sx2261_hw2

September 25, 2019

1 COMS 4731 Computer Vision – Homework 2

- This homework contains the following components:
  - **Problem 1: Image Denoising (40 points)**
    * Implement a mean filter using "for" loop.
    * Implement the `convolve_image` function.
    * Implement a mean filter using a filter matrix.
    * Implement a Gaussian filter.
  - **Problem 2: Edge Detection (30 points)**
    * Implement a delta filter.
    * Implement a Laplacian filter.
  - **Problem 3: Hybrid Images (30 points)**
    * Fourier transform.
    * Implement low and high pass filters and apply them to images.
    * Create a hybrid image using high-pass and low-pass fitlered images.
- Your job is to implement the sections marked with `TODO` to complete the tasks.
- Submission.
  - Please submit the notebook (ipynb and pdf) including the output of all cells.

2 Problem 1: Image Denoising

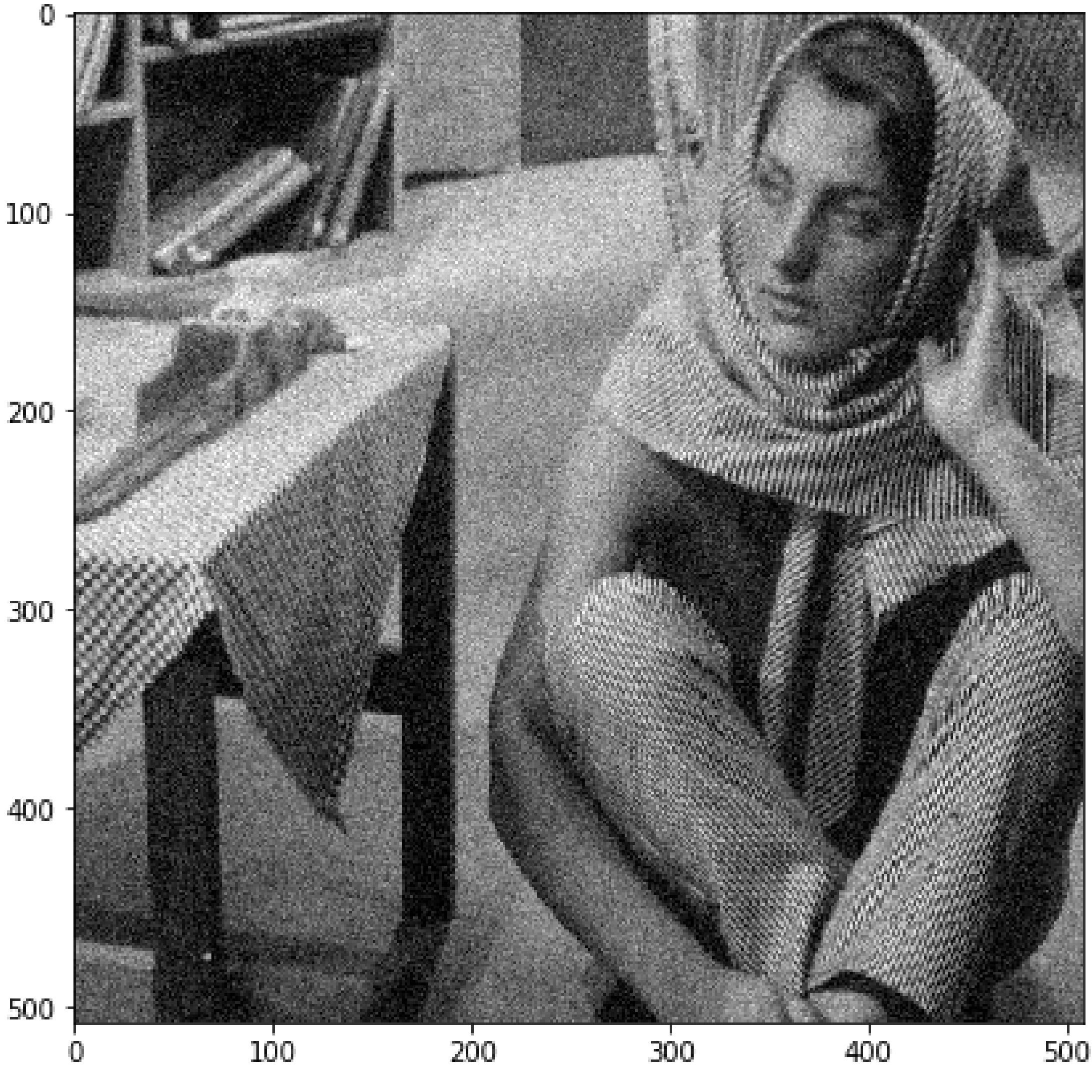
Taking pictures at night is challenging because there is less light that hits the film or camera sensor. To still capture an image in low light, we need to change our camera settings to capture more light. One way is to increase the exposure time, but if there is motion in the scene, this leads to blur. Another way is to use sensitive film that still responds to low intensity light. However, the trade-off is that this higher sensitivity increases the amount of noise captured, which often shows up as grain on photos. In this problem, your task is to clean up the noise with signal processing.

2.1 Visualizing the Grain

To start off, let's load up the image and visualize the image we want to denoise.

