BIOMEDIN 260/RAD260: Problem Set 2 - Digital Image Processing

Spring 2020

Group Members ¶

Person 1:

Ruben Krueger

Person 2:

Marc Huo

Welcome to Problem Set 2!

In this problem set, we will be exploring some fundamental concepts in digital image processing. We will be also be doing some math! Luckily, iPython notebooks have the additional ability of interpreting LaTeX ($\LaTeX$) for typesetting mathematical expressions. For example:

$$f(t) * g(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

This says that a convolution is defined as the integral of the product of the two functions after one is reversed and shifted. Note the word reversed, as in the kernel is mirrored along both axes. Here's another example:

$$y(x) = \frac{\sum_{i=2}^{i=3} x^i}{M}$$

This says nothing meaningful. ¯ \ (ツ)/¯

If you are not familiar with $\LaTeX$, don't worry! You can learn it very quickly just by doing this assignment. If you are looking for more practice, try some of these (http://www.personal.ceu.hu/tex/cookbook.html) examples out.

Problem 1: Linear Filtering (50 points)

a. Image matrix A consists of a small diamond surrounded by a layer of zeros. Compute the 2D convolution between image matrix A and filter kernel B. The output image will be the same shape as the input image, so you can assume a single zero padding layer on each side of the image.

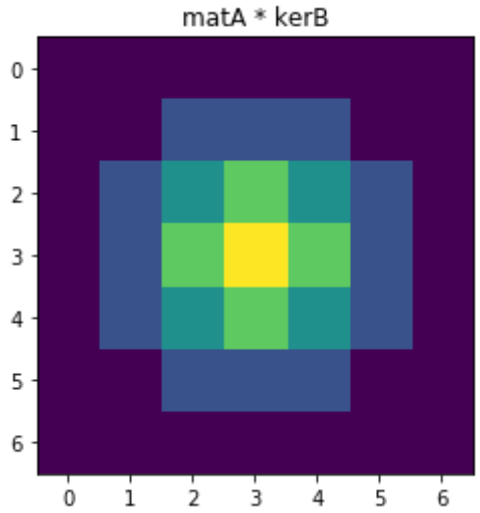
We have given you the image matrix A and the filter kernel B in the cell below. First, do this 2D convolution by hand (good midterm practice!). Then, check your answer in the cell below that. *Hint: you may find a particular function in the* `scipy.signal` *package helpful for this task.*

In [2]:
```
# from google.colab import files
# from IPython.display import Image


# #upload = files.upload()
```

In [3]:
```
import numpy as np
from scipy.signal import convolve
import matplotlib.pyplot as plt


matA = np.array([[0., 0., 0., 0., 0.],
                 [0., 0., 1., 0., 0.],
                 [0., 1., 0., 1., 0.],
                 [0., 0., 1., 0., 0.],
                 [0., 0., 0., 0., 0.]])

kerB = np.array([
                 [1., 1., 1.],
                 [1., 1., 1.],
                 [1., 1., 1.],
                ])
```

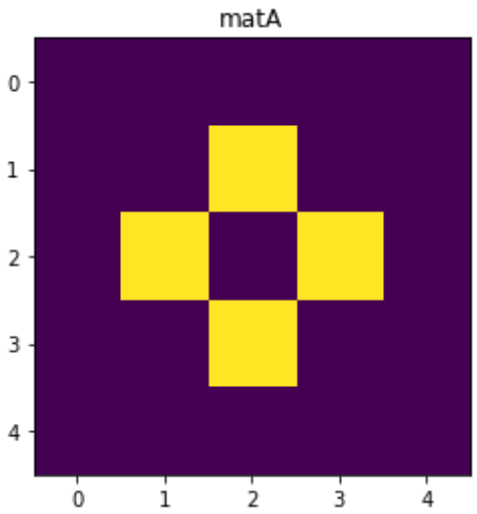
In [4]:
```
### WRITE CODE IN HERE. You can have up to 2 cells for this question, bu
t only one is required #######

# TODO: by hand :(

ab = convolve(kerB, matA)

plt.title('matA * kerB')
plt.imshow(ab)
################################################################################
##############################
```

Out[4]: <matplotlib.image.AxesImage at 0x1a1d140690>



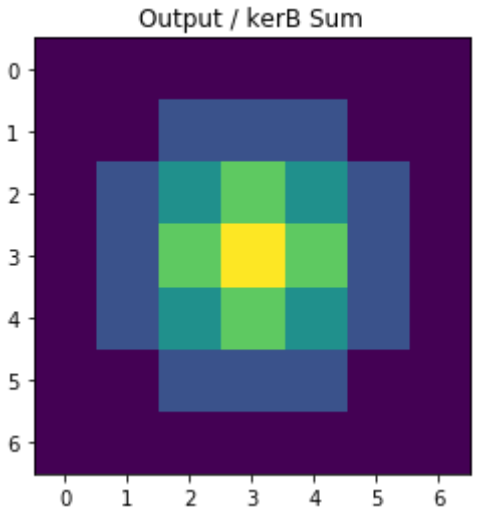
In [5]:
```
plt.title('matA')
plt.imshow(matA)
```

Out[5]: <matplotlib.image.AxesImage at 0x1a1e0f0e10>



In [6]:
```
plt.title('Output / kerB Sum')
plt.imshow(ab / np.sum(kerB))
```

Out[6]: <matplotlib.image.AxesImage at 0x1a1e2a4d10>



b. How does the output matrix compare to the original image matrix A? What type of filter is kernel B acting as? Alternatively, if you divide the output image by np.sum(kerB), what have you essentially done to the image?

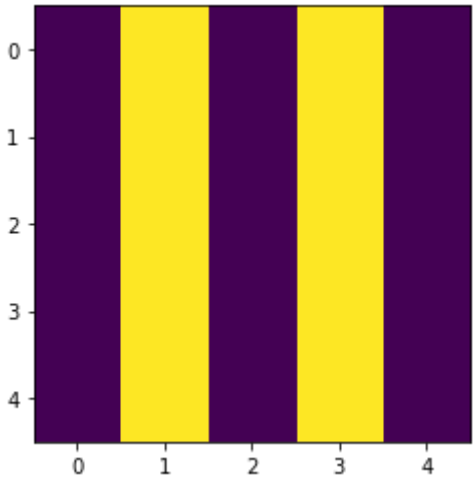
Answer: As one can see the plotted image above, the convolution of kernel B and A results in a blurred image. Kernel B is acting as a box blur or a spatial domain linear filter. Each pixel in the output image equals the average of its neighboring pixels in the input image. By dividing the output by np.sum(kerB), you have normalized all of the pixel values.

c. Let matrix O be the output from the 2D convolution between image matrix C and kernel D. Compute O.

The same way that we write out C and D for you, write out the output for us as a NumPy array, as in output the matrix O. You may choose to write out the full convolution (by zero-padding image matrix C and then convolving with kernel D) or simply crop the output matrix to be the same size as the input matrix. Try this by hand first for some more midterm practice.

