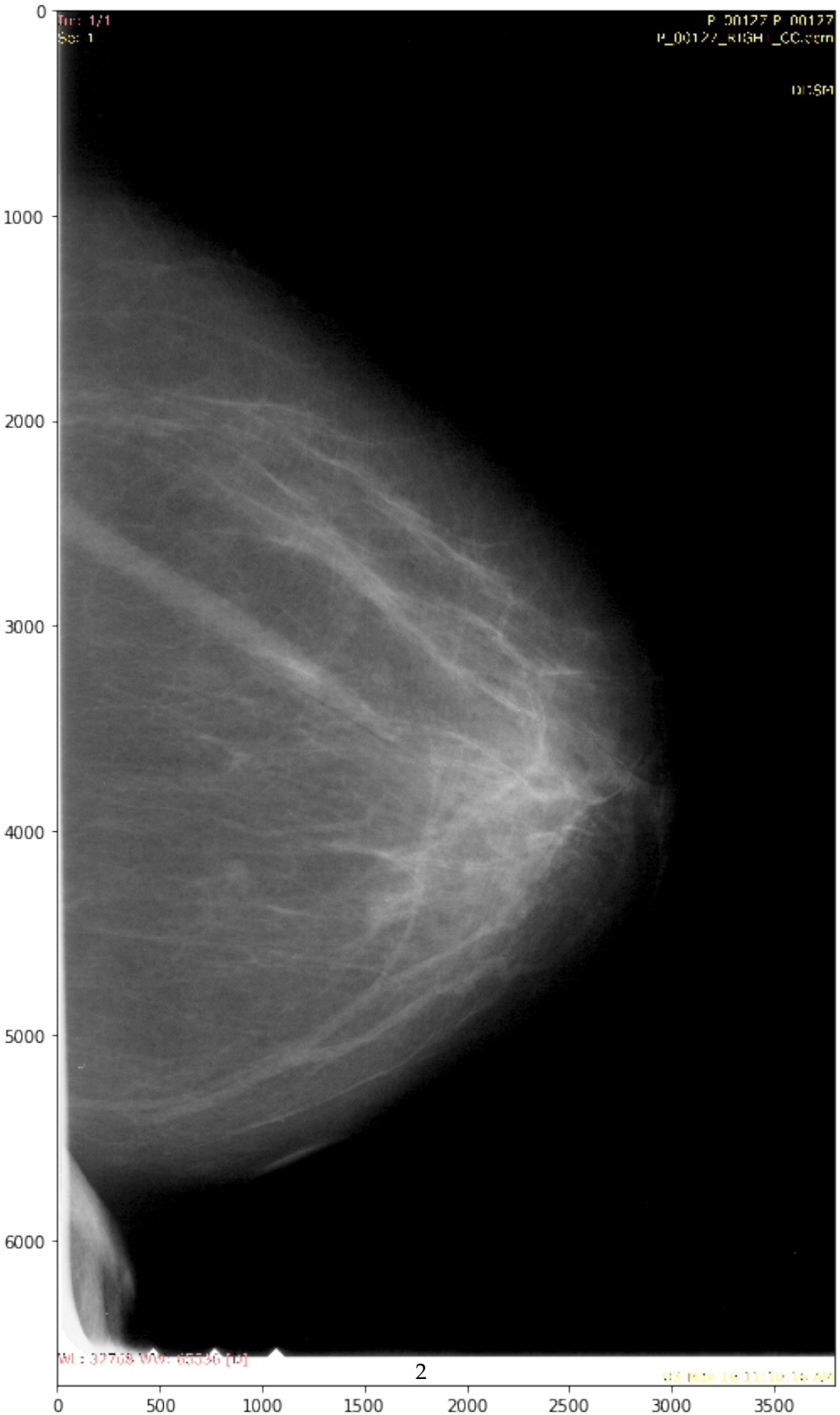
MedicalFeatureExtraction

October 3, 2018

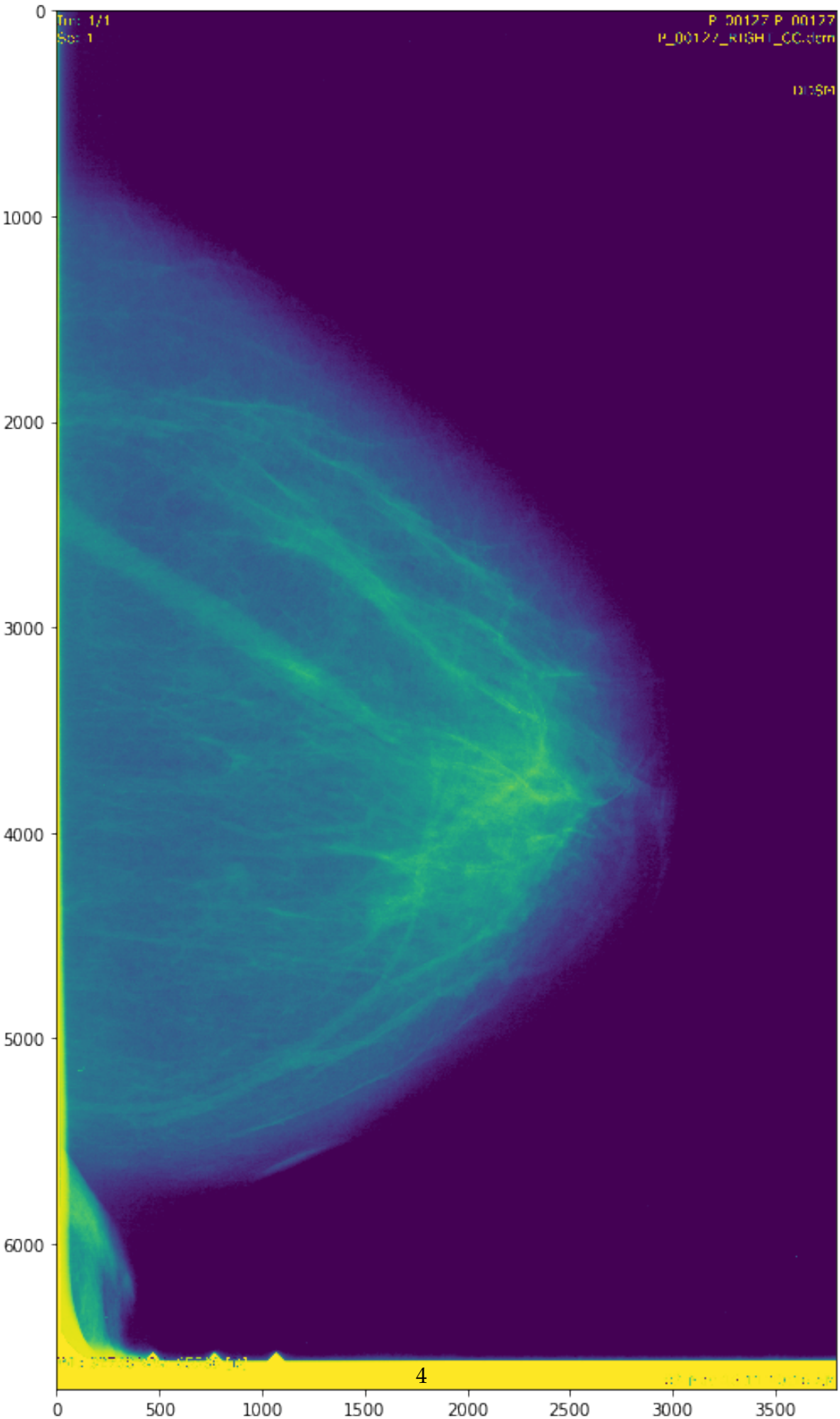
1 Medical Feature Extraction

```
In [6]: import matplotlib.pyplot as plt
        import matplotlib.image as mpimg
        import numpy as np
```

