EX 3-2

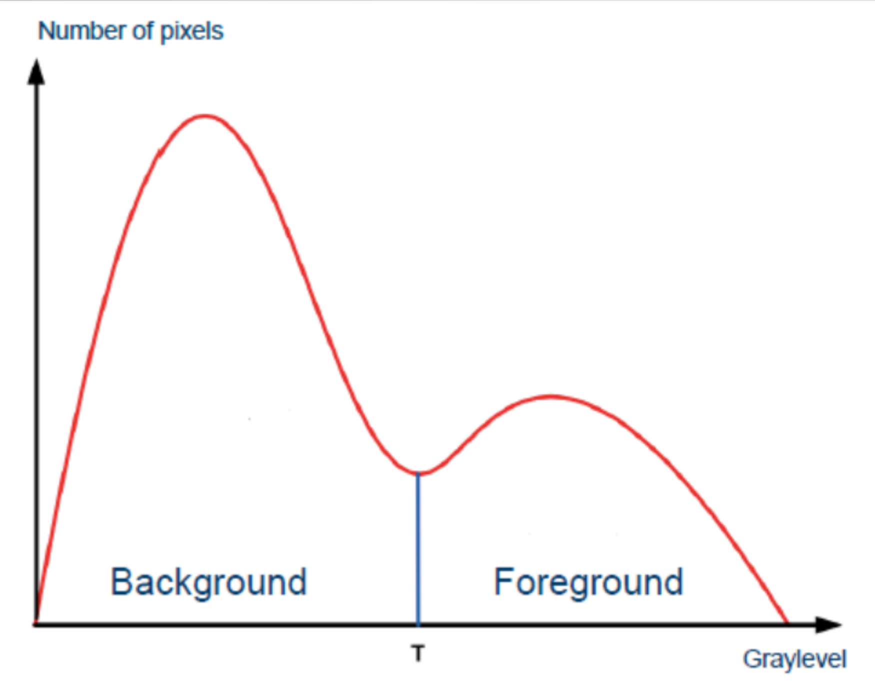
SEGMENTATION

SEGMENTATION

- Thresholding
- Region growing
- K-means
- Gaussian Mixture Model

THRESHOLDING

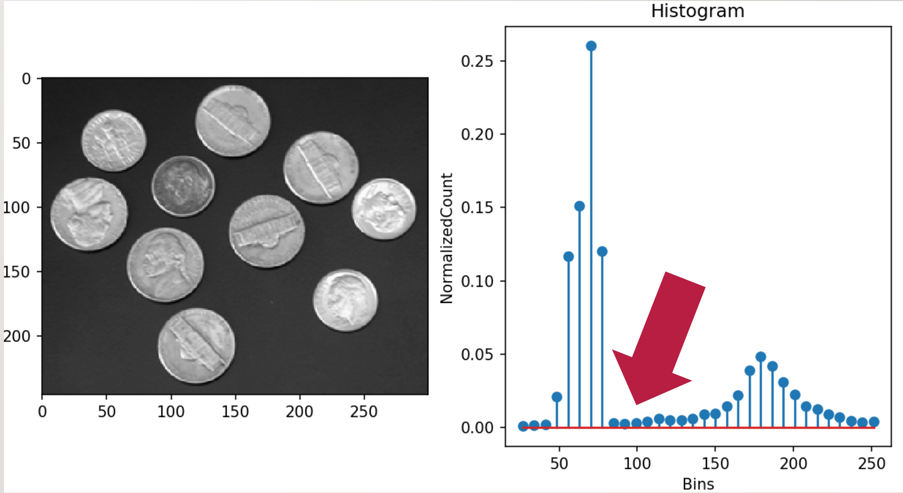
- Binarizing



THRESHOLDING

- Histogram

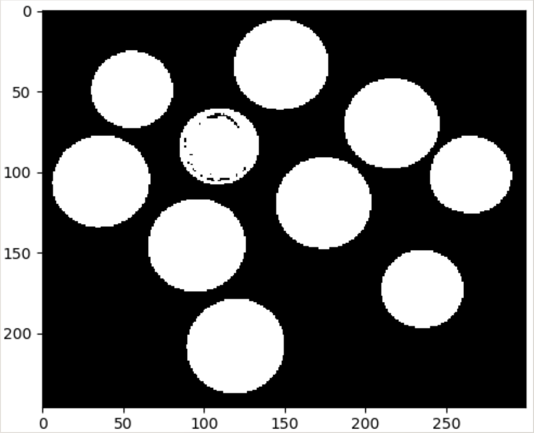
```python
import matplotlib.pyplot as plt
from skimage import io
import numpy


# Load image file
fpath = 'F:/MIPL/SDS/0.excercise/3/ex3-2_segmentation/'
image = io.imread(fpath + 'coins.png')

# Histogram
histY, binEdges = numpy.histogram(image, bins=32)
histY = histY/histY.sum()
histX = (binEdges[1:33] + binEdges[0:32])/2

# Plot
plt.figure(figsize=(10, 5), dpi=150)
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.subplot(1, 2, 2)
plt.stem(histX, histY)
plt.xlabel('Bins')
plt.ylabel('NormalizedCount')
plt.title('Histogram')

plt.show()
```



THRESHOLDING

- Arbitrary set the threshold value = 100

```
# Threshold
image = image > 100

# Plot
plt.figure()
plt.imshow(image, cmap='gray')

plt.show()
```