[1]:

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from IPython import display
from scipy.signal import convolve2d
from math import *
import time
%matplotlib inline

plt.rcParams['figure.figsize'] = [7, 7]

def load_image(filename):
    img = np.asarray(Image.open(filename))
    img = img.astype("float32") / 255.
    return img

def gray2rgb(image):
    return np.repeat(np.expand_dims(image, 2), 3, axis=2)

def show_image(img):
    if len(img.shape) == 2:
        img = gray2rgb(img)
    plt.imshow(img, interpolation='nearest')

# load the image
im = load_image('noisy_image.jpg')
im = im.mean(axis=2) # convert to grayscale
show_image(im)
```

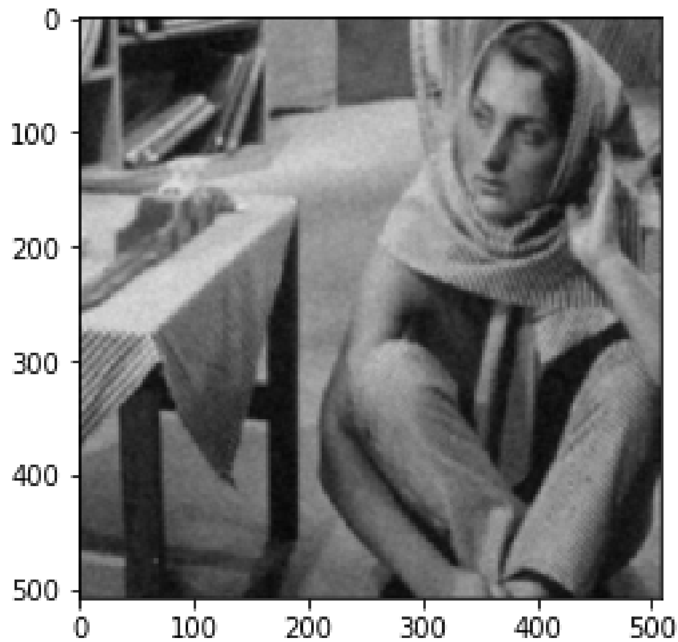


2.2 Mean Filter using "for" loop

Let's try to remove this grain with a mean filter. For every pixel in the image, we want to take an average (mean) of the neighboring pictures. Implement this operation using "for" loops and visualize the result:

[2]:
```
im_pad = np.pad(im, 5, mode='constant')  # pad the border of the original image
im_out = np.zeros_like(im)  # initialize the output image array
''' TODO: Implement a mean filter using "for" loop here (modify the im_out␣
↪matrix). '''
# assume the kernel size as 5*5
row_count, column_count = np.shape(im_pad)

im_temp = np.zeros_like(im_pad)
```

```
for i in range(2, row_count-2):
    for j in range(2, column_count-2):
        mean_kernel_res = im_pad[i-2:i+3,j-2:j+3]
        im_temp[i,j] = np.mean(mean_kernel_res)

im_out = im_temp[5:row_count-5,5:column_count-5]
show_image(im_out)
```



2.3 Implement the `convolve_image` function.

In practice, applying filters to images can be more efficient by using convolution, which is a function that takes as input the raw image and a filter matrix, and outputs the convolved image. Implement your `convolve_image` function below.

[3]:
```
# mode = same
def convolve_image(image, filter_matrix):
    ''' Convolve a 2D image using the filter matrix.
    Args:
        image: a 2D numpy array.
        filter_matrix: a 2D numpy array.
    Returns:
        the convolved image, which is a 2D numpy array same size as the input␣
 ↪image.

    TODO: Implement the convolve_image function here.
```

```
    '''
    #assume the size of width is odd
    width, height = np.shape(filter_matrix)

    padding_size = floor(width/2)
    im_pad = np.pad(im, padding_size, mode='constant')  # pad the border of the␣
↪original image
    row_count, column_count = np.shape(im_pad)
    im_out = np.zeros_like(im)  # initialize the output image array
    im_temp = np.zeros_like(im_pad)

    for i in range(padding_size, row_count-padding_size):
        for j in range(padding_size, row_count-padding_size):
            #crop the region applied to convolution with filter_matrix
            image_temp = im_pad[i-padding_size:
↪i-padding_size+width,j-padding_size:j-padding_size+height]
            # pratical calculation
            im_temp[i,j] = np.sum(image_temp * filter_matrix)