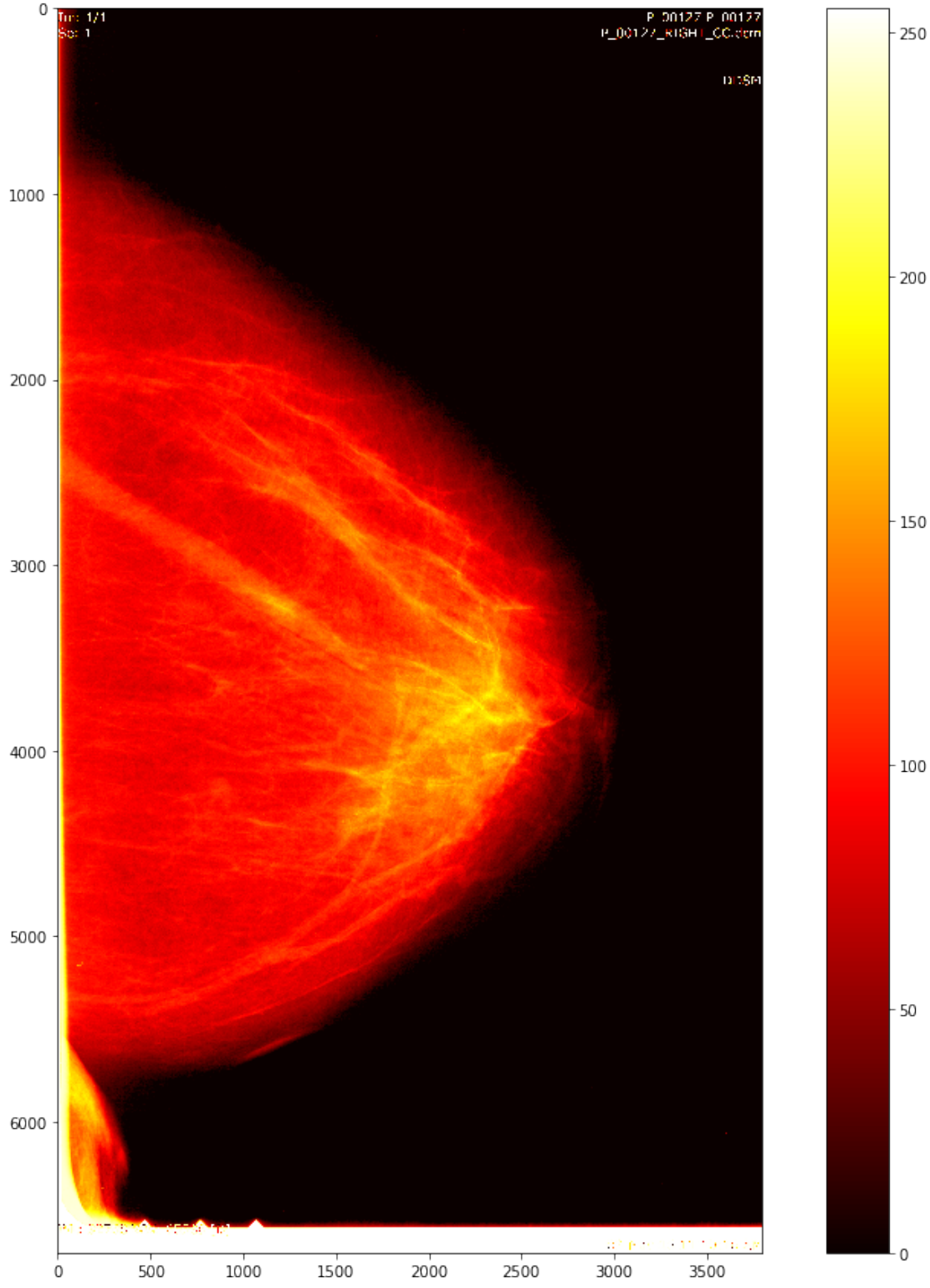
```
In [145]: img = mpimg.imread('IMG-0001-00001.jpg')
          plt.figure(figsize=(15, 15))
          plt.imshow(img)
          plt.show()
```


0
1000
2000
3000
4000
5000
6000
0
500
1000
1500
2000
2500
3000
3500

```
In [8]: lum_img = img[:,:,0]
        plt.figure(figsize=(15, 15))
        plt.imshow(lum_img)
        plt.show()
```