THRESHOLDING

- Otsu's method
  - Select a threshold that maximizes the between-class variance
    = exhaustively search for the threshold that minimizes the intra-class variance
  - $\sigma_b^2(t) = \omega_0(t)\omega_1(t)[\mu_0(t) - \mu_1(t)]^2$
    , where
    $\omega_0(t) = \sum_{i=0}^{t-1} p(i)$
    $\omega_1(t) = \sum_{i=t}^{nBin-1} p(i)$
    $\mu_0(t) = \frac{\sum_{i=0}^{t-1} ip(i)}{\omega_0(t)}$
    $\mu_1(t) = \frac{\sum_{i=t}^{nBin-1} ip(i)}{\omega_1(t)}$
    $p_i = n_i/N$ (N: total number of pixels)

THRESHOLDING

```python
import matplotlib.pyplot as plt
from skimage import io, filters
import numpy


# Load image file
fpath = 'F:/MIPL/SDS/0.excercise/3/ex3-2_segmentation/'
image = io.imread(fpath + 'coins.png')

# Histogram
histY, binEdges = numpy.histogram(image, range=(0, 255), bins=256)
histY = histY/histY.sum()

# Otsu's Threshold
sigmaBsquared = numpy.zeros((256, 1))
t = 0
while t <= 255:
    w_0 = histY[0:t+1].sum()
    w_1 = histY[t:256].sum()

    if w_0 != 0 and w_1 != 0:
        m_0 = numpy.multiply(numpy.linspace(0, t, t+1), histY[0:t+1]).sum()/w_0
        m_1 = numpy.multiply(numpy.linspace(t+1, 255, 255-t), histY[t+1:256]).sum()/w_1
        sigmaBsquared[t] = w_0 * w_1 * pow(m_0 - m_1, 2)

    t = t + 1

OstuTh = sigmaBsquared.argmax() + 1
imageThresholded = image >= OstuTh

# Plot
plt.figure(figsize=(10, 5), dpi=150)
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.colorbar()
plt.subplot(1, 2, 2)
plt.imshow(imageThresholded, cmap='gray')
plt.title('Ostu Threshold = ' + str(OstuTh))

plt.show()
```