    im_out = im_temp[padding_size:row_count-padding_size,padding_size:
↪column_count-padding_size]
    return im_out
```

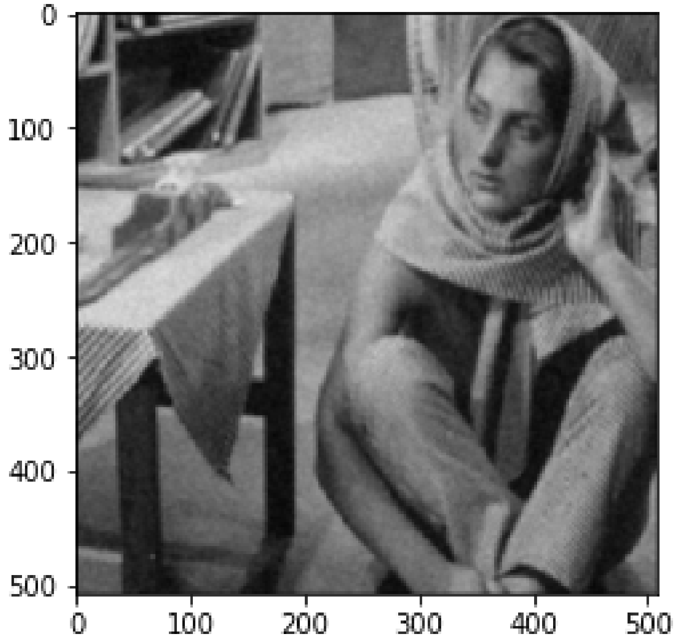
2.4 Mean Filter with Convolution

Implement this same operation with a convolution instead. Fill in the mean filter matrix here, and visualize the convolution result.

[4]:
```
mean_filt = None
''' TODO: Create a mean filter matrix here. '''
#assume the filter size: 5*5
mean_filt = np.ones((5,5)) / 25.0
```

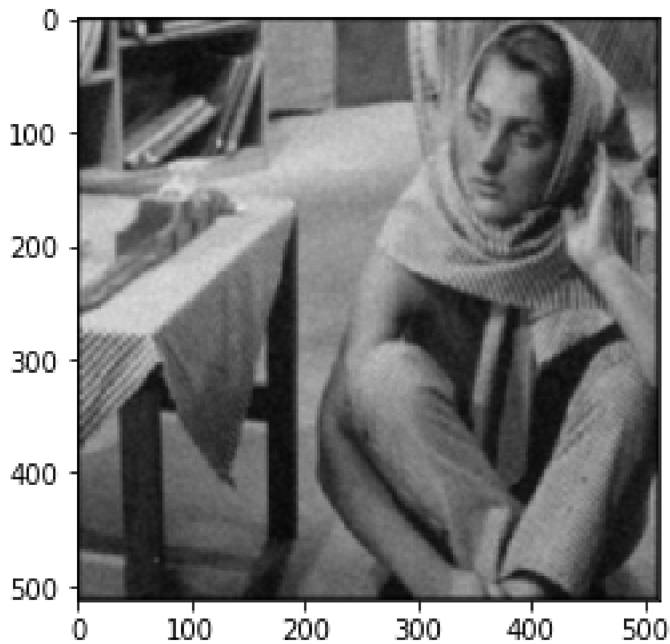
Apply mean filter convolution using your `convolve_image` function and the `mean_filt` matrix.

[5]:
```
show_image(convolve_image(im, mean_filt))
```



Compare your convolution result with the scipy.signal.convolve2d function (they should look the same).

[6]:
```
#mode = full
show_image(convolve2d(im, mean_filt))
```



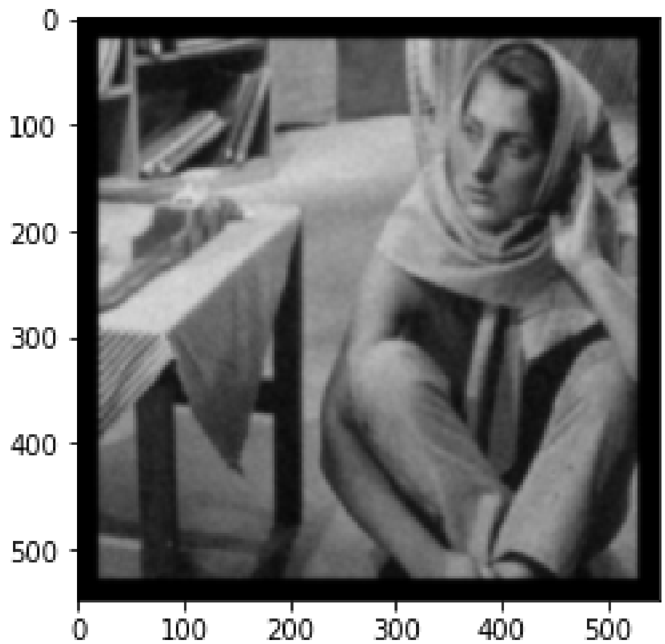
Note: In the sections below, we will use the `scipy.signal.convolve2d` function for grading. But fill free to test your `convolve_image` function on other filters as well.

2.5 Gaussian Filter

Instead of using a mean filter, let's use a Gaussian filter. Create a 2D Gaussian filter, and plot the result of the convolution.

Hint: You can first construct a one dimensional Gaussian, then use it to create a 2D dimensional Gaussian.

[7]:
```
def gaussian_filter(sigma, k=20):
    '''
    Args:
        sigma: the standard deviation of Gaussian kernel.
        k: controls size of the filter matrix.
    Returns:
        a 2D Gaussian filter matrix of the size (2k+1, 2k+1).