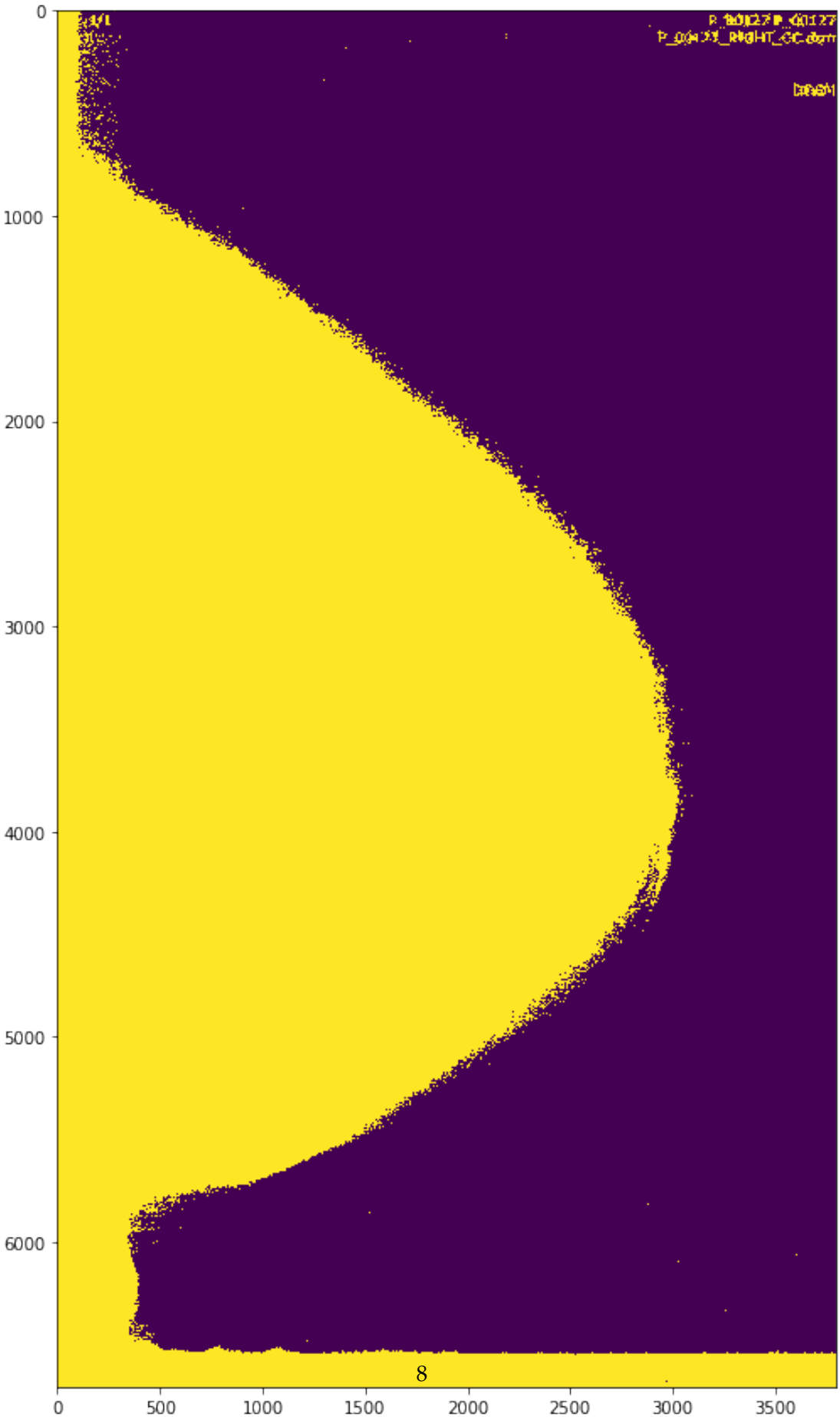

0
1000
2000
3000
4000
5000
6000
0
500
1000
1500
2000
2500
3000
3500

In [9]:
```
plt.figure(figsize=(15, 15))
plt.imshow(lum_img, cmap='hot')
plt.colorbar()
plt.show()
```

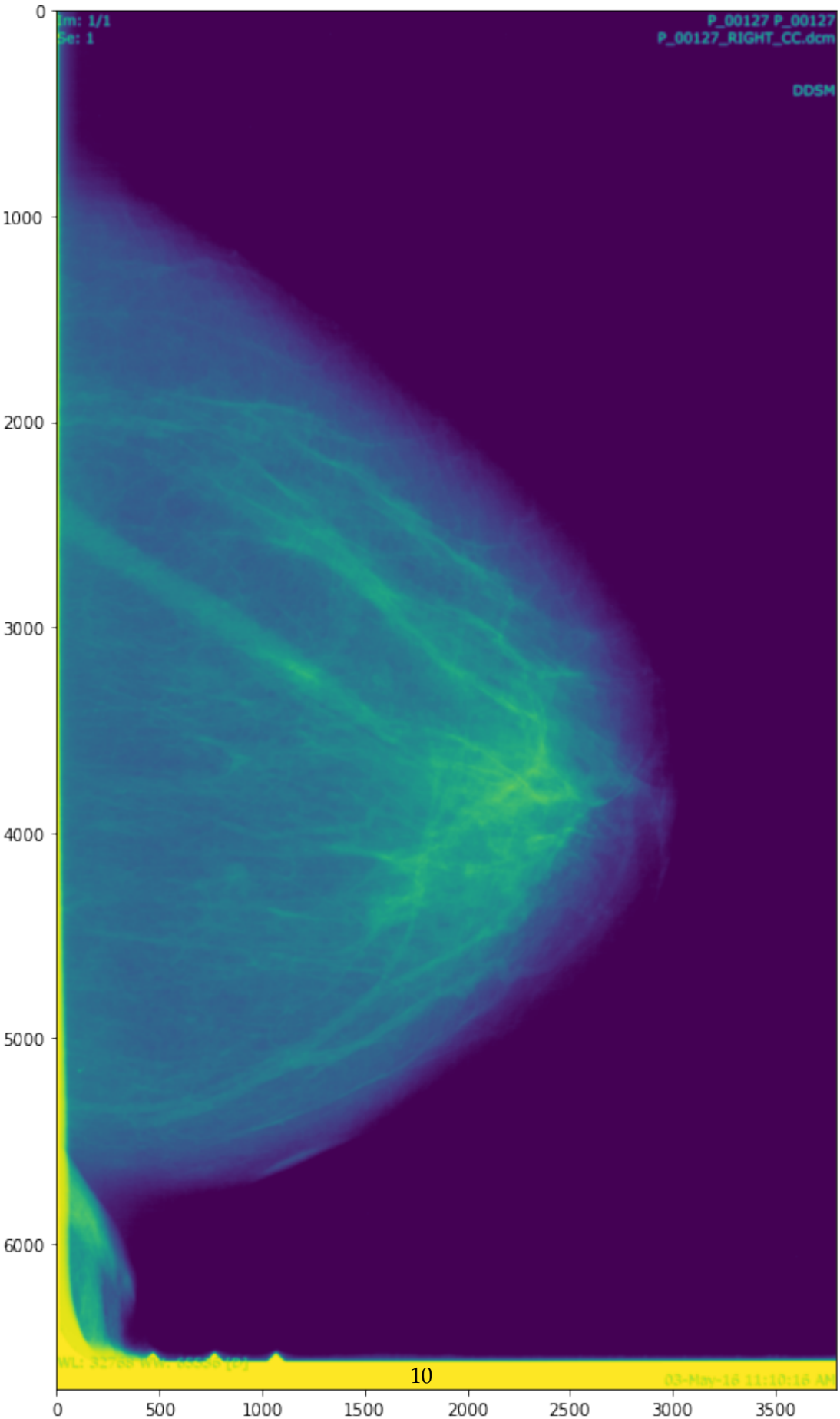


In [10]: plt.figure(figsize=(15, 15))

```
plt.imshow(lum_img, clim=(0.0, 0.7))
plt.show()
```


0
1000
2000
3000
4000
5000
6000
0
500
1000
1500
2000
2500
3000
3500

```
In [11]: plt.figure(figsize=(15, 15))
         plt.imshow(lum_img, interpolation='bicubic')
         plt.show()
```