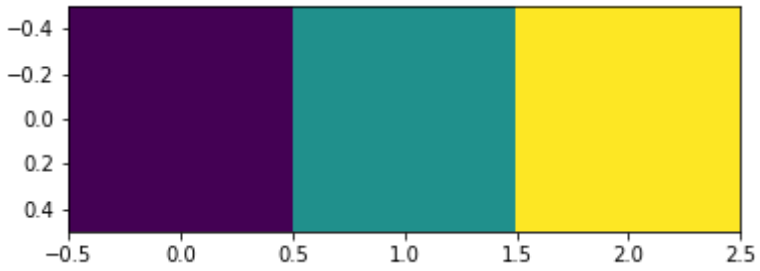
In [7]:
```
# Create image matrix C.
C = np.zeros((5, 5))
C[:,[1,3]] = 1

C
plt.imshow(C)
```

Out[7]: <matplotlib.image.AxesImage at 0x10a4affd0>



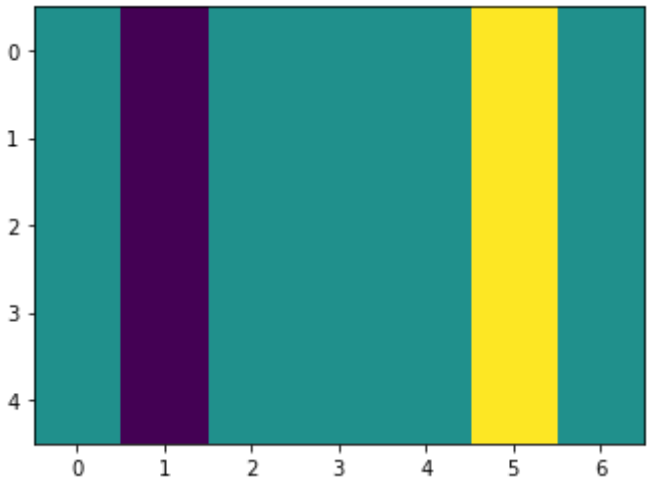
In [8]:
```
# Create filter kernel D.
D = np.array([-1, 0, 1])
D = np.expand_dims(D, 0)
plt.imshow(D)
D
```

Out[8]:
```
array([[-1,  0,  1]])
```



In [21]:
```
## WRITE MATRIX O IN HERE ######

O = convolve(D, C)

plt.imshow(O)
################################
```

Out[21]:
```
<matplotlib.image.AxesImage at 0x7f213c693e80>
```



d. How does the output matrix O compare to the original image matrix C? In other words, what kind of a filter is kernel D?

Answer: Kernel D is a Prewitt operator which calculates the gradient of image intensity. The output matrix O gives the direction of increase from light to dark and the rate of change. It shows how likely a part of the input image represents an edge and its orientation. In this case, we have a prewitt operator in the horizontal direction, which tells us the horizontal derivatives approximations. Kernel D is an edge detector.

e. Convolve image matrix C with kernel E to get matrix F. How does F differ from E? Again, try this by hand first for some more midterm practice.

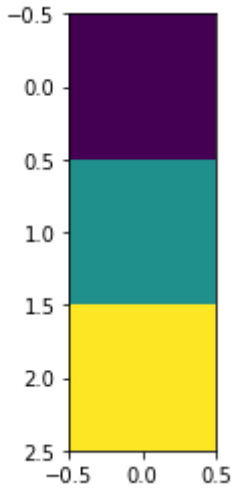
In [0]:
```
```

In [22]:
```
# Create filter kernel E.
E = D.T
E
```

Out[22]:
```
array([[-1],
       [ 0],
       [ 1]])
```

In [23]:
```
plt.imshow(E)
```

Out[23]:
```
<matplotlib.image.AxesImage at 0x7f213c8339b0>
```

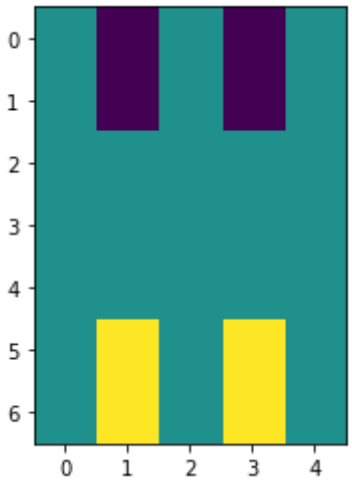


In [24]:
```
# Convolve C and E to make F.#

F = convolve(E,C)

##############################
F
```

Out[24]:
```
array([[ 0., -1.,  0., -1.,  0.],
       [ 0., -1.,  0., -1.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  1.,  0.],
       [ 0.,  1.,  0.,  1.,  0.]])
```

In [25]:
```
plt.imshow(F)
```

Out[25]:
```
<matplotlib.image.AxesImage at 0x7f213c6d3d30>
```



Answer: Similar to the above question, this again is an edge detection kernel. F is diffferent from E, because it is a transposed central finite difference. The edge detection kernel resembles a Sobel operator.

f. Import your favorite image as a NumPy array. Try to keep it reasonably sized (like 256-by-256) to avoid overly long computation times. If you don't have a favorite image, you can create a matrix of zeros and ones to form shapes (like a disk or rectangle).

Apply some additive white Gaussian noise to your image. There are many ways to do this, and you should choose your favorite method! Next, try to smooth out your noisy image using your favorite linear smoothing filter.

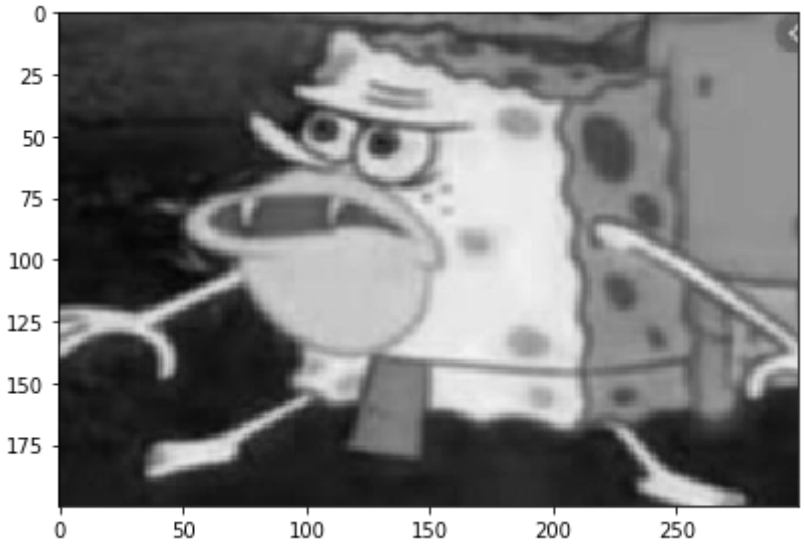
Show all three images (the original image, the noisy image, and the smoothed noisy image) side-by-side using `plt.subplots()`.

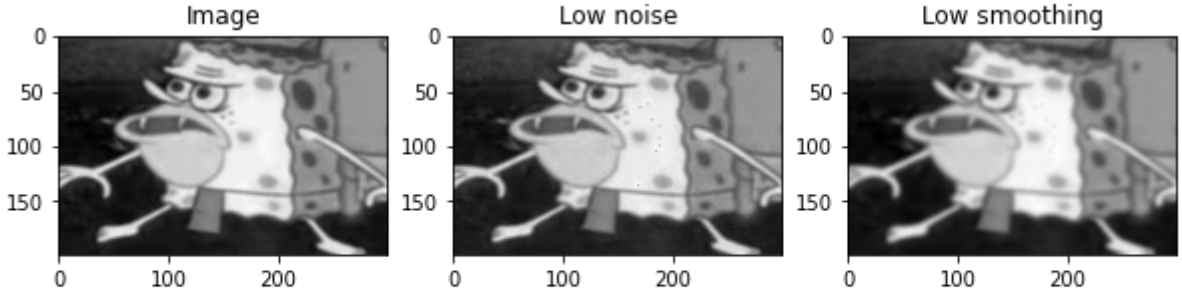
Play around with the level of noise as well as the smoothing filter parameters. At what level of noise can you no longer distinguish the main features of the original image?