$$\sigma_b^2(t) = \omega_0(t)\omega_1(t)[\mu_0(t) - \mu_1(t)]^2$$

, where

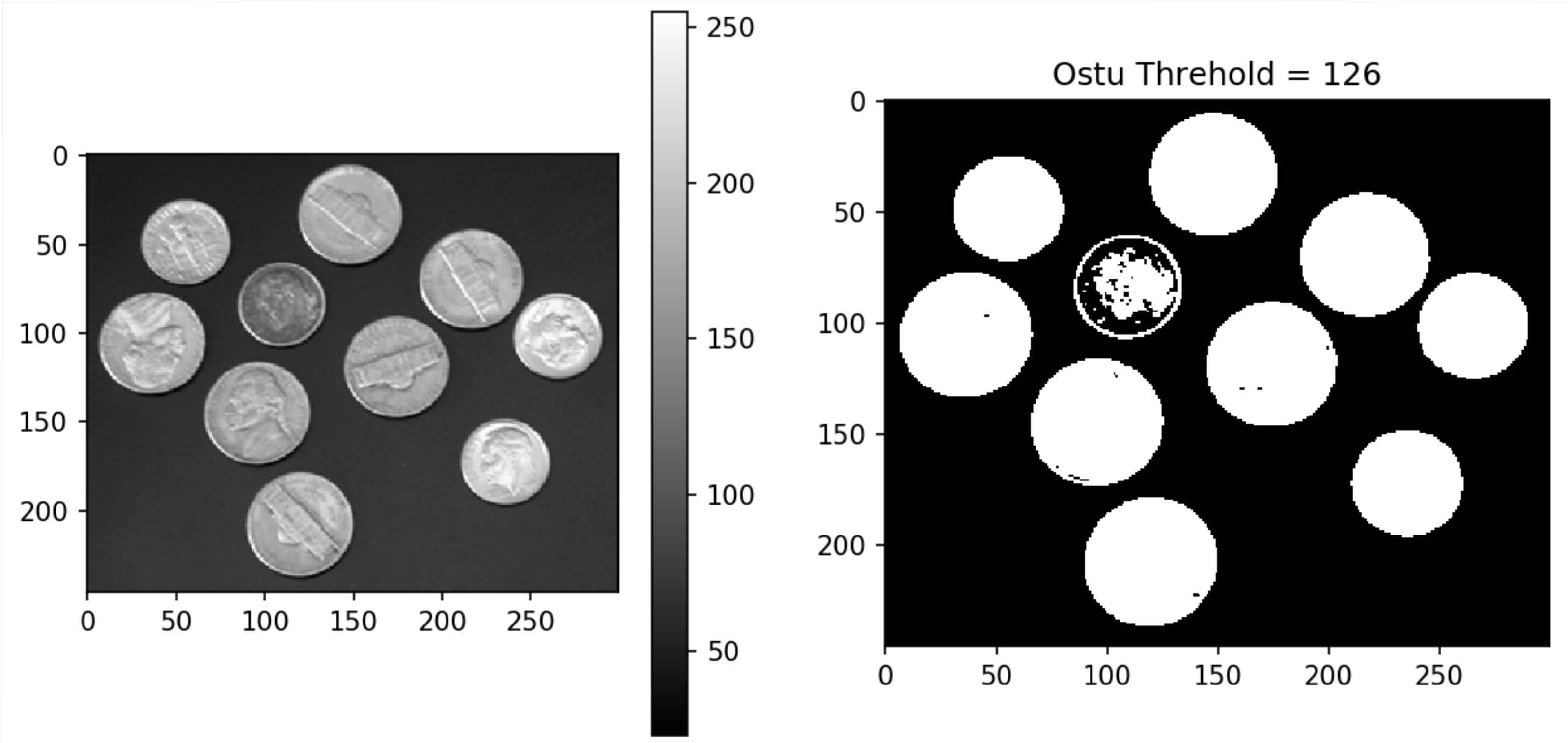
$$\omega_0(t) = \sum_{i=0}^{t-1} p(i)$$

$$\omega_1(t) = \sum_{i=t}^{nBin-1} p(i)$$

$$\mu_0(t) = \frac{\sum_{i=0}^{t-1} ip(i)}{\omega_0(t)}$$

$$\mu_1(t) = \frac{\sum_{i=t}^{nBin-1} ip(i)}{\omega_1(t)}$$

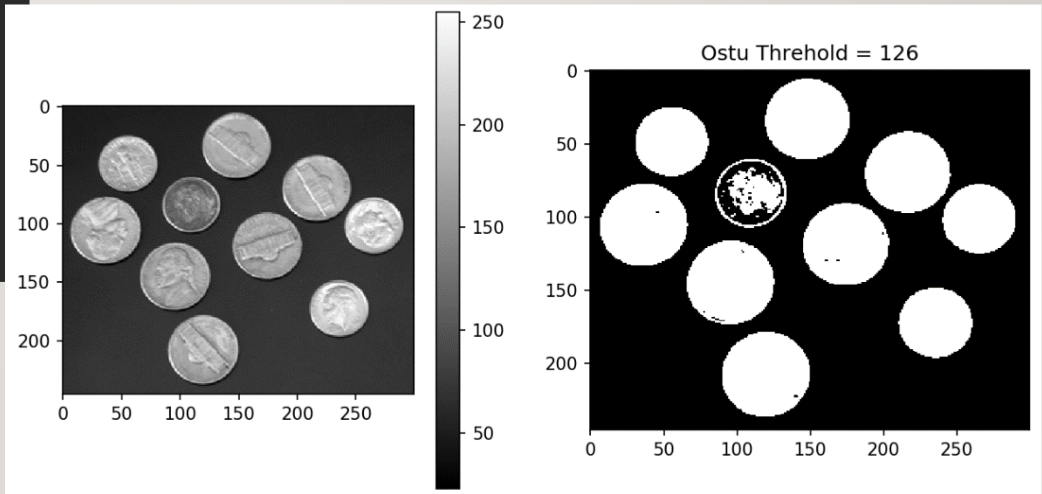
THRESHOLDING



THRESHOLDING

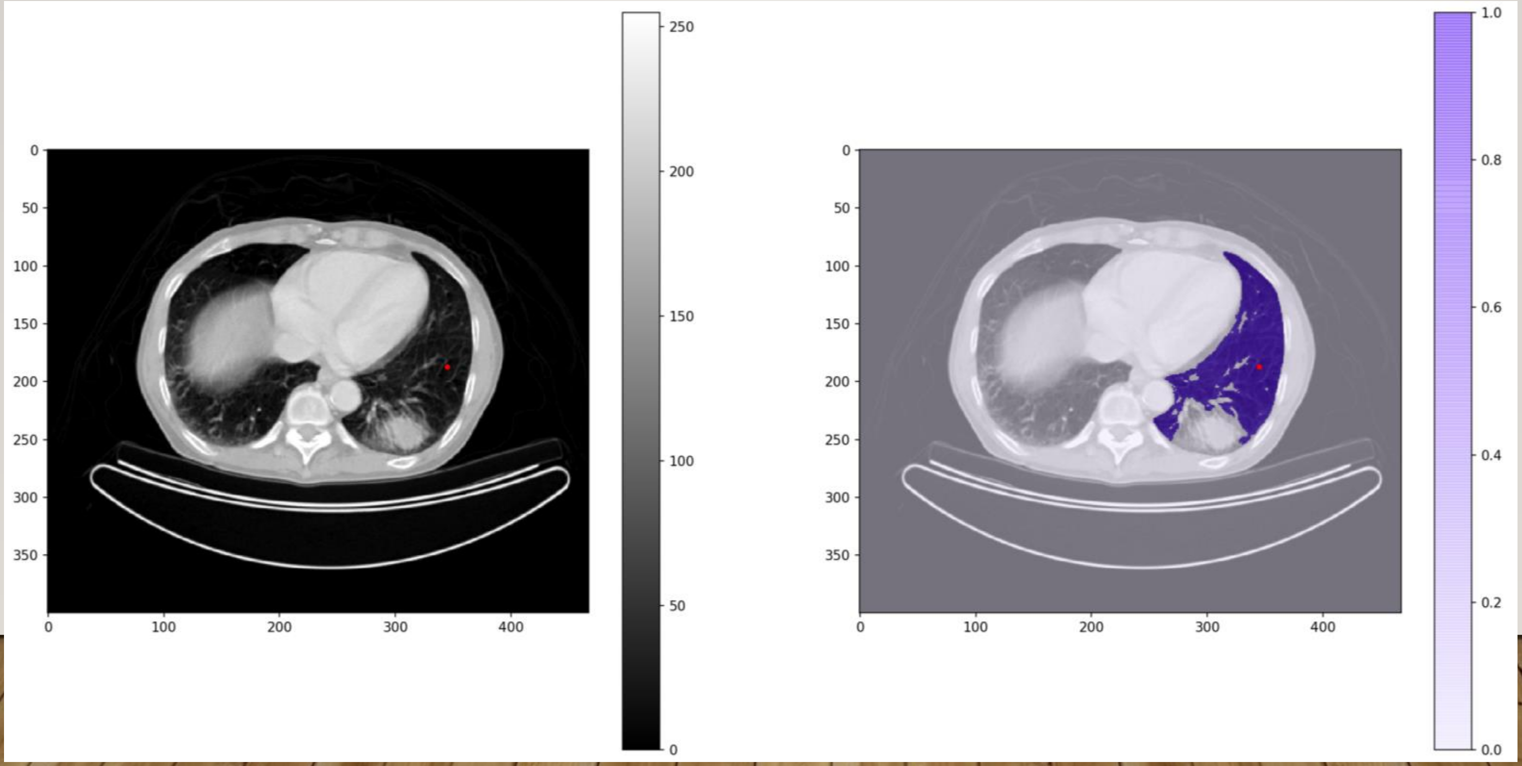
- Otsu's method: Easy way to implement

```python
import matplotlib.pyplot as plt
from skimage import io, filters
import numpy


# Load image file
fpath = 'F:/MIPL/SDS/0.excercise/3/ex3-2_segmentation/'
image = io.imread(fpath + 'coins.png')

# Otsu's Threshold
OstuTh = filters.threshold_otsu(image, nbins=256)
imageThresholded = image >= OstuTh

# Plot
plt.figure(figsize=(10, 5), dpi=150)
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.colorbar()
plt.subplot(1, 2, 2)
plt.imshow(imageThresholded, cmap='gray')
plt.title('Ostu Threhold = ' + str(OstuTh))

plt.show()
```