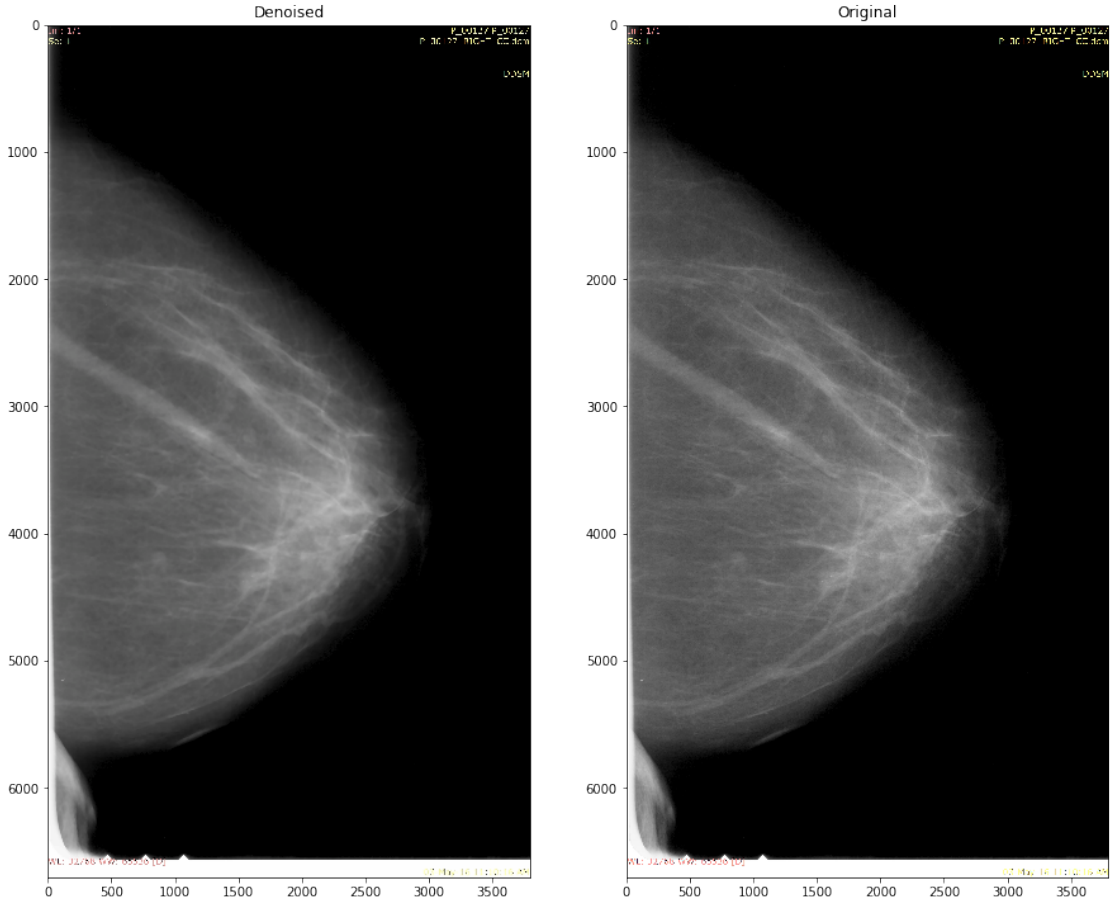

Im: 1/1
Se: 1
P_00127 P_00127
P_00127_RIGHT_CC.dcm
DDSM
0
1000
2000
3000
4000
5000
6000
0
500
1000
1500
2000
2500
3000
3500

1.1 Processing the Image

```
In [146]: import cv2
```

```
In [148]: dst = cv2.fastNlMeansDenoisingColored(img,None,10,10,7,21)
```

```
In [16]: plt.figure(figsize=(15, 15))
         plt.subplot(121),plt.imshow(dst), plt.title('Denoised')
         plt.subplot(122),plt.imshow(img), plt.title('Original')
         plt.show()
```



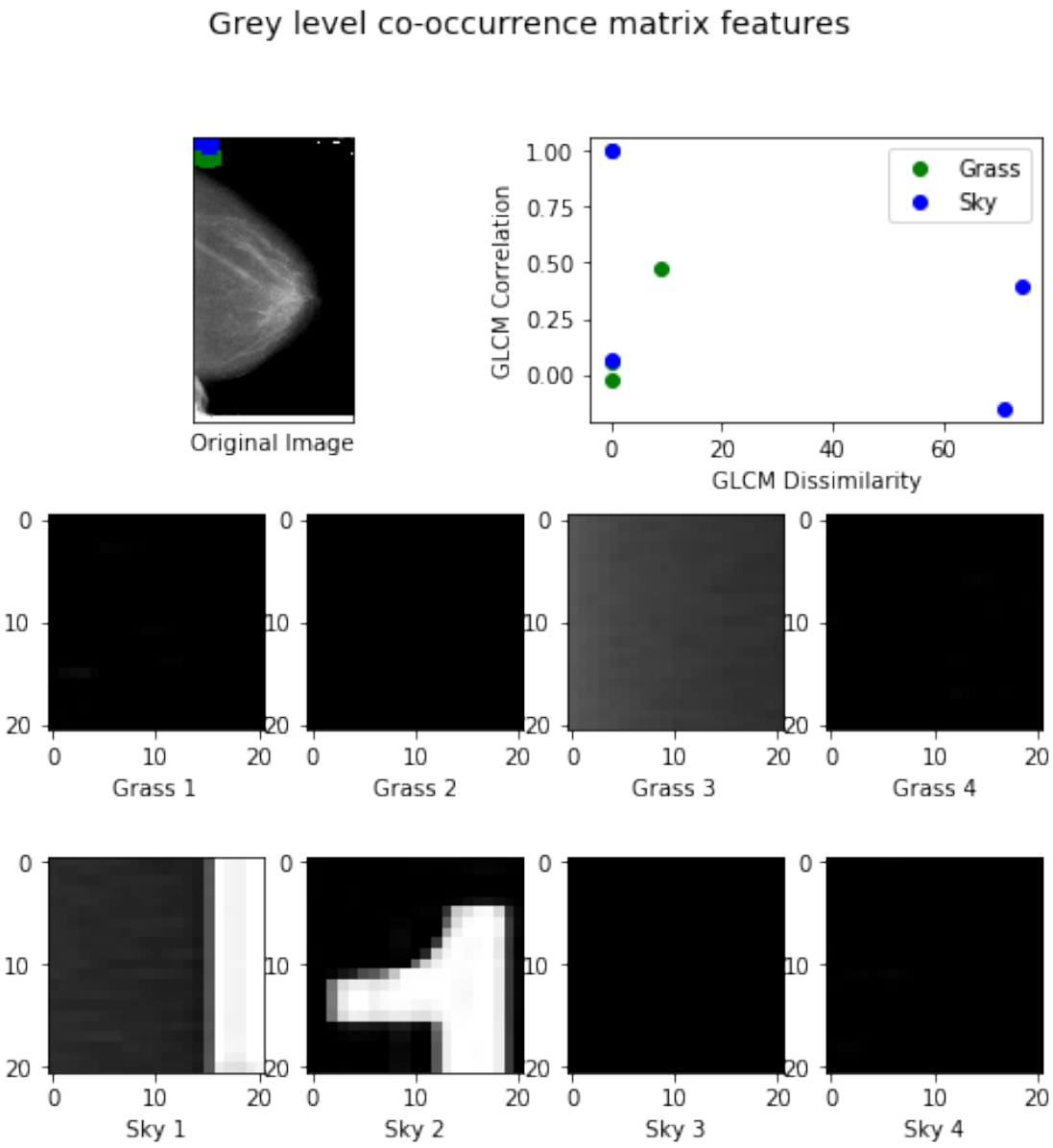
1.2 GLCM - Gray Level Co-Occurence Matrix

```
In [21]: from skimage.feature import greycomatrix, greycoprops
         from skimage import data
```

```
PATCH_SIZE = 21

# open the camera image
# image = data.camera()
image = lum_img

# select some patches from grassy areas of the image
grass_locations = [(474, 291), (440, 433), (466, 18), (462, 236)]
grass_patches = []
for loc in grass_locations:
    grass_patches.append(image[loc[0]:loc[0] + PATCH_SIZE,
                               loc[1]:loc[1] + PATCH_SIZE])

# select some patches from sky areas of the image
sky_locations = [(54, 48), (21, 233), (90, 380), (195, 330)]
sky_patches = []
for loc in sky_locations:
    sky_patches.append(image[loc[0]:loc[0] + PATCH_SIZE,
                             loc[1]:loc[1] + PATCH_SIZE])

# compute some GLCM properties each patch
xs = []
ys = []
for patch in (grass_patches + sky_patches):
    glcm = greycomatrix(patch, [5], [0], 256, symmetric=True, normed=True)
    xs.append(greycoprops(glcm, 'dissimilarity')[0, 0])
    ys.append(greycoprops(glcm, 'correlation')[0, 0])

# create the figure
fig = plt.figure(figsize=(8, 8))

# display original image with locations of patches
ax = fig.add_subplot(3, 2, 1)
ax.imshow(image, cmap=plt.cm.gray, interpolation='nearest',
          vmin=0, vmax=255)
for (y, x) in grass_locations:
    ax.plot(x + PATCH_SIZE / 2, y + PATCH_SIZE / 2, 'gs')
for (y, x) in sky_locations:
    ax.plot(x + PATCH_SIZE / 2, y + PATCH_SIZE / 2, 'bs')
ax.set_xlabel('Original Image')
ax.set_xticks([])
ax.set_yticks([])
ax.axis('image')

# for each patch, plot (dissimilarity, correlation)
ax = fig.add_subplot(3, 2, 2)
ax.plot(xs[:len(grass_patches)], ys[:len(grass_patches)], 'go',
```

```
        label='Grass')
ax.plot(xs[len(grass_patches):], ys[len(grass_patches):], 'bo',
        label='Sky')
ax.set_xlabel('GLCM Dissimilarity')
ax.set_ylabel('GLCM Correlation')
ax.legend()

# display the image patches
for i, patch in enumerate(grass_patches):
    ax = fig.add_subplot(3, len(grass_patches), len(grass_patches)*1 + i + 1)
    ax.imshow(patch, cmap=plt.cm.gray, interpolation='nearest',
              vmin=0, vmax=255)
    ax.set_xlabel('Grass %d' % (i + 1))

for i, patch in enumerate(sky_patches):
    ax = fig.add_subplot(3, len(sky_patches), len(sky_patches)*2 + i + 1)
    ax.imshow(patch, cmap=plt.cm.gray, interpolation='nearest',
              vmin=0, vmax=255)
    ax.set_xlabel('Sky %d' % (i + 1))


# display the patches and plot
fig.suptitle('Grey level co-occurrence matrix features', fontsize=14)
plt.show()
```

```
C:\Users\shadowleaf\Anaconda3\lib\site-packages\skimage\feature\texture.py:109: FutureWarning:
  if np.issubdtype(image.dtype, np.float):
```