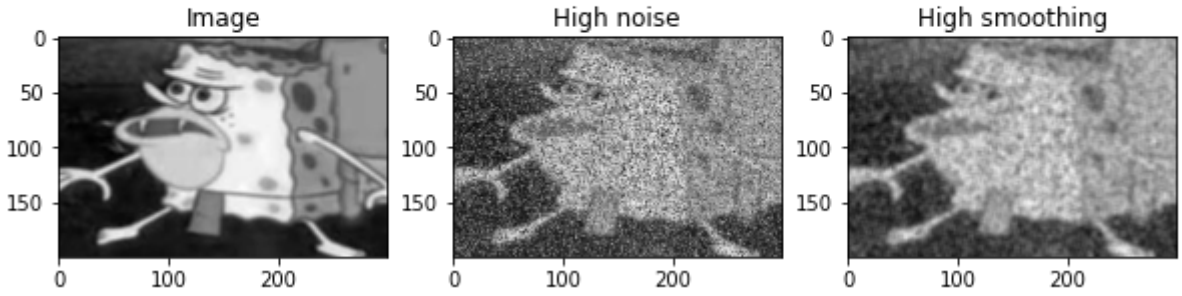
As the level of noise increases, how should you adjust your smoothing filter parameters? What is the main downside to smoothing images?

In [26]:
```
import cv2
from skimage.color import rgb2gray
from skimage import io

img = io.imread('sponge.jpg')[:,:,0]
plt.gray()
io.imshow(img)
w = img.shape[0]
h = img.shape[1]
noise1 = cv2.randn(np.empty((w,h), np.uint8),(0),(7))
low_noise = img + noise1
kernel1 = np.ones((5,5),np.float32)/5
kernel2 = np.ones((5,5),np.float32)/25
low_smooth = cv2.filter2D(low_noise,-1,kernel2)


noise2 = cv2.randn(np.empty((w,h), np.uint8),(0),(100))
high_noise = img + noise2
high_smooth = cv2.filter2D(high_noise,-1,kernel2)


fig, ((ax1, ax2, ax3), (ax4, ax5, ax6)) = plt.subplots(2, 3, figsize=(10
,10) )
ax1.set_title('Image')
ax2.set_title('Low noise')
ax3.set_title('Low smoothing')
ax4.set_title('Image')
ax5.set_title('High noise')
ax6.set_title('High smoothing')
ax1.imshow(img)
ax2.imshow(low_noise)
ax3.imshow(low_smooth)
ax4.imshow(img)
ax5.imshow(high_noise)
ax6.imshow(high_smooth)
```

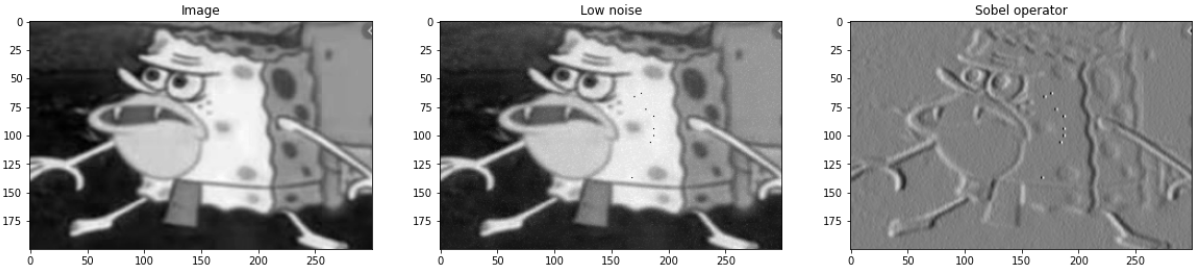
Out[26]: <matplotlib.image.AxesImage at 0x7f2139c585f8>







In [27]:
```
import scipy
from scipy import ndimage
from skimage import color

fig = plt.figure(figsize=(20,20))
ax1 = fig.add_subplot(131)
ax2 = fig.add_subplot(132)
ax3 = fig.add_subplot(133)  #img = img.astype(np.int16)
result = ndimage.filters.sobel(color.rgb2grey(low_noise.astype(np.int16
)))
ax1.imshow(img)
ax2.imshow(low_noise)
ax3.imshow(result)
ax1.set_title('Image')
ax2.set_title('Low noise')
ax3.set_title('Sobel operator')
plt.show()
```

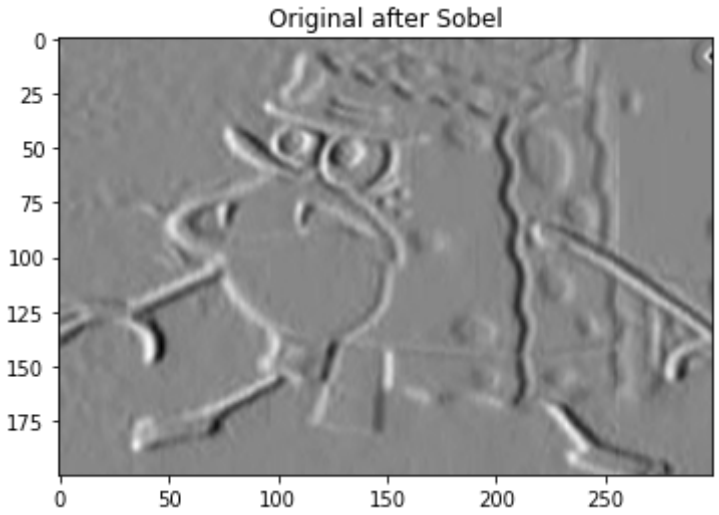


Answer: Smoothing images does not work well with "salt and pepper noise." As one can see, marks are created in the smoothed image from where the noise was. It does not take much noise to lose the main features of the image. In the high smoothing case, it was Gaussian noise with a standard deviation of 100. Smoothing imsage can lose major features of the image. As the level of noise increases, the smoothing parameters should become more aggressive.

g. Try applying a linear edge detection kernel (such as a central finite difference kernel or Sobel operator) to both your original image and the noisy image. How does it do?