REGION GROWING

- Choose the seed pixel
- Check the neighboring pixels and add them to the region if they are similar to the seed
- Repeat step 2 for each of the newly added pixels
- Stop if no more pixels can be added



REGION GROWING

- Load image and define initial points & hyper-parameter

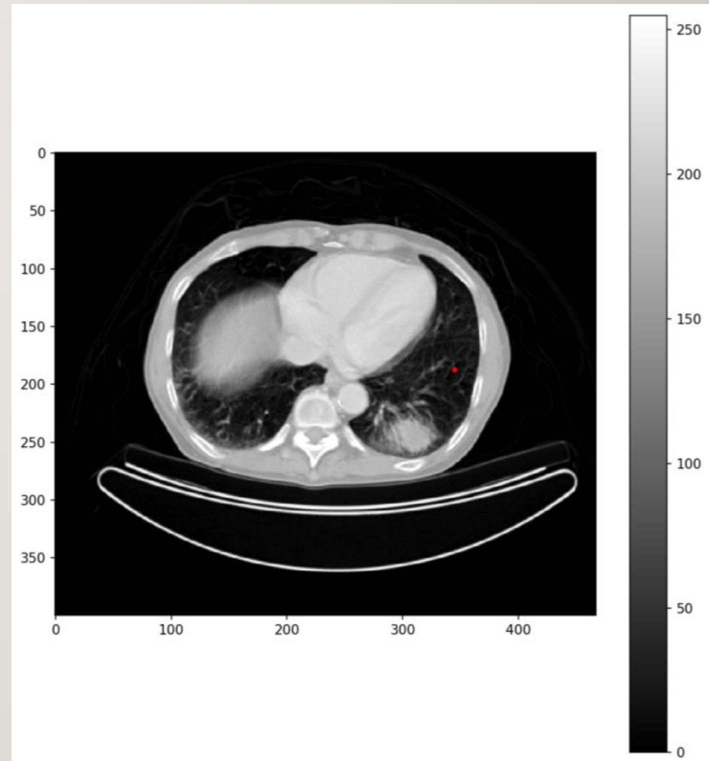
```python
import matplotlib.pyplot as plt
from skimage import io
import numpy
import seaborn as sns

# Load image file
fpath = 'F:/MIPL/SDS/0.excercise/3/ex3-2_segmentation/'
image = io.imread(fpath + 'medtest.png').astype('int')

# Region growing
seedI = 187
seedJ = 345

maxDiff = 50

sizeI, sizeJ = image.shape

segMap = numpy.zeros(image.shape)
segMap[seedI, seedJ] = 1

BFSQueue = numpy.zeros((1000, 2)).astype('int')
BFSQueue[0, ] = [seedI, seedJ]

SearchDirections = numpy.array([[0, -1],
                                [1, 0],
                                [0, 1],
                                [-1, 0]])
```

Initial point

Similar condition

Segmentation map

4-neighborhood



REGION GROWING

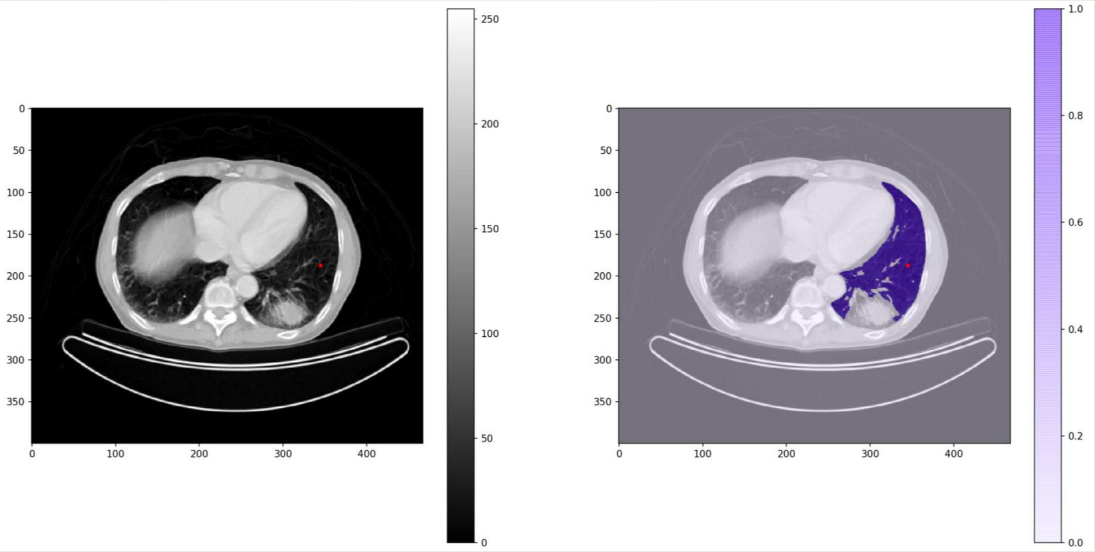
- Start region growing using BFS

```
head = 0
tail = 1
while head != tail:
    headI = BFSQueue[head, 0]
    headJ = BFSQueue[head, 1]

    i = 0
    while i < 4:
        newI = int(BFSQueue[head, 0] + SearchDirections[i, 0])
        newJ = int(BFSQueue[head, 1] + SearchDirections[i, 1])

        # BFS searching condition, Chained comparision rule is different from C
        if (-1 < newI < sizeI) and (-1 < newJ < sizeJ) and segMap[newI, newJ] == 0:
            # Segmentation condition
            if numpy.absolute(image[seedI, seedJ] - image[newI, newJ]) <= maxDiff:
                # print('[Push] : New tail node [' + str(newI) + ', ' + str(newJ) + '] wi
                segMap[newI, newJ] = 1
                BFSQueue[tail, ] = newI, newJ
                tail += 1

        i += 1

    # Pop
    # print('[Pop] : Head node [' + str(headI) + ', ' + str(headJ) + '] will be deleted')
    BFSQueue[0:tail-1, ] = BFSQueue[1:tail, ]
    tail -= 1
```