In [42]: %%file Texture.py

```
import numpy
import threading
import logging

class CooccurenceMatrixTextures(object):
    '''The class performs determines the cooccurence matrix
    for a set window size over and image.  It also computes
    and returns a number of statisitcs computed using the
    output cooccurence matrix'''
```

```
def __init__(self, image, windowRadius = 2):
    self.image = image
    self.windowRadius = windowRadius
    self.xRange = range(-self.windowRadius,self.windowRadius+1)
    self.yRange = range(-self.windowRadius,self.windowRadius+1)
    self.lock = threading.Lock()
    self.__getMatrix()

def acquireLock(self):
    """Hook for multitreaded operation to acquire a lock."""
    self.lock.acquire()

def releaseLock(self):
    """Hook for multithreaded operation to release a lock."""
    self.lock.release()

def getDissimlarity(self):
    '''Getter method to extract the Dissimilarity'''
    return self.__dissimlarity()


def getEntropy(self):
    '''Getter method to extract the Entropy'''
    return self.__entropy()


def getASM(self):
    '''Getter method to extract the Angular Second Momentum'''
    return self.__asm()


def getMean(self):
    '''Getter method to extract the Mean'''
    return self.__mean()


def getVarMean(self):
    '''Getter method to obtain the Mean and Variance'''
    #get texture
    mean = self.__mean()
    var = self.__variance(mean)
    return var, mean

def getCorrVarMean(self):
    '''Getter method to obtain the Mean, Variance and Correlation'''
    #get texture
    mean = self.__mean()
```

```
        var = self.__variance(mean)
        corr = self.__correlation(mean, var)
        return corr, var, mean

    def __dissimlarity(self):
        '''This method gets the contrast image derived from the GLCM'''

        #first create the contrast matrix
        topHalf = self.__diagMatrix()
        weights = topHalf + topHalf.T
        print(weights)

        #flatten
        weights = weights.flatten()

        #apply to the image through broadcasting
        weightedGLCM = self.GLCM * weights

        #sum along the third axis
        dissimilarity = numpy.sum(weightedGLCM,2)

        #return the result
        return dissimilarity

    def __entropy(self):
        '''This method gets the entropy from the GLCM'''

        #mask off the glcm
        maGLCM = numpy.ma.MaskedArray(self.GLCM, self.GLCM == 0)

        #calculate the logs and sum along the 3rd dimension
        entropy = numpy.sum(numpy.log(maGLCM) * maGLCM * (-1), 2)

        #return the result
        return entropy

    def __asm(self):
        '''This method returns the Angular Second Momentum for the GLCM'''

        return numpy.sum(self.GLCM * self.GLCM, 2)


    def __mean(self):
        '''This method gets the GLCM mean'''

        #create the output array
        shape = numpy.shape(self.GLCM)
        meanGLCM = numpy.zeros([shape[0],shape[1]])
```

```
        #Loop over each quantisation
        steps = range(0,256,16)
        coeff = 0
        for x in steps:

            #sum the contents of the row and multiply by the coocurrence (GLCM Mean)
            summedStep = numpy.sum((self.GLCM[:,:,x:x+16] * coeff),2)
            coeff += 1

            #sum into the output image
            meanGLCM += summedStep

        #return the output image
        return meanGLCM

    def __variance(self, mean):
        '''This method gets the GLCM variance and mean'''
        #create the output array
        shape = numpy.shape(self.GLCM)
        varGLCM = numpy.zeros([shape[0],shape[1]])

        #Loop over each quantisation
        steps = range(256)
        coeff = numpy.repeat(range(16), 16)
        for x in steps:

            #sum the contents of the row and multiply by the coocurrence (GLCM Var)
            step = self.GLCM[:,:,x] * numpy.power(coeff[x] - mean, 2)

            #sum into the output image
            varGLCM += step

        #return the output image
        return varGLCM

    def __correlation(self, mean, var):
        '''This method calucaltes the GLCM correlation, variance and mean'''
        #create the output array
        shape = numpy.shape(self.GLCM)
        corrGLCM = numpy.zeros([shape[0],shape[1]])

        #Loop over quantisation steps
        steps = range(256)
        coeffA = numpy.repeat(range(16), 16)
        coeffB = numpy.tile(range(16), 16)
        for x in steps:
```

```
            #determine correlation (small additive value to prevent divide by zero)
            step = self.GLCM[:,:,x] * ((coeffA[x] - mean) * (coeffB[x] - mean)) / (va

            #sum into the output image
            corrGLCM += step

        #return the output image
        return corrGLCM


    def __getMatrix(self):
        '''This function computes the cooccurence matrix'''

        #create output arrays
        shape = numpy.shape(self.image)
        self.GLCM = numpy.zeros([shape[0],shape[1],256], numpy.int16)

        #set image as masked array
        maImage = numpy.ma.masked_array(self.image, self.image < 0)

        #scale the image to 4bit
        scaledImage = self.__scaleImage(maImage)

        #create the x and y indeices for the lookup
        [self.indexX, self.indexY] = numpy.meshgrid(range(shape[1]), range(shape[0]))

        #shift over all directions to create rotational invariance
        x = [0,1,1,1]
        y = [1,1,0,-1]
        for i in range(4):

            #roll the image
            shiftedImage = self.__roll2d(scaledImage, x[i],y[i])

            #compute the indices based upon the pixel values
            self.index = self.__findIndex(scaledImage, shiftedImage)
            self.indexInverse = self.__findIndex(shiftedImage, scaledImage)

            #do processing
            self.processElements()

        #get denominator
        den = (self.windowRadius * 2 + 1) ** 2 * 8.0

        #convert to probabilities
        self.GLCM = self.GLCM / den

    def __scaleImage(self, image, scaledMin = 0., scaledMax = 15.):
```

```
        '''This private method scales the image to 4bit range'''

        #gets min and max
        imageMax = numpy.max(image)
        imageMin = numpy.min(image)

        #scale and replace the image
        scaledIm = (((image - imageMin) * (scaledMax-scaledMin)) / (imageMax - imageMi
        scaledIm = numpy.round(scaledIm).astype(numpy.int32)
        #imgplot = plt.imshow(scaledIm) #@UnusedVariable
        #imgplot.set_cmap('Greys')
        #plt.show()
        return scaledIm

    def appendGLCM(self,i,j):

        #roll the indexes
        rolledIndex = self.__roll2d(self.index, i, j)
        rolledIndexInverse = self.__roll2d(self.indexInverse, i, j)

        #use the indices to fill the GLCM
        self.acquireLock()
        self.GLCM[self.indexY,self.indexX,rolledIndex] += 1
        self.GLCM[self.indexY,self.indexX,rolledIndexInverse] += 1
        self.releaseLock()

    def processYElements(self, i):
        for j in self.yRange:
            self.appendGLCM(i,j)

    def processElements(self):
        threadList = []
        for i in self.xRange:
            t = threading.Thread(name = str(i), target = self.processYElements, args =
            threadList.append(t)
            logging.info('Starting thread %d', i)
            t.start()
        logging.debug('Waiting to join threads')
        for t in threadList:
            logging.debug('About to join thread %s', t.getName())
            t.join()

    def __roll2d(self, image, xdir, ydir):
        '''This private method rolls the image in the given directions'''
        tmp = numpy.roll(image, ydir, 0)
        return numpy.roll(tmp, xdir, 1)

    def __findIndex(self, referenceImage, neighbourImage):
```

```
            '''Finds the index into which a count must be inserted'''
            index = referenceImage * 16 + neighbourImage
            return index

        def __diagMatrix(self):
            '''Method to create a diagonal matrix'''

            xRange = range(16)
            yRange = range(16)
            dMatrix = numpy.zeros([16,16], numpy.int16)

            #Loop over the array and fill it in
            xInc = 0
            for y in yRange:
                val = 0
                for x in xRange:

                    xPos = x + xInc

                    if xPos >= 16:
                        continue
                    else:
                        #place the value into the matrix
                        dMatrix[y,xPos] = val

                    val+=1
                xInc+=1
            return dMatrix


        def getGLCM(self):
            '''This returns the GLCM'''
            return self.GLCM
```

```
Overwriting Texture.py
```

```
In [43]: from Texture import *
         CooccurenceMatrixTextures(lum_img)
```

```
---------------------------------------------------------------------------

MemoryError                               Traceback (most recent call last)

<ipython-input-43-be2d1f47fa3b> in <module>()
  1 from Texture import *
```