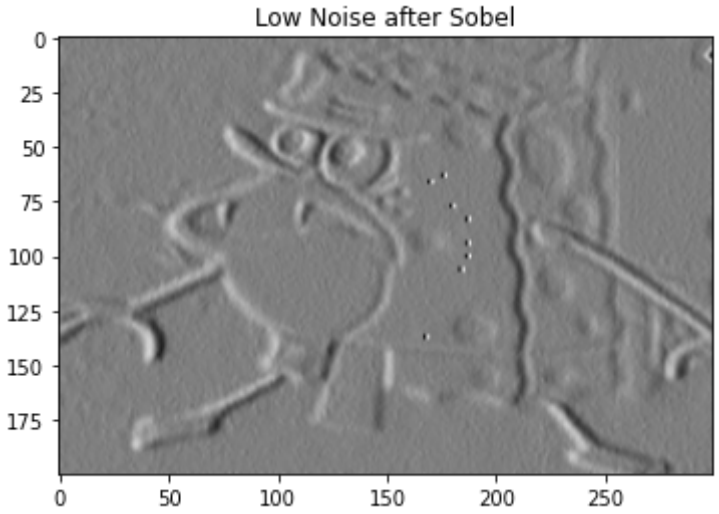
In [0]:
```
original = io.imread('sponge.jpg')[:,:,0]
original_sobel = ndimage.filters.sobel(color.rgb2grey(original.astype(np
.int16)))
plt.title("Original after Sobel")
plt.imshow(original_sobel)
```

Out[0]: <matplotlib.image.AxesImage at 0x7f2137c7ff98>



In [36]:
```
noisy_sobel = ndimage.filters.sobel(color.rgb2grey(low_noise.astype(np.i
nt16)))
plt.title("Low Noise after Sobel")
plt.imshow(noisy_sobel)
```

Out[36]: <matplotlib.image.AxesImage at 0x7f2139d01f98>



Answer: The original image after sobel edge detection seemed to perform better with darker and more refined edges. The low noise image after sobel edge detection had issues with looking a bit washed out as well as having 8 cases of artifacts.

h. Assume we are working with an 1D input image $I(x)$ (though this could be easily generalized to 2D or 3D). Suppose we create a new type of filter defined as:

$$I'(x) = \frac{\sum_i I(x_i) f(x_i - x)}{\sum_i f(x_i - x)}$$

where $I'(x)$ is the output image, $f(x)$ is an zero-mean Gaussian function, and i indexes over pixels that are close to x.

What kind of filter is this? For example, is it linear or non-linear? Is it a low-pass filter or a high-pass filter? (Hint: you may have seen a similar type of filter in this problem.)

Answer: The above is a smoothing low-pass filter, where it averages pixel values with the coefficient $\sum_i I(x_i)$. This is linear.

i. How does this filter perform at edges and boundaries within the image? Why? Write an expression for a new filter that attempts to address this problem. Is your new filter still linear?

Hint: introduce a function $g(\ldots)$ into the above expression that encourages the preservation of edges.

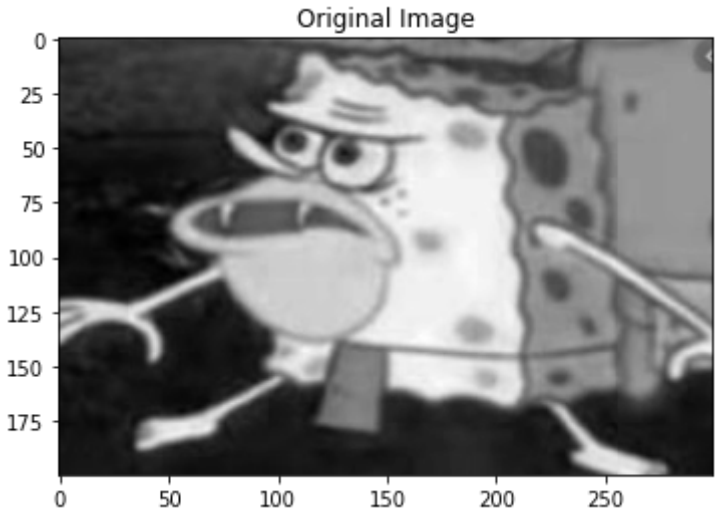
$$I'(x) = \frac{\sum_i I(x_i) f(x_i - x) g(\ldots)}{\sum_i f(x_i - x) g(\ldots)}$$

Answer: The issue with the previous filter was that it was inadequate for preserving our edges as well as defining the boundaries within the image. Let the function $g(\ldots)$ be the function $g(x_i) = argmin[f(x_i)]$. This allows us to take the median of each window that both preserves our edges as well as smooths.

j. Linear smoothing filters are an efficient way to reduce the presence of noise in the image at the cost of lowering image resolution and blurring out edges. Several non-linear filters were developed to both reduce noise while attempting to preserve the high frequency edges in the image. Choose a non-linear de-noising filter and apply it to your noisy image from part *f*. How well does it do compared to the image that was de-noised using a linear smoothing filter?

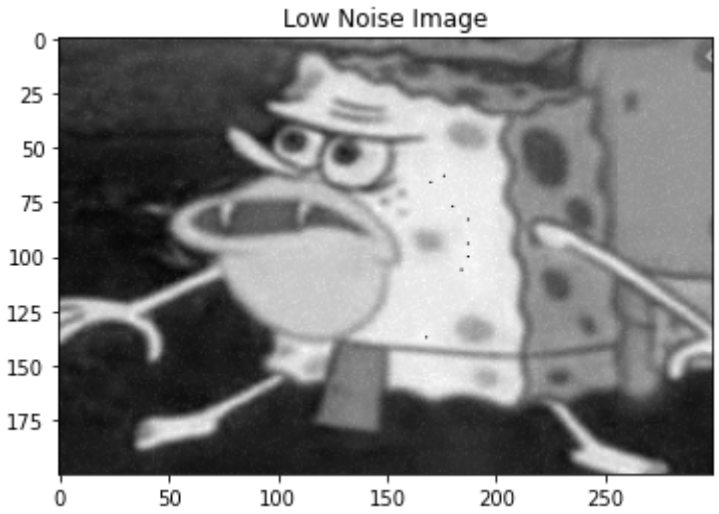
In [41]:
```
plt.title("Original Image")
plt.imshow(img)
```

Out[41]: <matplotlib.image.AxesImage at 0x7f2137a6b780>



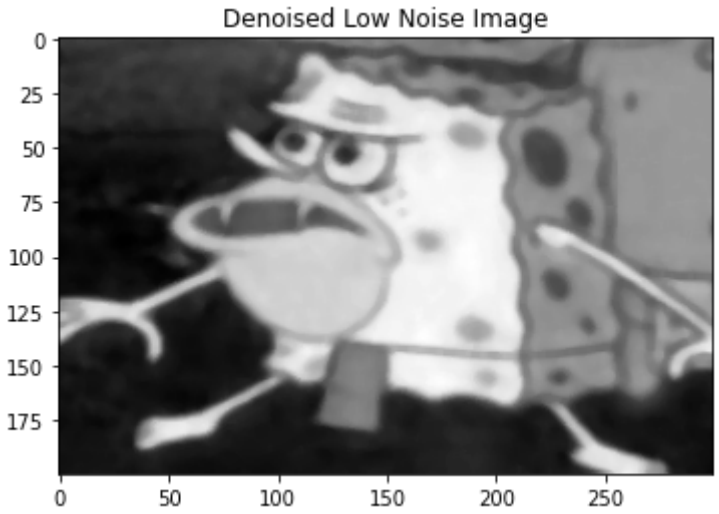
In [45]:
```
plt.title("Low Noise Image")
plt.imshow(low_noise)
```

Out[45]: <matplotlib.image.AxesImage at 0x7f21378854e0>



In [44]:
```
### WRITE CODE IN HERE. ##############################################
##############################
denoised_low_noise = ndimage.filters.median_filter(low_noise, size = 5)
plt.title("Denoised Low Noise Image")
plt.imshow(denoised_low_noise)

######################################################################
##############################
```

Out[44]: <matplotlib.image.AxesImage at 0x7f2137916470>



Answer: The median filter that was used performed quite well in comparison to the linear smoothing filter. We can see that the edges hold their integrity and noise is reduced.