REGION GROWING

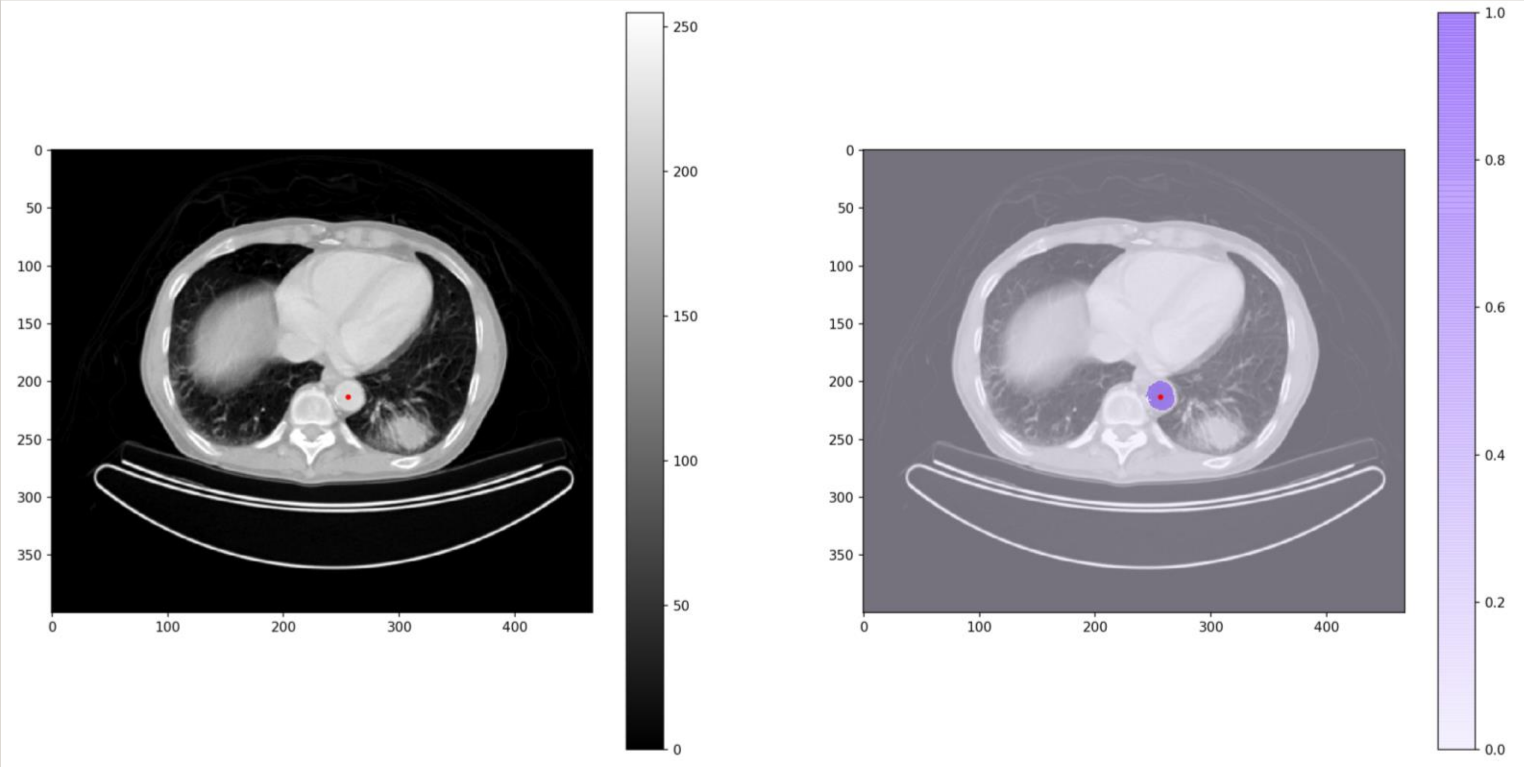
- Visualize the result

```python
# Plot
plt.figure(figsize=(20, 10), dpi=150)
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray', vmin=0, vmax=255)
plt.colorbar()
plt.plot(seedJ, seedI, 'r.')

plt.subplot(1, 2, 2)
plt.imshow(image, cmap='gray', vmin=0, vmax=255)
my_cmap = sns.light_palette(sns.xkcd_rgb["purplish blue"], input="his", as_cmap=True, reverse=False)
plt.imshow(segMap, cmap=my_cmap, alpha=0.5, vmin=0, vmax=1)
plt.colorbar()
plt.plot(seedJ, seedI, 'r.')

plt.show()
```