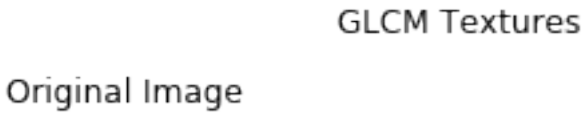
```
----> 2 CooccurenceMatrixTextures(lum_img)


    D:\MachineLearningProjects\BreastCancerDetection\Texture.py in __init__(self, image, wi
     15         self.xRange = range(-self.windowRadius,self.windowRadius+1)
     16         self.yRange = range(-self.windowRadius,self.windowRadius+1)
---> 17         self.lock = threading.Lock()
     18         self.__getMatrix()
     19


    D:\MachineLearningProjects\BreastCancerDetection\Texture.py in __getMatrix(self)
    167
    168         #create output arrays
--> 169         shape = numpy.shape(self.image)
    170         self.GLCM = numpy.zeros([shape[0],shape[1],256], numpy.int16)
    171


    MemoryError:
```

```
In [78]: import matplotlib.pyplot as plt
         import gdal, gdalconst
         import numpy as np
         from skimage.feature import greycomatrix, greycoprops

         #Read SAR image into Numpy Array
         filename = "greyscale.png"
         sarfile = gdal.Open(filename, gdalconst.GA_ReadOnly)
         sarraster = sarfile.ReadAsArray()
         print(sarraster.ndim)

         #Create rasters to receive texture and define filenames
         contrastraster = np.copy(sarraster)
         contrastraster[:] = 0

         dissimilarityraster = np.copy(sarraster)
         dissimilarityraster[:] = 0

         homogeneityraster = np.copy(sarraster)
         homogeneityraster[:] = 0

         energyraster = np.copy(sarraster)
         energyraster[:] = 0

         correlationraster = np.copy(sarraster)
         correlationraster[:] = 0
```