    TODO: Implement a Gaussian filter here.
    '''
    width = 2*k + 1
    height = 2*k + 1
    filter_gaussian = np.zeros((width, height))
    c = 1.0/(2 * pi * sigma * sigma )
    u, v = k, k
    for i in range(width):
        for j in range(height):
            exp_temp = -((i-u)*(i-u) + (j-v)*(j-v))/(2*sigma*sigma)
            filter_gaussian[i,j] = c * pow(e, exp_temp)
    return filter_gaussian
# mode = full
show_image(convolve2d(im, gaussian_filter(2)))
```

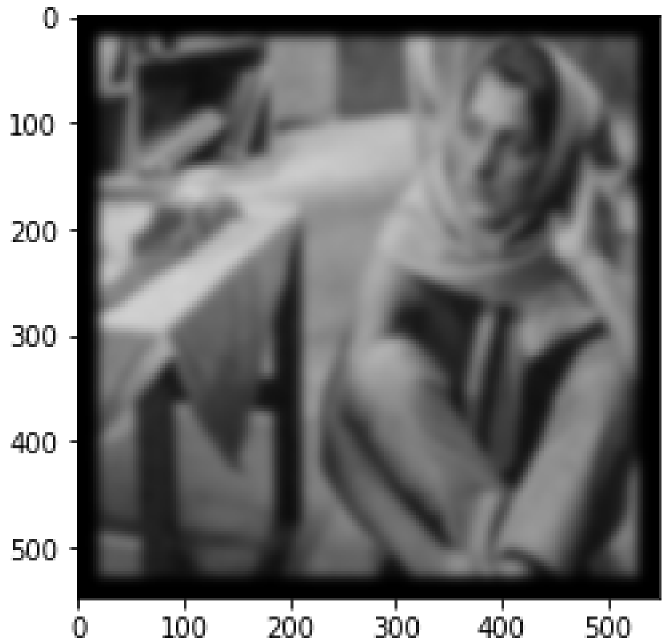


[8]:
```
# 3 sigma rule
#kernel.size >  (6*sigma)*(6*sigma)
g = gaussian_filter(2)
np.sum(g)
```

[8]: 1.0000000000000002

The amount the image is blurred changes depending on the sigma parameter. Change the sigma parameter to see what happens. Try a few different values.

[9]:
```
show_image(convolve2d(im, gaussian_filter(5)))
```



[10]:
```
np.sum(gaussian_filter(5))
```

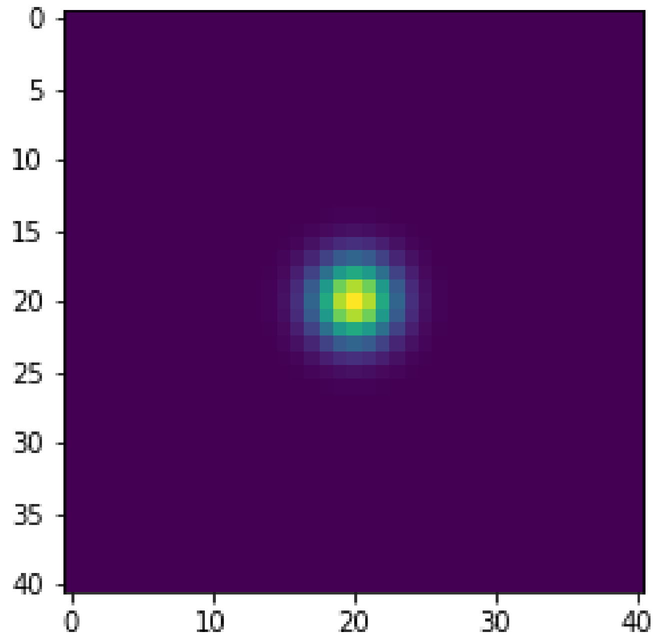
[10]: 0.9999197724777404

2.6 Visualizing Gaussian Filter

Try changing the sigma parameter below to visualize the Gaussian filter directly. This gives you an idea of how different sigma values create different convolved images.

[11]:
```
plt.imshow(gaussian_filter(sigma=2))
```

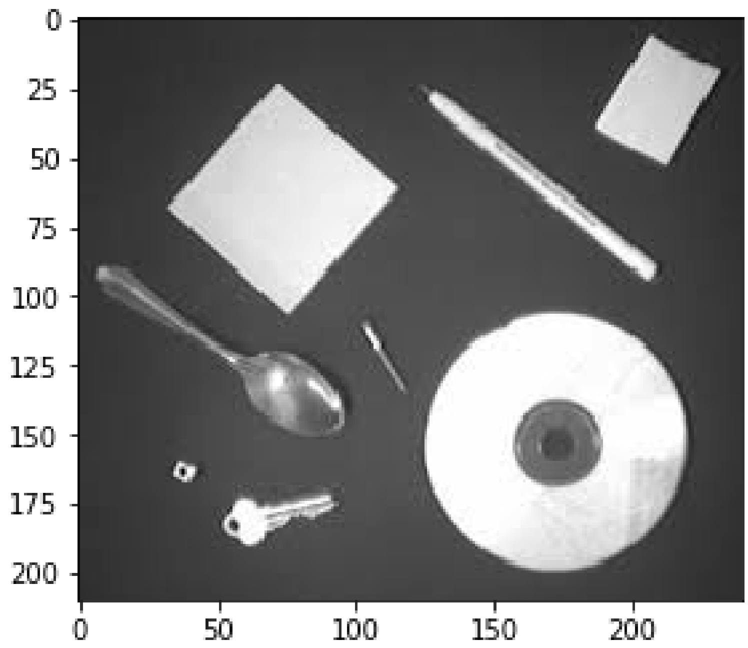
[11]: <matplotlib.image.AxesImage at 0x1c14dba4a8>



3 Problem 2: Edge Detection

There are a variety of filters that we can use for different tasks. One such task is edge detection, which is useful for finding the boundaries regions in an image. In this part, your task is to use convolutions to find edges in images. Let's first load up an edgy image.

[12]:
```
im = load_image('edge_detection_image.jpg')
im = im.mean(axis=2) # convert to grayscale
show_image(im)
```

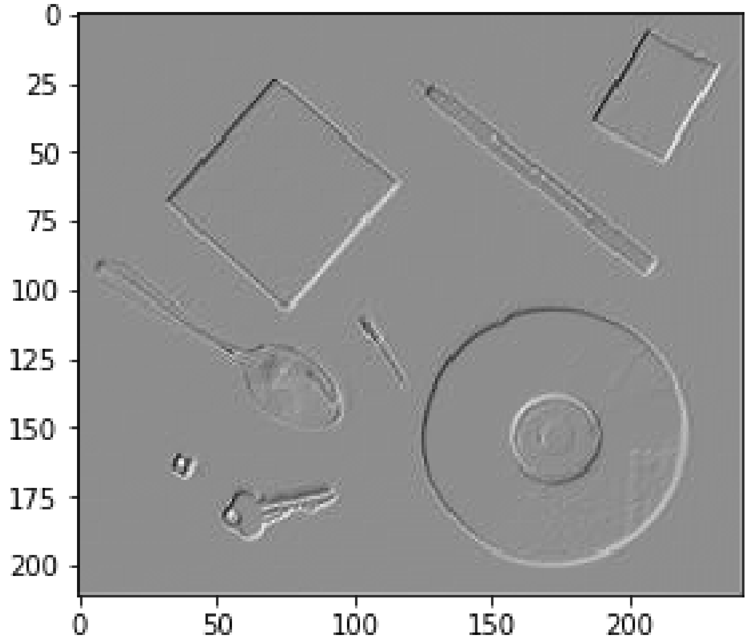


3.1 Delta Filters

The simplest edge detector is a delta filter. Implement a delta filter below, and convolve it with the image. Show the result.

[13]:
```
delta_filt = None

