstruction(seeds,a,method='dilation')-1

    return glbl
```

You could also use the `skimage.segmentation.flood_fill` instead of the reconstruction algorithm. That would require you to implement a loop over the seed values.

Apply the label assignment to the grains

```
glbl = assignGroups(bi,grains);
```

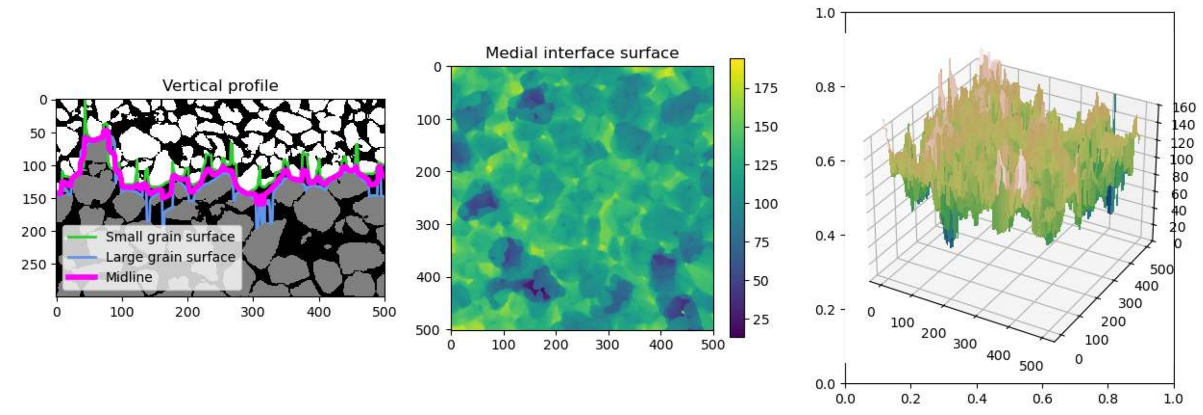
```
fig,ax= plt.subplots(1,3,figsize=(15,5))
ax[0].imshow(glbl[250,:,:].transpose(), interpolation='none')
ax[1].imshow(glbl[:,250,:].transpose(), interpolation='none')
ax[2].imshow(glbl[:,:,150], interpolation='none');
```



The interface surface

```
import skimage.filters as flt
bottomB = glbl.shape[2]-np.argmax(glbl[:,:,::-1]==2,axis=2)
topA    = np.argmax(glbl==1,axis=2)

mAB = flt.median(0.5*(topA+bottomB),footprint=np.ones([7,7]))
```

```
fig,ax = plt.subplots(1,3,figsize=(15,5))

ax[0].imshow(glbl[250,:,:].transpose(), interpolation='none',cmap='gray')
ax[0].plot(bottomB[250,:],color='limegreen',lw=2,label='Small grain surface')
ax[0].plot(topA[250,:],color='cornflowerblue',lw=2,label='Large grain surface')
ax[0].plot(mAB[250,:],color='magenta',lw=4,label='Midline')
ax[0].set_title('Vertical profile')
ax[0].legend()
im=ax[1].imshow(mAB);
ax[1].set_title('Medial interface surface')
fig.colorbar(im,ax=ax[1],shrink=0.75);
ax1 = plt.subplot(1,3,3,projection='3d')
xx,yy = np.meshgrid(np.arange(501),np.arange(501))
ax1.plot_surface(xx,yy,mAB,cmap='gist_earth',linewidth=0, antialiased=False);
ax1.set_zlim(0, 160);
```



0.15 Summary

0.15.1 Skeletons

- Minimal structure of an object
- Object topology is preserved

0.15.2 Watershed segmentation

- Labels touching item
- May oversegment
- Frequently used algorithm