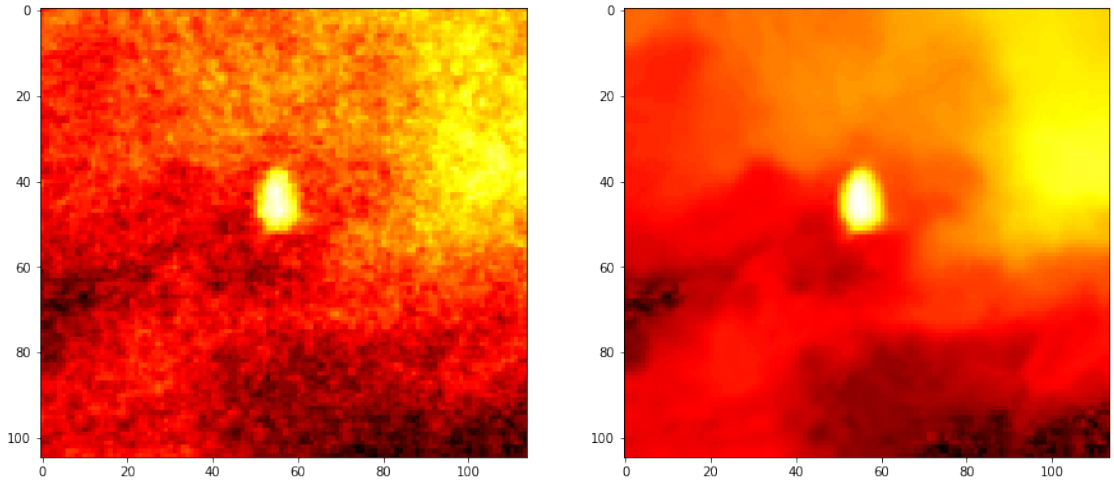
```
ASMraster = np.copy(sarraster)
ASMraster[:] = 0

# Create figure to receive results
fig = plt.figure()
fig.suptitle('GLCM Textures')

# In first subplot add original SAR image
ax = plt.subplot(241)
plt.axis('off')
ax.set_title('Original Image')
plt.imshow(sarraster, cmap = 'gray')


for i in range(sarraster.shape[0] ):
    print(i),
    for j in range(sarraster.shape[1] ):

        # windows needs to fit completely in image
        if i > (contrastraster.shape[0] - 4) or j > (contrastraster.shape[0] - 4):
            continue

        # Define size of moving window
        glcm_window = sarraster[i-3: i+4, j-3 : j+4]
        # Calculate GLCM and textures
        glcm = greycomatrix(glcm_window, [1], [0],  symmetric = True, normed = True )

        # Calculate texture and write into raster where moving window is centered
        contrastraster[i,j]      = greycoprops(glcm, 'contrast')
        dissimilarityraster[i,j] = greycoprops(glcm, 'dissimilarity')
        homogeneityraster[i,j]   = greycoprops(glcm, 'homogeneity')
        energyraster[i,j]        = greycoprops(glcm, 'energy')
        correlationraster[i,j]   = greycoprops(glcm, 'correlation')
        ASMraster[i,j]           = greycoprops(glcm, 'ASM')
        glcm = None
        glcm_window = None

texturelist = {1: 'contrast', 2: 'dissimilarity', 3: 'homogeneity', 4: 'energy', 5: '
for key in texturelist:
    ax = plt.subplot(2,3,key)
    plt.axis('off')
    ax.set_title(texturelist[key])
    plt.imshow(eval(texturelist[key] + "raster"), cmap = 'gray')
    print(eval(texturelist[key]+"raster"))
plt.show()
```

3

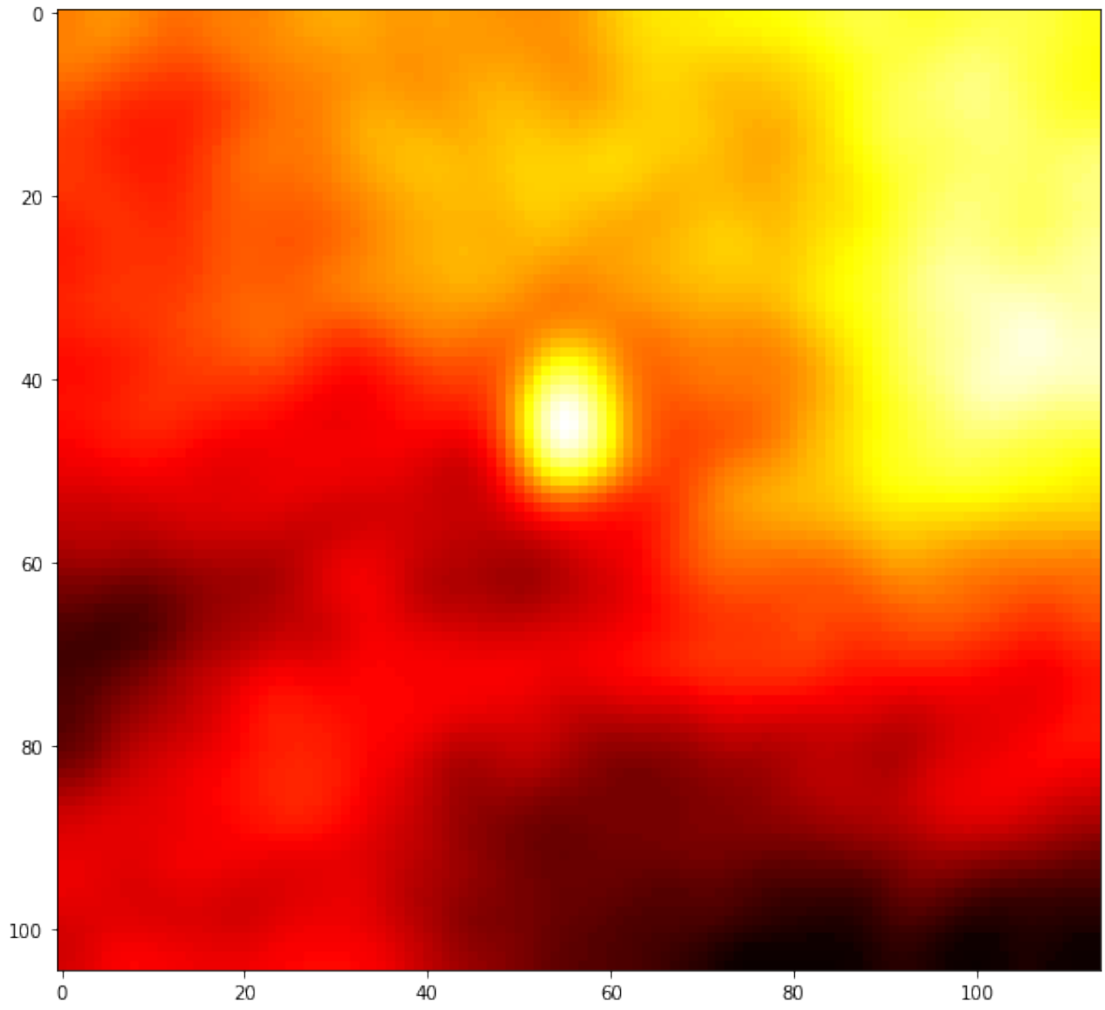
```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-78-119e100a18aa> in <module>()
     37 plt.axis('off')
     38 ax.set_title('Original Image')
---> 39 plt.imshow(sarraster, cmap = 'gray')
     40
     41


~\Anaconda3\lib\site-packages\matplotlib\pyplot.py in imshow(X, cmap, norm, aspect, int
   3203                         filternorm=filternorm, filterrad=filterrad,
   3204                         imlim=imlim, resample=resample, url=url, data=data,
-> 3205                         **kwargs)
   3206     finally:
   3207         ax._hold = washold


~\Anaconda3\lib\site-packages\matplotlib\__init__.py in inner(ax, *args, **kwargs)
   1853                         "the Matplotlib list!)" % (label_namer, func.__name__),
   1854                         RuntimeWarning, stacklevel=2)
-> 1855             return func(ax, *args, **kwargs)
   1856
   1857         inner.__doc__ = _add_data_doc(inner.__doc__,


~\Anaconda3\lib\site-packages\matplotlib\axes\_axes.py in imshow(self, X, cmap, norm, a
   5485                                 resample=resample, **kwargs)
   5486
-> 5487         im.set_data(X)
   5488         im.set_alpha(alpha)
   5489         if im.get_clip_path() is None:


~\Anaconda3\lib\site-packages\matplotlib\image.py in set_data(self, A)
    651         if not (self._A.ndim == 2
    652                 or self._A.ndim == 3 and self._A.shape[-1] in [3, 4]):
--> 653             raise TypeError("Invalid dimensions for image data")
    654
    655         if self._A.ndim == 3:


TypeError: Invalid dimensions for image data
```