''' TODO: Implement a delta filter here. '''

delta_filt1 = np.array([[-1,1],[0,0]])
delta_filt2 = np.array([[-1,0],[1,0]])
# delta x + delta y
delta_filt = delta_filt1 + delta_filt2

#or linear propertity of convolution: same result
'''
f1 = convolve2d(im, delta_filt1)
f2 = convolve2d(im, delta_filt2)
f = f1 + f2
plt.imshow(f, cmap='gray')
'''
plt.imshow(convolve2d(im, delta_filt), cmap='gray')
```

[13]: `<matplotlib.image.AxesImage at 0x1c1596c898>`

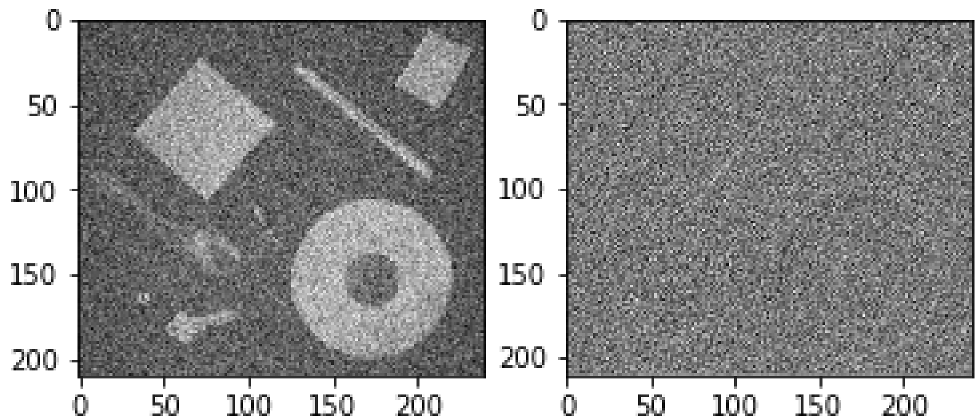


3.2 Noise

The issue with the delta filter is that it is sensitive to noise in the image. Let's add some Gaussian noise to the image below, and visualize what happens. The edges should be hard to see.

[14]:
```
im = load_image('edge_detection_image.jpg')
im = im.mean(axis=2)
im = im + 0.2*np.random.randn(*im.shape)

f, axarr = plt.subplots(1,2)
axarr[0].imshow(im, cmap='gray')
axarr[1].imshow(convolve2d(im, delta_filt), cmap='gray')
```

[14]:
```
<matplotlib.image.AxesImage at 0x1c14736748>
```

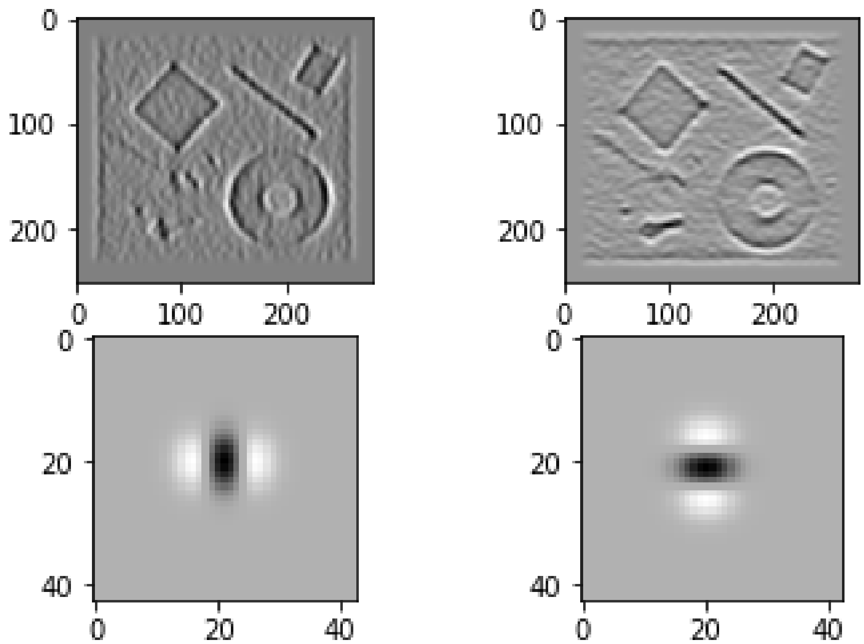


3.3 Laplacian Filters

Laplacian filters are edge detectors that are robust to noise (Why is this? Think about how the filter is constructed.). Implement a Laplacian filter below for both horizontal and vertical edges.

[15]:
```
lap_x_filt = None
''' TODO: Implement a Laplacian filter for horizontal edges. '''
temp = np.array([[-1,0],[1,0]])
temp_x_g1 = convolve2d(gaussian_filter(sigma=3), temp)
lap_x_filt = convolve2d(temp_x_g1,temp)