Problem 2: Geometric Features (20 points)

As you may recall from class, the geometric features of a tumor can sometimes reveal significant information about whether it is benign or malignant. Radiologists frequently assess the geometry of lesions in mammograms when attempting to diagnose breast cancer (https://www.sciencedirect.com/science/article/pii/S2211568413003872).

In this exercise, we will attempt to explore some quantitative methods of differentiating between certain lesion shapes.

Download the two mammograms from Canvas. `LEFT_CC_BENIGN.tif` depicts a cranial-caudal view (looking down on the breast from the patient's perspective) of the left breast. There is a benign lesion present in the mammogram. Can you locate it?

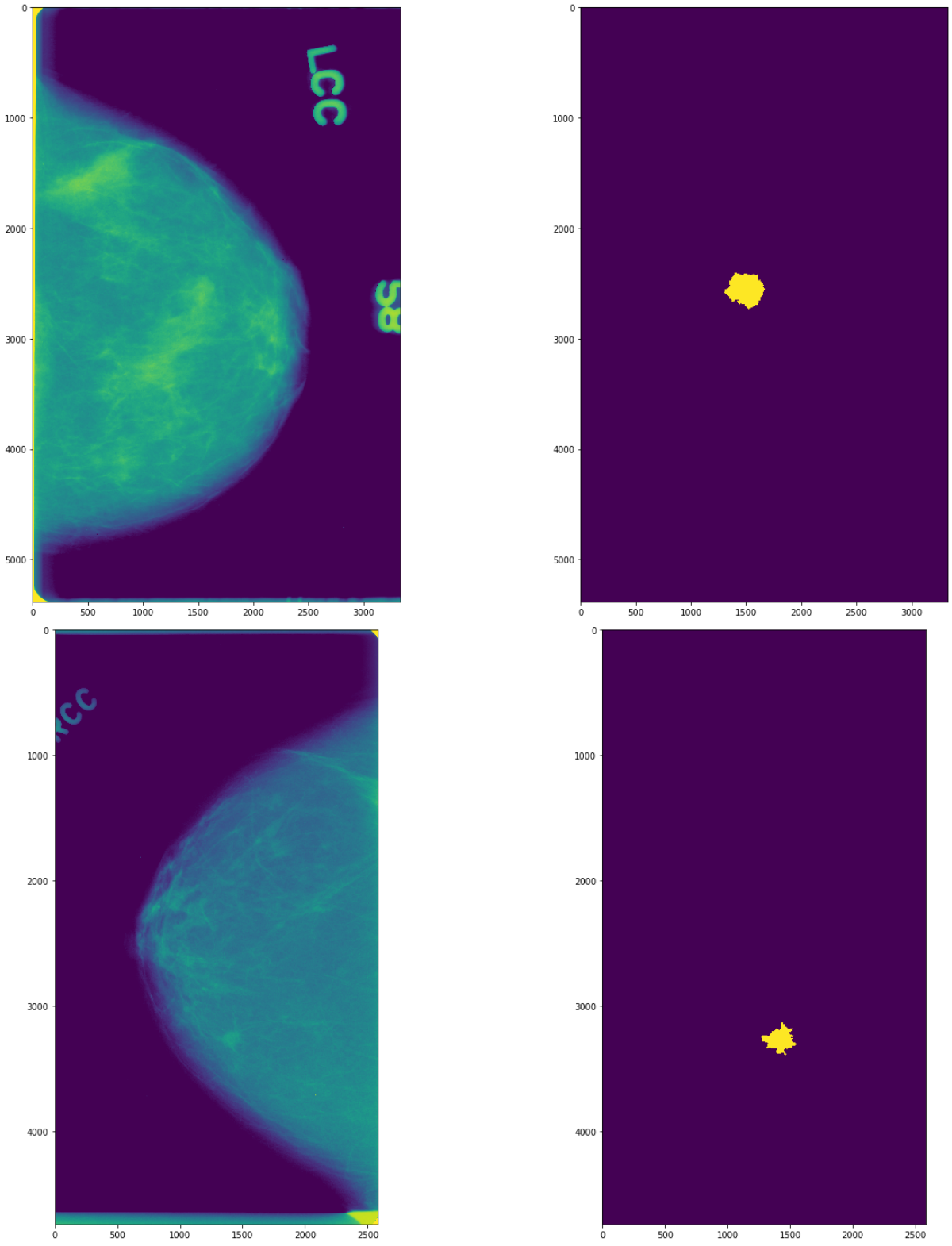
If you are having difficulty locating the lesion, try seeing if you can spot it in the mask `LEFT_CC_BENIGN_MASK.tif` , which consists of a rough segmentation of the lesion. `RIGHT_CC_MALIGNANT.tif` and `RIGHT_CC_MALIGNANT_MASK.tif` are the mammogram and corresponding lesion segmentation of a different patient, but this mammogram depicts a breast with a **malignant** tumor.

a. Start by loading these two mammograms and their corresponding masks as image matrices. Plot them on a single figure (using a 2-by-2 grid of subplots). You should use reasonable **x** and **y** limits in your figures to get a reasonable depiction of your lesion segmentations.

Qualitatively describe the shape differences between the two lesions. What types of shapes and margins are indicative of malignant tumors?

In [0]:
```
#img = color.rgb2gray(img_benign)
#plt.imshow(img)
```

In [9]:
```
import pydicom as dcm
import matplotlib.pyplot as plt
from PIL import Image
from skimage.filters import threshold_otsu
from skimage import io, color
%matplotlib inline

## Load mammograms showing benign and malignant tumors.

img = Image.open('LEFT_CC_BENIGN.tif')
img.save('LEFT_CC_BENIGN.png')

img_benign = io.imread('LEFT_CC_BENIGN.png', as_gray=True)
mask_benign = plt.imread('LEFT_CC_BENIGN_MASK.tif')

img_malignant = plt.imread('RIGHT_CC_MALIGNANT.tif')
mask_malignant = plt.imread('RIGHT_CC_MALIGNANT_MASK.tif')

## Plotting code here.
fig = plt.figure(figsize=(20, 20))

### WRITE CODE IN HERE. You can have up to 2 cells for this question, bu
t only one is required #######


ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222)
ax3 = fig.add_subplot(223)
ax4 = fig.add_subplot(224)

ax1.imshow(img_benign)
ax2.imshow(mask_benign)
ax3.imshow(img_malignant)
ax4.imshow(mask_malignant)


#########################################################################
##############################

fig.tight_layout() # Fixes spacing between subplots.
```



Answer: To delineate between benign and malignant tumors, the most important features to look for in mammograms is irregularities in both shapes and margins. Round and circumscribed oval shaped masses are usually benign, as shown in the benign mammogram above. In contrast, jagged or irregularly shaped masses are more likely to be malignant, which is shown by the malignant mammogram. Furthermore, circumscribed tumors are usually benign tumors while malignant tumors are usually non-circumscribed. The margins, as seen in the malignant mammogram, are circumscribed while the non-malignant mammogram is not.

b. Next, try computing some basic geometric features of the two lesions. Which features are the most significant for differentiating between them? Is this what you expected? The `regionprops()` function in the `skimage` library will automatically compute several geometric features for you, but you might choose to code up some of your own. If you don't trust the provided segmentations, feel free to create your segmentation method or modify the provided ones to get a better estimate of the lesion shapes! Finally, it could be interesting to apply some edge-detection filters to the mammograms to better emphasize the margins.