REGION GROWING

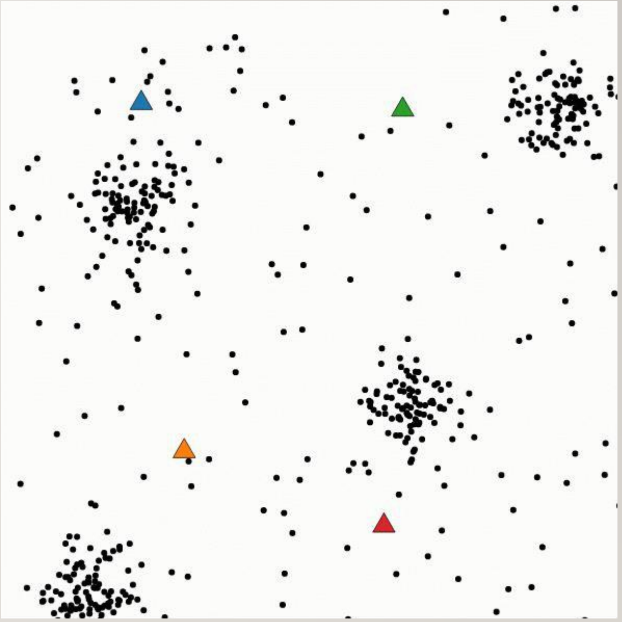
- Additional work: Aorta segmentation
- Do it your self (Adjust seed point & maxDiff)



K-MEANS

- Initial step
  - Pick *k* cluster centers randomly
  - Assign each sample to closest center

- Iteration steps
  - Compute means in each cluster: $\mu_i = \frac{1}{|D_i|}\sum_{x \in D_i} x$
  - Re-assign each sample to the closest mean
  - Iterate until clusters stop changing



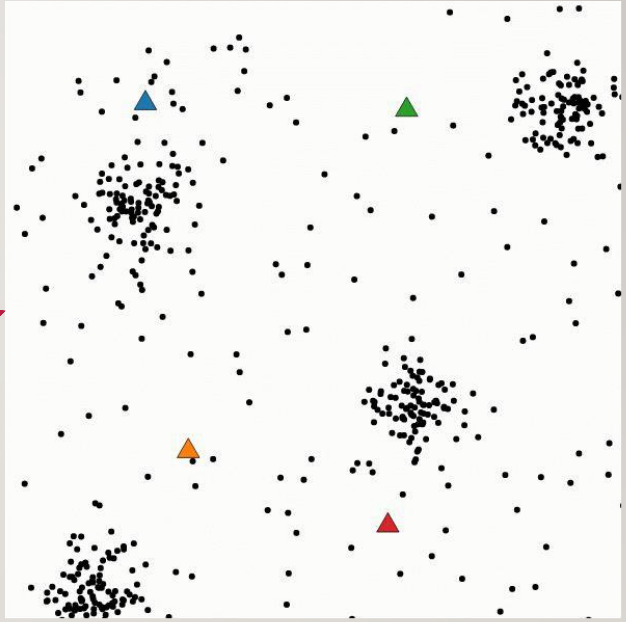
K-MEANS

- Segmentation using color (RGB value) clustering
- Sky, Cloud, Ground segmentation (k=3)

K-MEANS

- Set hyper-parameters for k-means clustering

```
import matplotlib.pyplot as plt
from skimage import io
import numpy

# Load image file
fpath = 'F:/MIPL/SDS/0.excercise/3/ex3-2_segmentation/'
image = io.imread(fpath + 'wallpaper.jpg')

# image -> RGB vector conversion
sizeI, sizeJ, sizeK = image.shape
imageRGBvect = numpy.zeros((sizeI*sizeJ, sizeK))
imageRGBvectIdx = 0
i = 0
while i < sizeI:
    j = 0
    while j < sizeJ:
        imageRGBvect[imageRGBvectIdx, :] = image[i, j, :]
        imageRGBvectIdx += 1
        j += 1
    i += 1

# Kmeans initialization
k = 3
ClusterCenters = numpy.random.randint(low=0, high=255, size=(k, 3))
```



K-MEANS

- Perform k-means clustering

1. Calculate every distance from center to points
2. Cluster assignment (= closest center)
3. Cluster Update

- Until cluster center converge

```
# Kmeans
convergeLimit = 10
deltaMeans = 99999
maxIter = 10000
iterN = 0
while deltaMeans > convergeLimit and iterN < maxIter:
    iterN += 1

    # Calculate every distance from center to points
    dist2Centers = numpy.zeros((sizeI*sizeJ, 3))
    i = 0
    while i < k:
        temp = imageRGBvect - ClusterCenters[i, ]
        temp = numpy.square(temp)
        temp = numpy.sqrt(temp.sum(axis=1))
        dist2Centers[:, i] = temp
        i += 1

    # Cluster assignment
    clustersAssigned = dist2Centers.argmin(axis=1)

    # ClusterCenter update
    NewClusterCenters = numpy.zeros((3,3))
    i = 0
    while i < k:
        clusterList = numpy.where(clustersAssigned == i)
        valueInlist = imageRGBvect[clusterList, ]
        NewClusterCenters[i, ] = valueInlist.mean(axis=1)
        i += 1

    # Calculate convergence
    deltaMeans = NewClusterCenters - ClusterCenters
    deltaMeans = numpy.square(deltaMeans)
    deltaMeans = numpy.sqrt(deltaMeans.sum(axis=1)).sum()

    ClusterCenters = NewClusterCenters
```