''' TODO: Implement a Laplacian filter for vertical edges. '''

temp = np.array([[-1,1],[0,0]])
temp_y_g1 = convolve2d(gaussian_filter(sigma=3), temp)
lap_y_filt = convolve2d(temp_y_g1,temp)



f, axarr = plt.subplots(2,2)
axarr[0,0].imshow(convolve2d(im, lap_y_filt), cmap='gray')
axarr[0,1].imshow(convolve2d(im, lap_x_filt), cmap='gray')
axarr[1,0].imshow(lap_y_filt, cmap='gray')
axarr[1,1].imshow(lap_x_filt, cmap='gray')
```

[15]:
```
<matplotlib.image.AxesImage at 0x1c1458ccf8>
```

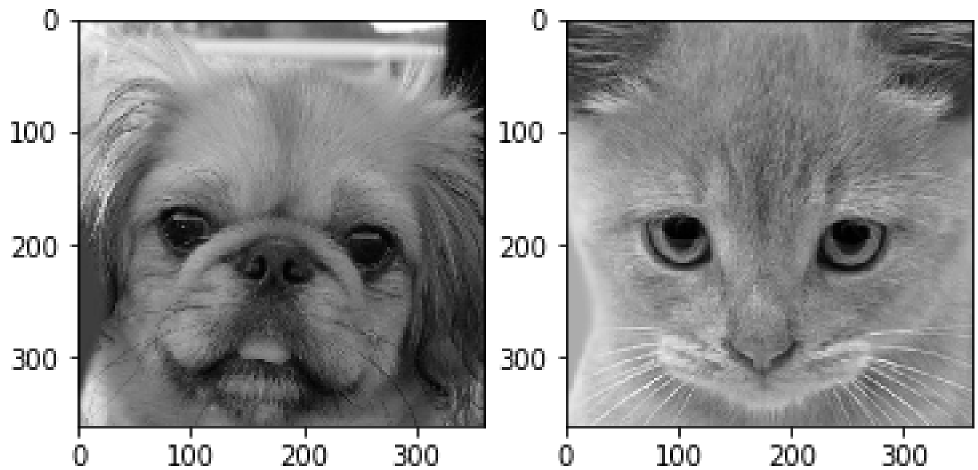


4 Problem 3: Hybrid Images

Hybrid images is a technique to combine two images in one. Depending on the distance you view the image, you will see a different image. This is done by merging the high-frequency components of one image with the low-frequency components of a second image. In this problem, you are going to use the Fourier transform to make these images. But first, let's visualize the two images we will merge together.

[16]:
```
from numpy.fft import fft2, fftshift, ifftshift, ifft2

dog = load_image('dog.jpg').mean(axis=-1)[:, 25:-24]
cat = load_image('cat.jpg').mean(axis=-1)[:, 25:-24]

f, axarr = plt.subplots(1,2)
axarr[0].imshow(dog, cmap='gray')
axarr[1].imshow(cat, cmap='gray')
```

[16]: `<matplotlib.image.AxesImage at 0x1c144f3668>`

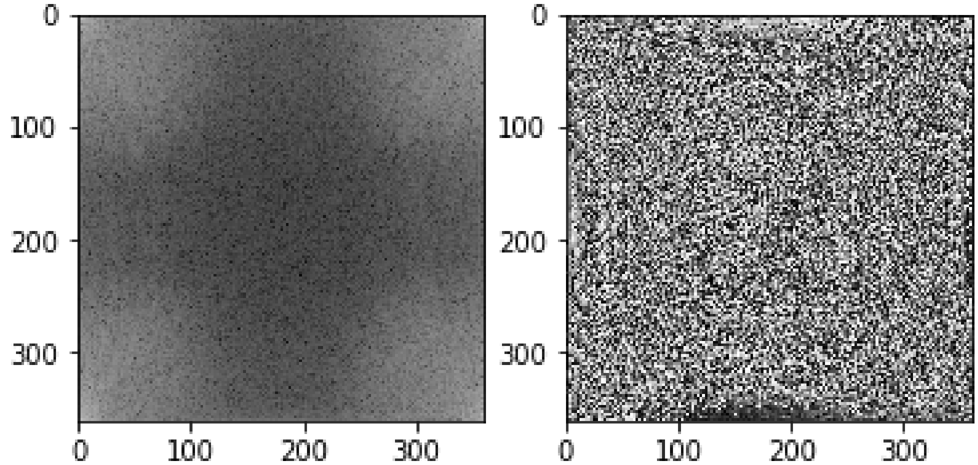


4.1 Fourier Transform

In the code box below, compute the Fourier transform of the two images. You can use the fft2 function. You can also use the fftshift function, which may help in the next section.

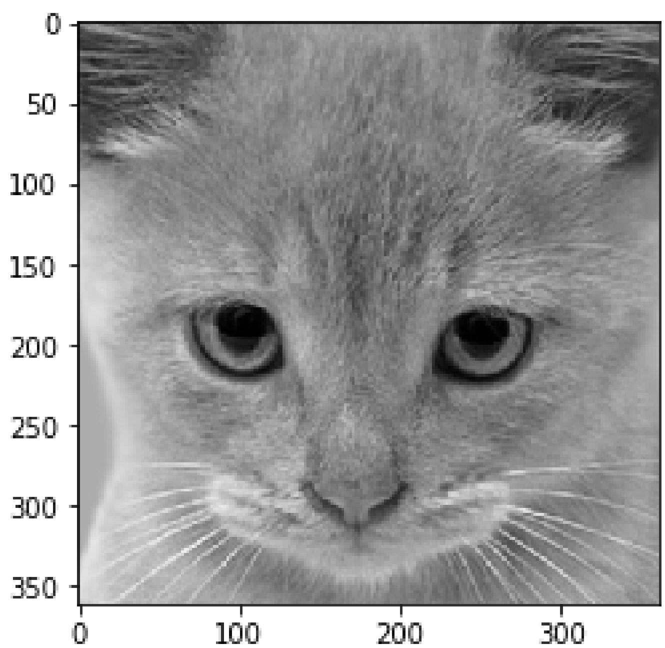
[17]:
```
cat_fft = None
''' TODO: compute the Fourier transform of the cat. '''
cat_fft =  np.fft.fft2(cat)

dog_fft = None
''' TODO: compute the Fourier transform of the dog. '''
dog_fft = np.fft.fft2(dog)