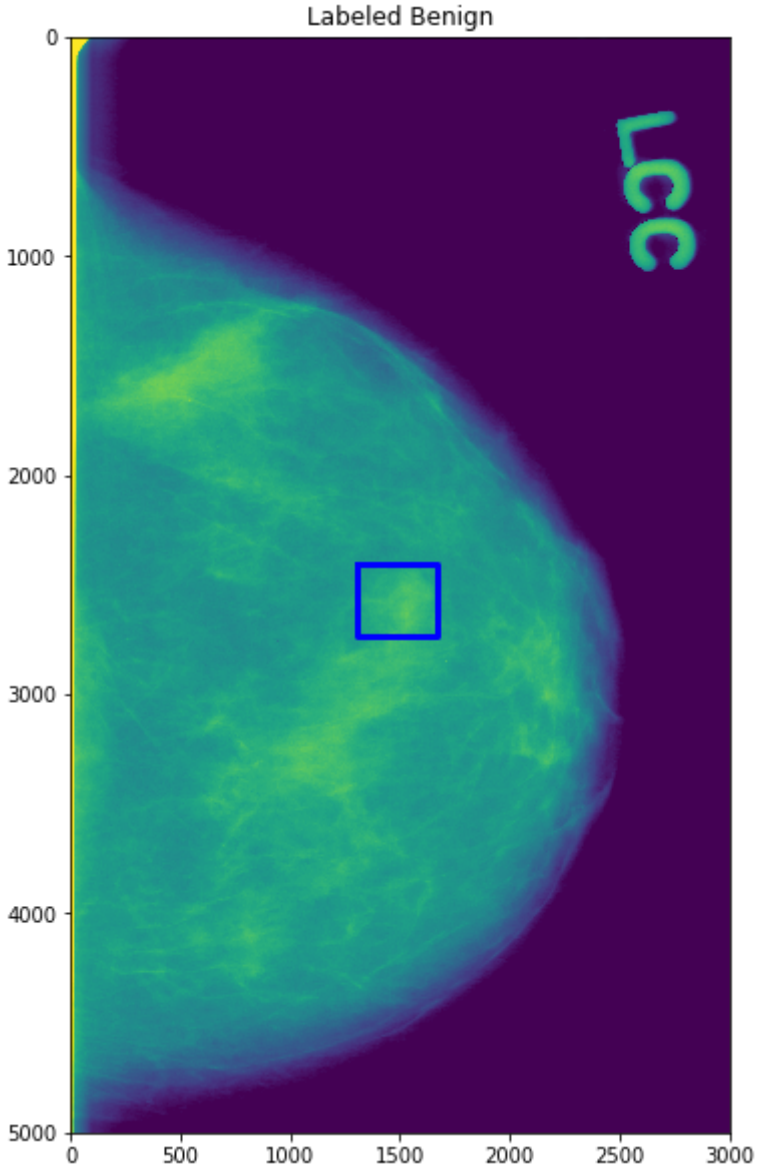
You can also test your approach on other mammograms from a larger dataset (~20 GB) (https://www.dropbox.com/sh/y0k6wokrq5fibpa/AAC43nULYqhI_cLu555qy2nla?dl=0). You will need these mammograms for the next problem set, so there's no harm in downloading the images now!

There is no single correct answer for this exercise, and you should definitely try several different approaches and explore!

Answer: The convex area of the tumor is useful for differentiating between the two tumors, both malignant and benign. We can use that curvature to then determine whether a tumor is malignant or benign. Similarly, after using regionprops(), we were able to derive the areas of each of the tumors, which is beneficial in helping us differentiate between a large vs. small tumor. As shown below, the benign tumor has convex area 88,751 while the malignant tumor has convex area 49,385. We can then divide the convex area by the total area to get a measure of solidity and judge the irregularity of the tumors. We see that the solidity of the benign tumor is around 0.89 while the solidity of the malignant tumor is around 0.72, showing that the malignant tumor is highly irregular in shape

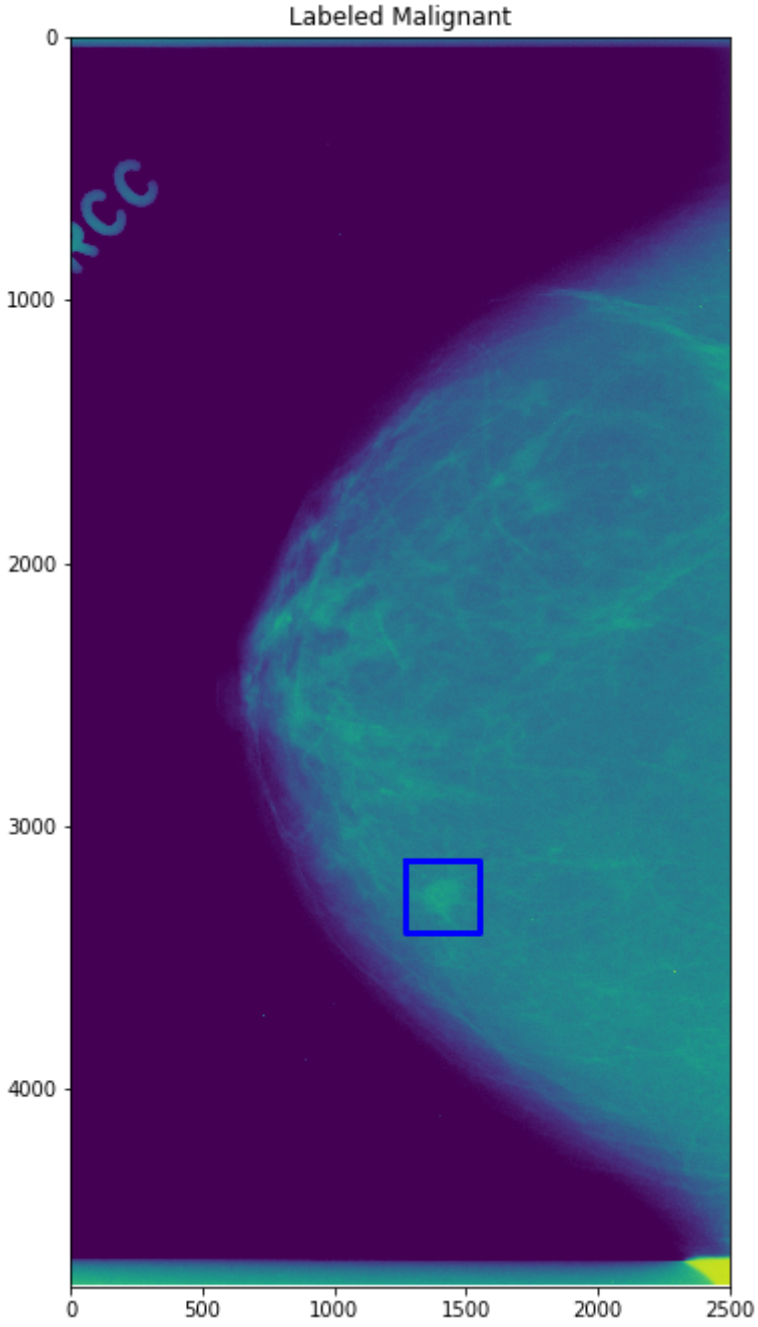
In [10]:
```
from skimage.measure import regionprops, regionprops_table, label
import matplotlib.pyplot as plt

img = plt.imread("LEFT_CC_BENIGN.tif", format = "RGB")
benign_mask = plt.imread("LEFT_CC_BENIGN_MASK.tif")

labels = label(benign_mask)
props = regionprops(labels)


fig = plt.figure(figsize=(20,20))
ax1 = fig.add_subplot(131)
ax1.set_title('Labeled Benign')


for prop in props:
  if prop.area > 1:
    minr, minc, maxr, maxc = prop.bbox
    box_x = (minc, maxc, maxc, minc, minc)
    box_y = (minr, minr, maxr, maxr, minr)
    ax1.plot(box_x, box_y, '-b', linewidth = 3)
    print("Benign tumor has convex area:", prop.convex_area)
    print('Label: {} >> Convex Area: {}'.format(prop.label, prop.convex_
area))
    print('Label: {} >> Area / Convex Area: {}'.format(prop.label, prop.
area/prop.convex_area))

ax1.axis((0, 3000, 5000, 0))
ax1.imshow(img)
plt.show()
```

```
Benign tumor has convex area: 88751
Label: 1 >> Convex Area: 88751
Label: 1 >> Area / Convex Area: 0.8917984022715237
```