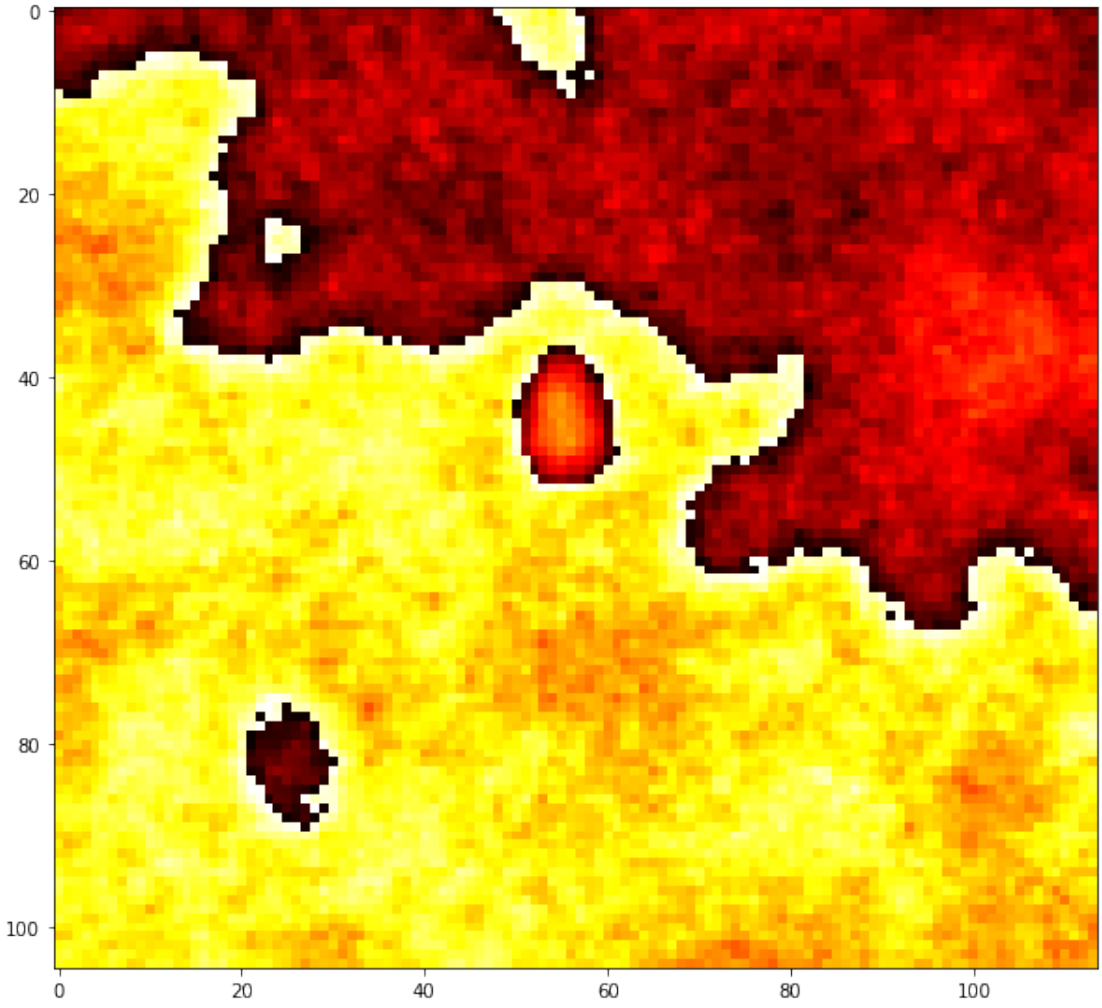

```
In [73]: from PIL import Image
         img = Image.open("IMG-0001-00001.jpg").convert('LA')
         img.save('greyscale.png')

In [89]: #plt.imshow(sarraster)
         print(sarraster)

[[[255 255 255 ...   0   0   0]
  [255 255 255 ...   0   0   0]
  [255 255 255 ...   0   0   0]
  ...
  [255 255 255 ... 255 255 255]
  [255 255 255 ... 255 255 255]
  [255 255 255 ... 255 255 255]]

 [[255 255 255 ... 255 255 255]
  [255 255 255 ... 255 255 255]
  [255 255 255 ... 255 255 255]
  ...
  [255 255 255 ... 255 255 255]
  [255 255 255 ... 255 255 255]
  [255 255 255 ... 255 255 255]]]


In [98]: import cv2
         crp_img = mpimg.imread('IMG-0003-00001.jpg')
         lum_crp_img = crp_img[:,:,0]
         crp_dst = cv2.fastNlMeansDenoisingColored(crp_img,None,10,10,7,21)
         plt.figure(figsize=(15, 15))
         plt.subplot(121),plt.imshow(lum_crp_img, cmap='hot')
         plt.subplot(122),plt.imshow(crp_dst[:,:,0], cmap='hot')
         plt.show()
```




```
In [139]: from scipy import ndimage
          lowpass = ndimage.gaussian_filter(crp_img, 3)
          plt.figure(figsize=(10, 10))
          plt.imshow(lowpass[:,:,0], cmap='hot')
          plt.show()
```




```
In [143]: gauss_highpass = crp_img - lowpass
          lowpass = ndimage.gaussian_filter(gauss_highpass, 3)
          plt.figure(figsize=(10, 10))
          plt.imshow(gauss_highpass[:,:,0], cmap='hot')
          plt.show()
```


0
20
40
60
80
100
0
20
40
60
80
100