K-MEANS

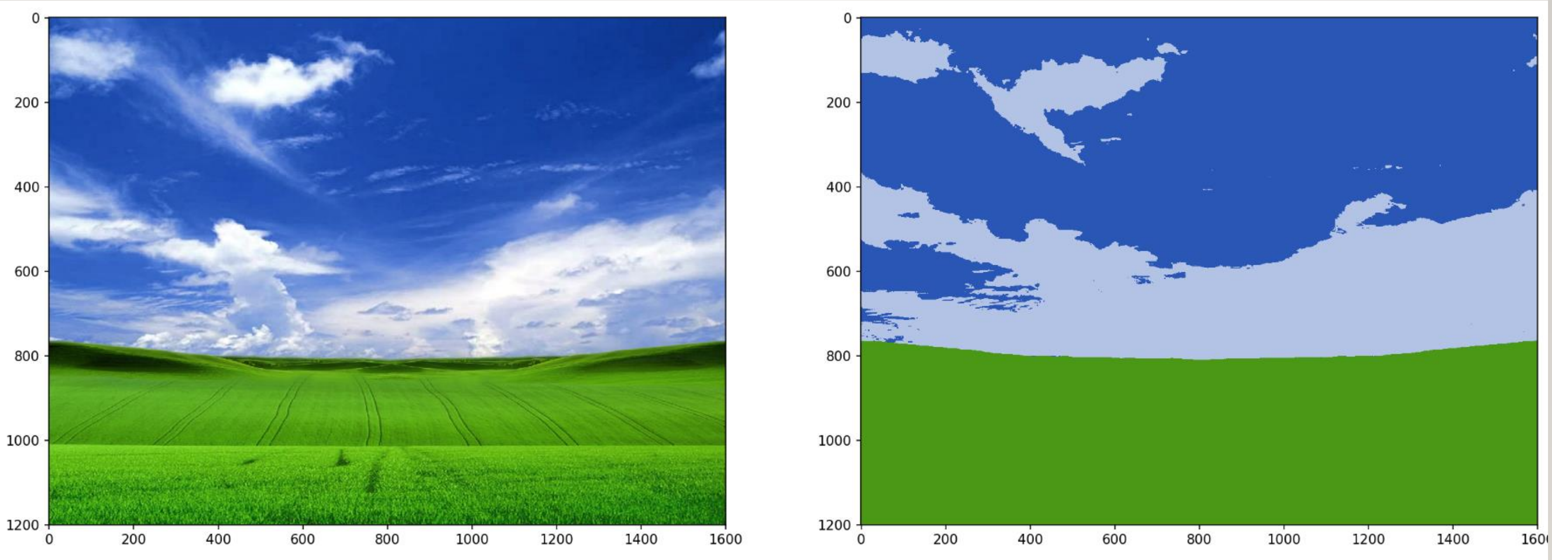
- Display the results

```
# Convert image value to ClusterCenters
ClusterCenters = numpy.round(ClusterCenters).astype('uint8')

imageClustered = numpy.zeros(image.shape).astype('uint8')

imageRGBvectIdx = 0
i = 0
while i < sizeI:
    j = 0
    while j < sizeJ:
        imageClustered[i, j, :] = ClusterCenters[clustersAssigned[imageRGBvectIdx],:]
        imageRGBvectIdx += 1
        j += 1
    i += 1


# Plot
plt.figure(figsize=(20, 10), dpi=150)
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray', vmin=0, vmax=255)

plt.subplot(1, 2, 2)
plt.imshow(imageClustered, cmap='gray', vmin=0, vmax=255)

plt.show()
```

Change pixel value to the cluster center

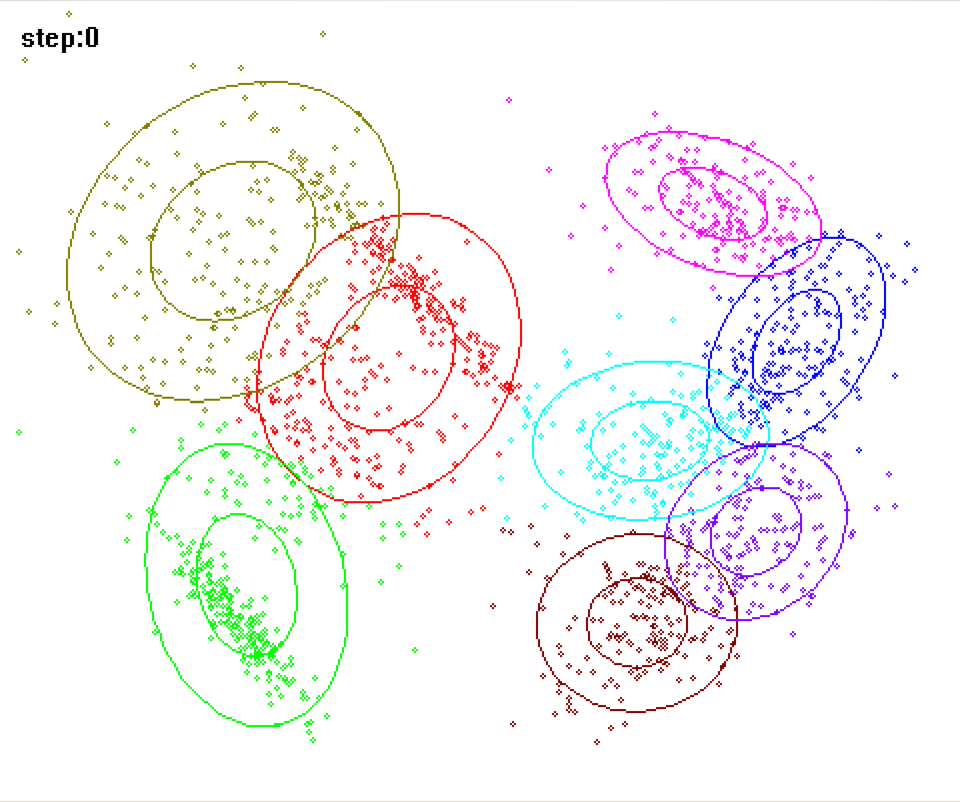
K-MEANS

- Segmentation result



GMM

- Find multiple Gaussian distributions in data
- Use expectation-maximization algorithm