# Visualize the magnitude and phase of cat_fft. This is a complex number, so we␣
 ↪visualize
# the magnitude and angle of the complex number.
# Curious fact: most of the information for natural images is stored in the␣
 ↪phase (angle).
f, axarr = plt.subplots(1,2)
axarr[0].imshow(np.log(np.abs(cat_fft)), cmap='gray')
axarr[1].imshow(np.angle(cat_fft), cmap='gray')
```

[17]:
```
<matplotlib.image.AxesImage at 0x1c15624ac8>
```



[18]:
```
#reconstruction test
cat_ifft = np.fft.ifft2(cat_fft)
cat_ifft_image = np.abs(cat_ifft)
plt.imshow(cat_ifft_image, cmap='gray')
```

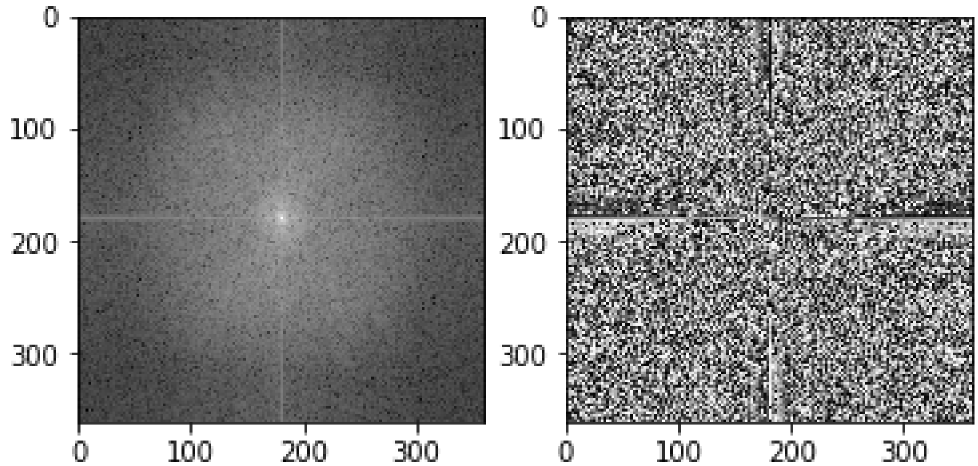
[18]: <matplotlib.image.AxesImage at 0x1c15d25b70>



[19]:
```
# shift the zero-frequency to the center
# shift the low frequency component to the center
```

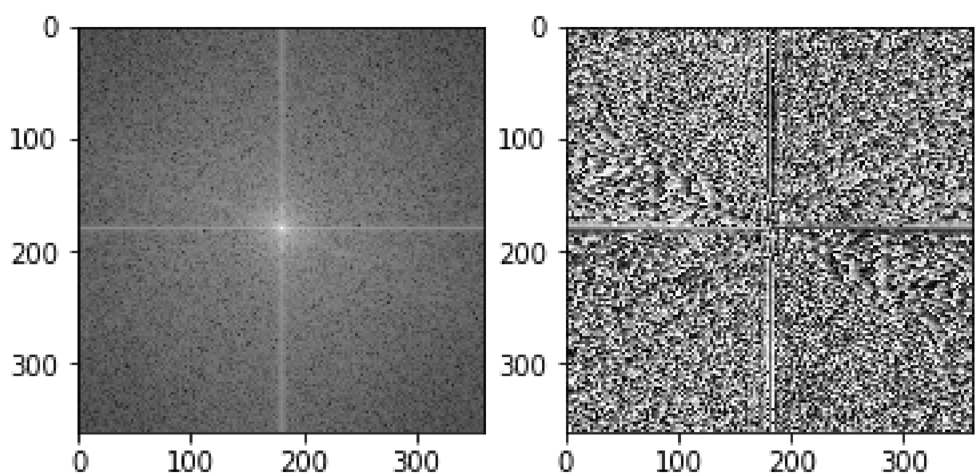
```python
shift_cat = np.fft.fftshift(cat_fft)
#shift_cat
f, axarr = plt.subplots(1,2)
axarr[0].imshow(np.log(np.abs(shift_cat)), cmap='gray')
axarr[1].imshow(np.angle(shift_cat), cmap='gray')
```

[19]: <matplotlib.image.AxesImage at 0x1c15d7ce10>



[20]:
```python
#shift_dog
# shift the zero-frequency to the center
# shift the low frequency component to the center
shift_dog = np.fft.fftshift(dog_fft)
f, axarr = plt.subplots(1,2)
axarr[0].imshow(np.log(np.abs(shift_dog)), cmap='gray')
axarr[1].imshow(np.angle(shift_dog), cmap='gray')
```

[20]: <matplotlib.image.AxesImage at 0x1c163e1e10>



4.2 Low and High Pass Filters

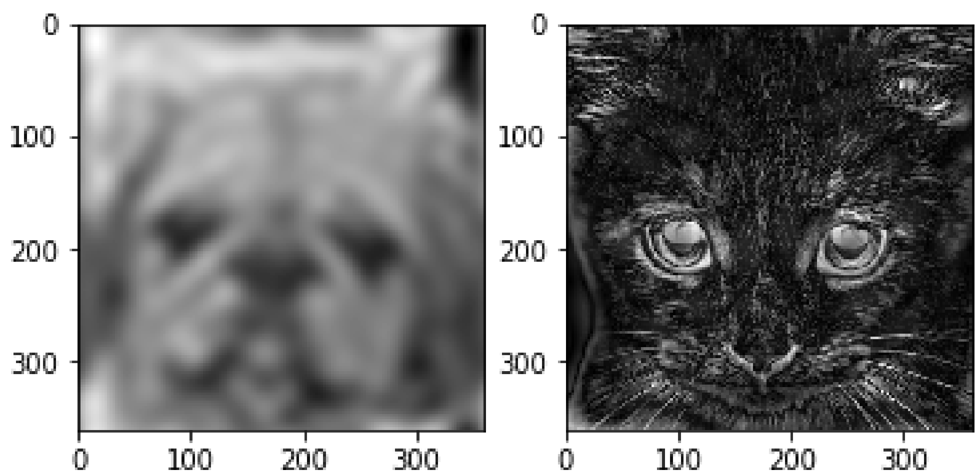
By masking the Fourier transform, you can compute both low and high pass of the images. In Fourier space, write code below to create the mask for a high pass filter of the cat, and the mask for a low pass filter of the dog. Then, convert back to image space and visualize these images.

You may need to use the functions ifft2 and ifftshift.

[21]:
```
#notes:
#after np.fft.fftshift: make low frequency component close to the center
# high frequency component far away from the center