In [11]:
```
from skimage.measure import regionprops, regionprops_table, label
import matplotlib.pyplot as plt

img = plt.imread("RIGHT_CC_MALIGNANT.tif", format = "RGB")
malignant_mask = plt.imread("RIGHT_CC_MALIGNANT_MASK.tif")

labels = label(malignant_mask)
props = regionprops(labels)

fig = plt.figure(figsize=(20,20))
ax1 = fig.add_subplot(131)
ax1.set_title('Labeled Malignant')

for prop in props:
  if prop.area > 1:
    minr, minc, maxr, maxc = prop.bbox
    box_x = (minc, maxc, maxc, minc, minc)
    box_y = (minr, minr, maxr, maxr, minr)
    ax1.plot(box_x, box_y, '-b', linewidth = 3)
    print("Malignant tumor has convex area:", prop.convex_area)
    print('Label: {} >> Convex area: {}'.format(prop.label, prop.convex_
area))
    print('Label: {} >> Area / Convex Area: {}'.format(prop.label, prop.
area/prop.convex_area))

ax1.axis((0, 2500, 4750, 0))
ax1.imshow(img)
plt.show()
```

```
Malignant tumor has convex area: 49385
Label: 2 >> Convex area: 49385
Label: 2 >> Area / Convex Area: 0.7189834970132631
```



Problem 3: Textural Features (30 points)

a. Compute the gray level co-occurrence matrix (GLCM) of image matrix A below using an offset of $\Delta i = 1$ and $\Delta j = 0$. Do not make any assumptions about located values outside image matrix A (only count co-occurrences of pixels located within the image boundary). Try to do this by hand first, but feel free to check your answer using your preferred method for computing GLCMs (*Hint: see* `skimage.feature`*'s* `greycomatrix` *function*). Also, please make sure your GLCM is a reasonable size. You don't need to show portions of the GLCM that are mainly comprised of zeroes.

$$A = \begin{pmatrix} 2 & 1 & 2 & 1 & 2 \\ 1 & 2 & 1 & 2 & 1 \\ 2 & 1 & 2 & 1 & 2 \\ 1 & 2 & 1 & 2 & 1 \\ 2 & 1 & 2 & 1 & 2 \end{pmatrix}$$

Answer:

GLCM(A) = See Below

In [13]:
```
from skimage.feature import greycomatrix, greycoprops

# Creating matrix A.
A = np.ones((25))
A[::2] = 2
A = np.reshape(A, (5,5))
A = np.uint8(A)

print("A:")
print(A)
print()

### WRITE CODE IN HERE. You can have up to 2 cells for this question, bu
t only one is required #######

# Compute the GLCM for matrix A.


glcm_a = greycomatrix(A, [1],  [0])
# Compute 2 GLCMs: One for a 1-pixel offset to the right, and one for a
 1-pixel offset upwards.
# greycomatrix(image, [1], [0, np.pi/2], levels=4)

print('GLCM(A):')
print(glcm_a)
# Print it out here.

##########################################################################
##############################
```

```
A:
[[2 1 2 1 2]
 [1 2 1 2 1]
 [2 1 2 1 2]
 [1 2 1 2 1]
 [2 1 2 1 2]]

GLCM(A):
[[[[ 0]]

  [[ 0]]

  [[ 0]]

  ...

  [[ 0]]

  [[ 0]]

  [[ 0]]]


 [[[ 0]]

  [[ 0]]

  [[10]]

  ...

  [[ 0]]

  [[ 0]]

  [[ 0]]]


 [[[ 0]]

  [[10]]

  [[ 0]]

  ...

  [[ 0]]

  [[ 0]]

  [[ 0]]]


 ...


 [[[ 0]]
```

```
 [[ 0]]

 [[ 0]]

 ...

 [[ 0]]

 [[ 0]]

 [[ 0]]]


[[[ 0]]

 [[ 0]]

 [[ 0]]

 ...

 [[ 0]]

 [[ 0]]

 [[ 0]]]


[[[ 0]]

 [[ 0]]

 [[ 0]]

 ...

 [[ 0]]

 [[ 0]]

 [[ 0]]]]
```

b. Try computing the GLCM of matrix A using a different offset of $\Delta i = 2$ and $\Delta j = 0$ as well as with an offset of $\Delta i = 1$ and $\Delta j = -1$. How does it compare to your GLCM from part a?

In [0]:
```
### WRITE CODE IN HERE. You can have up to 2 cells for this question, bu
t only one is required #######

# Compute the GLCM for matrix A with an offset of (2, 0).

print('GLCM(A) - offset of (2, 0):')
# Print it out here.

glcm_2 = greycomatrix(A, [2],  [0])


print(glcm_2)

# Compute the GLCM for matrix A with an offset of (1, -1).

print('GLCM(A) - offset of (-1, -1):')

glcm_3 = greycomatrix(A, [np.sqrt(2)],  [-np.pi/4])
print(glcm_3)

# Print it out here.

##########################################################################
##############################
```

```
GLCM(A) - offset of (2, 0):
[[[[0]]

  [[0]]

  [[0]]

  ...

  [[0]]

  [[0]]

  [[0]]]


 [[[0]]

  [[7]]

  [[0]]

  ...

  [[0]]

  [[0]]

  [[0]]]


 [[[0]]

  [[0]]

  [[8]]

  ...

  [[0]]

  [[0]]

  [[0]]]


 ...


 [[[0]]

  [[0]]

  [[0]]

  ...
```

```
  [[0]]

  [[0]]

  [[0]]]


 [[[0]]

  [[0]]

  [[0]]

  ...

  [[0]]

  [[0]]

  [[0]]]


 [[[0]]

  [[0]]

  [[0]]

  ...

  [[0]]

  [[0]]

  [[0]]]]
GLCM(A) - offset of (-1, -1):
[[[[0]]

  [[0]]

  [[0]]

  ...

  [[0]]

  [[0]]

  [[0]]]


 [[[0]]

  [[8]]

  [[0]]
```

```
 ...

 [[0]]

 [[0]]

 [[0]]]


[[[0]]

 [[0]]

 [[8]]

 ...

 [[0]]

 [[0]]

 [[0]]]


...


[[[0]]

 [[0]]

 [[0]]

 ...

 [[0]]

 [[0]]

 [[0]]]


[[[0]]

 [[0]]

 [[0]]

 ...

 [[0]]

 [[0]]

 [[0]]]
```

```
[[[0]]

 [[0]]

 [[0]]

 ...

 [[0]]

 [[0]]

 [[0]]]]
```

Answer: As one can see, diagonal directions are distinct.

c. Suppose you are working on a computer-aided detection algorithm for diagnosing hepatocellular carcinoma (HCC). You have written some code to perform textural analysis on several CT scans of the liver, and the first step in your processing pipeline involves computing GLCMs of various sub-sections of each CT scan to try and differentiate between healthy liver tissue and potential lesions.

Your friend is attempting to set up a similar computer-aided detection algorithm for diagnosing HCC, except he/she is planning to do the diagnosis based on a series of ultrasound B-mode images rather than CT scans. You lend your friend your textural analysis code, but it doesn't work well at all on the ultrasound images. It has difficulty differentiating between normal tissue and abnormal findings, even in the most obvious of cases.

Why doesn't the code work well? How might you and your friend go about fixing this issue? Briefly justify your answers.

Answer: Ultrasounds form images from continuous data ranges, whereas CT is discrete and normalized to Hounsfield units. Because GLCMs require discrete values, the code won't work. To fix this, the CT would need to be discretized.

d. Previously, we have computed GLCMs over the entire image, which is generally not useful. Any subtle changes in lesion texture could make a significant difference in terms of the radiologist's diagnosis, but it might only result in a minimal change in the image's GLCM. A simple solution is to break the image down into chunks and compute a localized GLCM of each image chunk.

Try doing this on one of the provided mammograms. Iterate through the full mammogram and extract overlapping chunks (similar to what you would do when computing a 2D convolution), then compute the GLCM of each chunk. From each GLCM, compute your favorite Haralick texture feature and save it to a separate matrix. If you've done this correctly, you should be able to produce a textural feature map that is about the same size as your image. Plot this texture map alongside the original image. Does it look like what you expect? Why or why not?

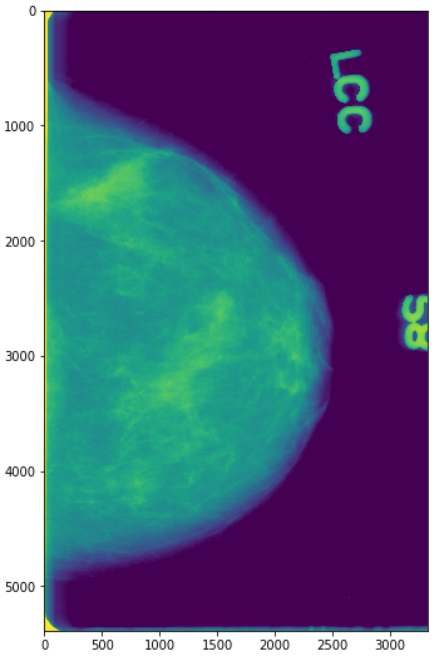
Tips:

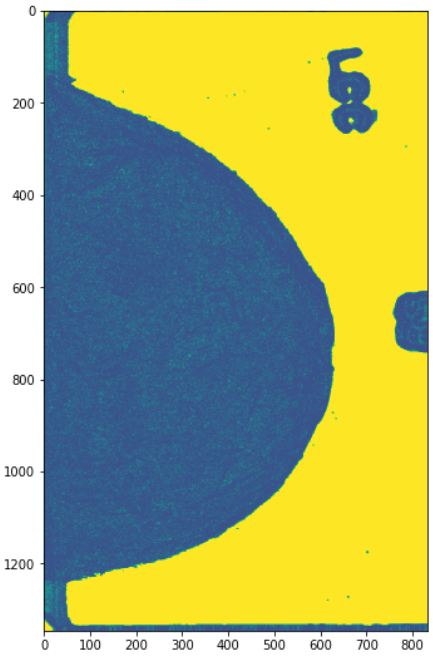
1. This could take really long ~ > 10 mins. If your code is taking too long to run, try downsampling the mammogram to a more reasonable size.
2. Don't worry too much about how the kernel behaves near the edges of the image, since these image regions are generally not as important.
3. You can choose whatever offsets you'd like when computing the GLCM. It is typical for researchers to make their textural features rotation-invariant by averaging features from multiple GLCMs computed using offsets oriented in all directions, and you can choose to do this if you want.
4. Explore different kernel sizes!

Answer: This did not do what we expected. We expected more dilineations of lesion.

In [32]:
```
### WRITE CODE IN HERE. You can have up to 3 cells for this question ###
############################

from skimage.transform import rescale, resize

def localized_features(img_name='LEFT_CC_BENIGN.png', feature='energy'):

    offset = [1]
    angle = [np.pi/4]

    img = io.imread(img_name, as_gray=True)

    img = resize(img, (img.shape[0] // 4, img.shape[1] // 4))

    img = 255 * img

    img = img.astype(np.uint8)


    dx, dy = 5, 5

    matrix = np.zeros((img.shape[0], img.shape[1]))

    for i in range(img.shape[0] - dx):
        for j in range(img.shape[1] - dy):
            A = img[i:i+dx,j:j+dy:]
            glcm = greycomatrix(A, offset,  angle)
            measure = greycoprops(glcm, feature)
            matrix[i:i+dx,j:j+dy:] = measure


    return matrix



texture_map = localized_features()

fig = plt.figure(figsize=(20, 20))

ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222)

ax1.imshow(img_benign)
ax2.imshow(texture_map)


############################################################################
############################
```

Out[32]: <matplotlib.image.AxesImage at 0x1a255157d0>





Problem 4: Interpolation (10 pts)

We have an isotropic 2D pixel that is 1mm on each side. The values at each corner of the square are given by variables A through D. Derive the equation for the linearly interpolated value along the diagonal from C to B as a parametric function of t where:

$t = 0$ at C

$t = 1$ at B

$x = y = t$

For example, if this were a line containing only A and B where $t = 0$ at A and $t = 1$ at B, then the linearly interpolated value along that line would be:

$$f(t, A, B) = (1 - x)A + Bx = (1 - t)A + Bt$$

Find $f(t, A, B, C, D)$.

Answer:

$$
\begin{aligned}
f_1(t, A, B) &= (1-t)A + Bt \\
f_2(t, C, D) &= C(\frac{1-x}{1-0}) + D(\frac{x-0}{1-0}) \\
&= C(1-x) + Dx \\
&= C(1-t) + Dt \\
f(t, A, B, C, D) &= f_1(\frac{y-0}{1-0}) + f_2(\frac{1-y}{1-0}) \\
&= f_1(t) + f_2(1-t)
\end{aligned}
$$