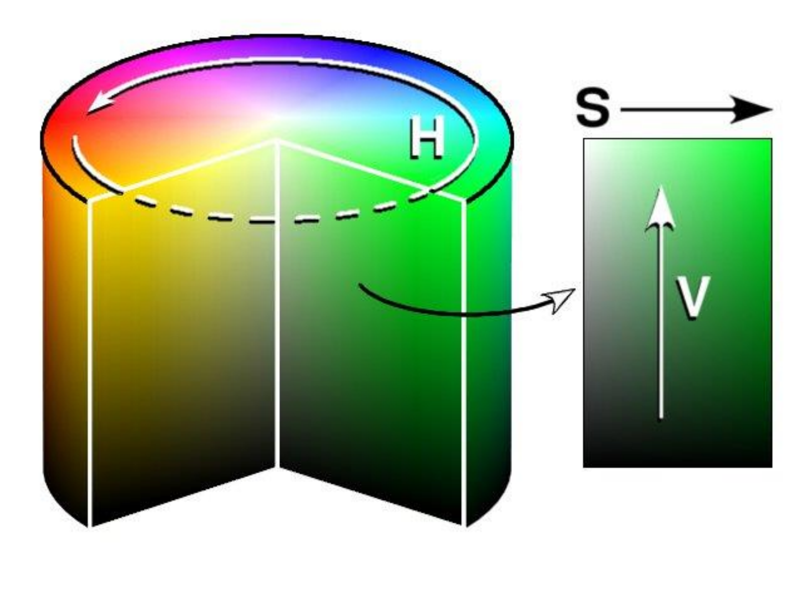
GMM

Color clustering using Hue channel + GMM

GMM

Color clustering using Hue channel + GMM

```
# image -> HSV vector conversion
imageHSV = color.rgb2hsv(image)

sizeI, sizeJ, sizeK = imageHSV.shape
imageHSVvect = numpy.zeros((sizeI*sizeJ, sizeK))
imageHSVvectIdx = 0
i = 0
while i < sizeI:
   j = 0
   while j < sizeJ:
      imageHSVvect[imageHSVvectIdx, :] = imageHSV[i, j, :]
      imageHSVvectIdx += 1
      j += 1
   i += 1

HueChannel = imageHSVvect[:, 0].reshape(-1, 1)
```



GMM

Color clustering using Hue channel + GMM

```
# GMM of HueChannel
clusterN = 3
GMMModel = GaussianMixture(clusterN).fit(HueChannel)

clustersAssigned = GMMModel.predict(HueChannel)
ClusterCenters = GMMModel.means_
clusteredHue = ClusterCenters[clustersAssigned]
```

```
# Convert image value to ClusterCenters
imageHSVvectClustered = imageHSVvect.copy()

imageHSVvectClustered = imageHSVvectClustered.transpose()
clusteredHue = clusteredHue.transpose()

imageHSVvectClustered[0, ] = clusteredHue
imageHSVvectClustered[1, ] = 1
imageHSVvectClustered[2, ] = 1
imageHSVvectClustered = imageHSVvectClustered.transpose()

imageClusteredHSV = numpy.zeros(image.shape)

imageHSVvectIdx = 0
i = 0
while i < sizeI:
    j = 0
    while j < sizeJ:
        imageClusteredHSV[i, j, :] = imageHSVvectClustered[imageHSVvectIdx, :]
        imageHSVvectIdx += 1
        j += 1
    i += 1

imageClustered = color.hsv2rgb(imageClusteredHSV)
```

GMM


0
250
500
750
1000
1250
1500
1750
0 200 400 600 800 1000 1200
0
250
500
750
1000
1250
1500
1750
0 200 400 600 800 1000 1200

PLOT GMM

```
# Plot GMM
xvec = numpy.arange(0, 1, .001).transpose()

plt.figure(figsize=(20, 10), dpi=150)

G1 = stats.norm(loc=ClusterCenters[0], scale=numpy.sqrt(GMMModel.covariances_[0]))
distributionG1 = GMMModel.weights_[0]*G1.pdf(xvec).transpose()

G2 = stats.norm(loc=ClusterCenters[1], scale=numpy.sqrt(GMMModel.covariances_[1]))
distributionG2 = GMMModel.weights_[1]*G2.pdf(xvec).transpose()

G3 = stats.norm(loc=ClusterCenters[2], scale=numpy.sqrt(GMMModel.covariances_[2]))
distributionG3 = GMMModel.weights_[2]*G3.pdf(xvec).transpose()

plt.hist(HueChannel, bins=128, density=True, alpha=0.3, label='Histogram')
plt.plot(xvec, distributionG1 + distributionG2 + distributionG3, alpha=0.2, linewidth=5, label='G1+G2+G3', color='magenta')
G1Plot = plt.plot(xvec, distributionG1, label='G1', color='b', linewidth=2)
G2Plot = plt.plot(xvec, distributionG2, label='G2', linewidth=2)
G3Plot = plt.plot(xvec, distributionG3, label='G3', linewidth=2)

plt.grid(True, which='major', axis='both')
plt.legend(fontsize=20)

plt.show()
```

GMM

GMM function output = Mean, Sigma
Real GMM = Weight1*G1 + Weight2*G2 + Weight3*G3