high_mask = None
# TODO: Create the mask for a high pass filter of the cat.
# remove low frequency

threshold_1 = 5
row_count, column_count =  np.shape(shift_cat)
high_mask = np.ones((row_count, column_count))
#the coordinators of the center
x = int(row_count/2)
y = int(column_count/2)
high_mask[x-threshold_1:x+threshold_1,y-threshold_1:y+threshold_1] = 0



low_mask = None
# TODO: Create the mask for a low pass filter of the dog.
#remove high frequency
threshold_2 = 10
row_count, column_count =  np.shape(shift_cat)
low_mask = np.zeros((row_count, column_count))
# the coordinate of center
x = int(row_count/2)
y = int(column_count/2)
low_mask[x-threshold_2:y+threshold_2,y-threshold_2:y+threshold_2] = 1



cat_filtered = None
#TODO: Apply the high pass filter on the cat and convert back to image space.

shift_cat_hm = high_mask * shift_cat
cat_ifft2_hm = np.fft.ifft2(shift_cat_hm)
cat_filtered = np.abs(cat_ifft2_hm)
```

```
dog_filtered = None
# TODO: Apply the low pass filter on the dog and convert back to image space.
shift_dog_lm = low_mask * shift_dog
dog_ifft2_lm = np.fft.ifft2(shift_dog_lm)
dog_filtered = np.abs(dog_ifft2_lm)


f, axarr = plt.subplots(1,2)
axarr[0].imshow(dog_filtered, cmap='gray')
axarr[1].imshow(cat_filtered, cmap='gray')
```

[21]: <matplotlib.image.AxesImage at 0x1c16538588>



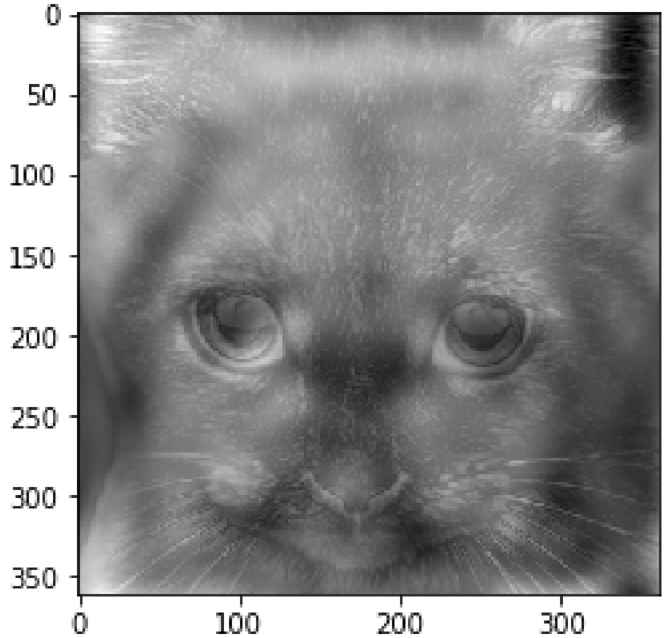
4.3 Hybrid Image Results

Now that we have the high pass and low pass fitlered images, we can create a hybrid image by adding them. Write the code to combine the images below, and visualize the hybrd image.

Depending on whether you are close or far away from your monitor, you should see either a cat or a dog. Try creating a few different hybrid images from your own photos or photos you found. Submit them, and we will show the coolest ones in class.

[22]:
```
hybrid = None
''' TODO: Compute the hybrid image here. '''

hybrid = dog_filtered + cat_filtered
# hybrid  = dog_filtered + cat_filtered * 2
# multiply high frequency component image cat_filtered with a coefficient may␣
 ↪get a better display effects
plt.imshow(hybrid, cmap='gray')
```

[22]: <matplotlib.image.AxesImage at 0x1c14627c50>



4.4 Acknowledgements

This homework is based on assignments from Aude Oliva at MIT, and James Hays at Georgia Tech.

[ ]: