jz2977_hw2

September 24, 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.

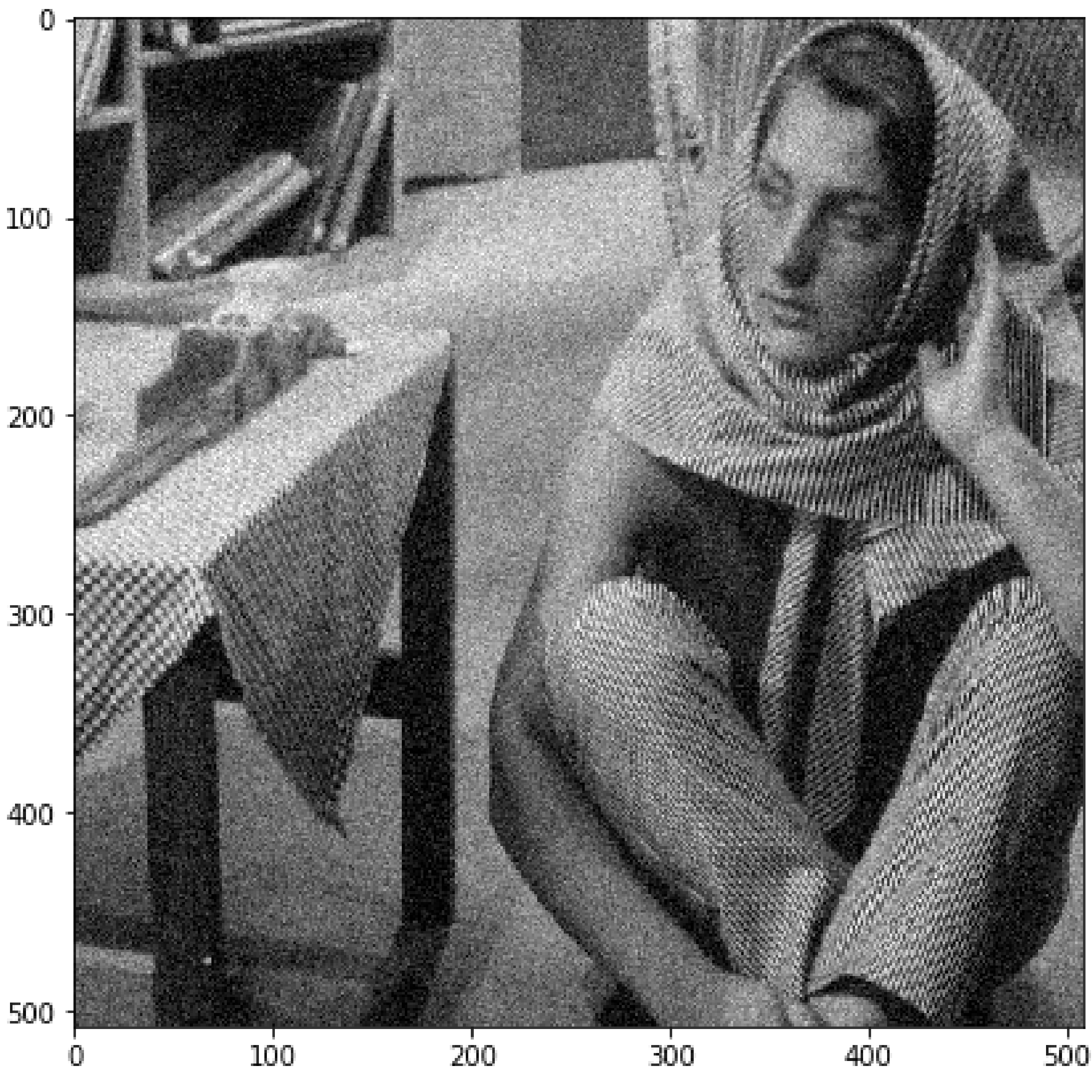
2 Problem 1: Image Denoising

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.

In [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)
```

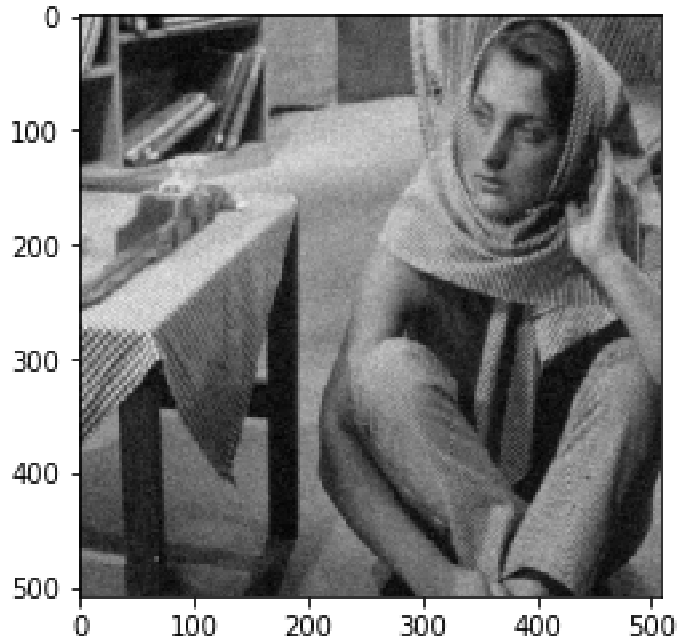


2.2 Mean Filter using "for" loop

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:

```
In [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). ''
        for i in range(im_out.shape[0]):
            for j in range(im_out.shape[1]):
                im_out[i,j] = np.sum(im_pad[i+4:i+7,j+4:j+7]) / 9
```

```
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.

```
In [3]: 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.
            '''

            kernel_h, kernel_w = filter_matrix.shape[0], filter_matrix.shape[1]

            if kernel_h != kernel_w:
                raise ValueError('Inconsistant kernel size!')

            if not (kernel_h & 1):
```

```
            raise ValueError('kernel size should be odd numbers!')

        pad_size = kernel_h // 2
        im_pad = np.pad(image, pad_size, mode='constant')
        im_out = np.zeros_like(image)
        for i in range(im_out.shape[0]):
            for j in range(im_out.shape[1]):
                im_out[i,j] = np.sum(filter_matrix * im_pad[1+i-pad_size:2+i+pad_size,1+j-p

        return im_out
```

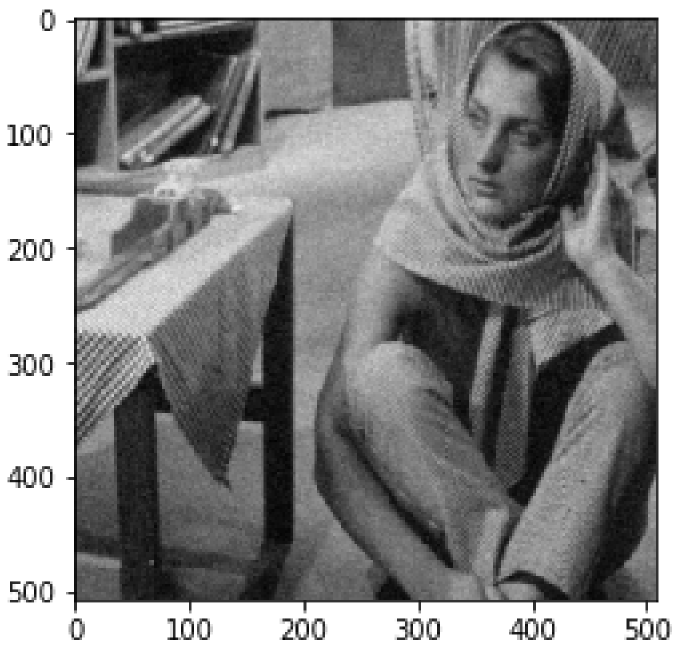
2.4 Mean Filter with Convolution

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

```
In [4]: ''' TODO: Create a mean filter matrix here. '''
        mean_filt = 1/9 * np.ones((3,3))
```

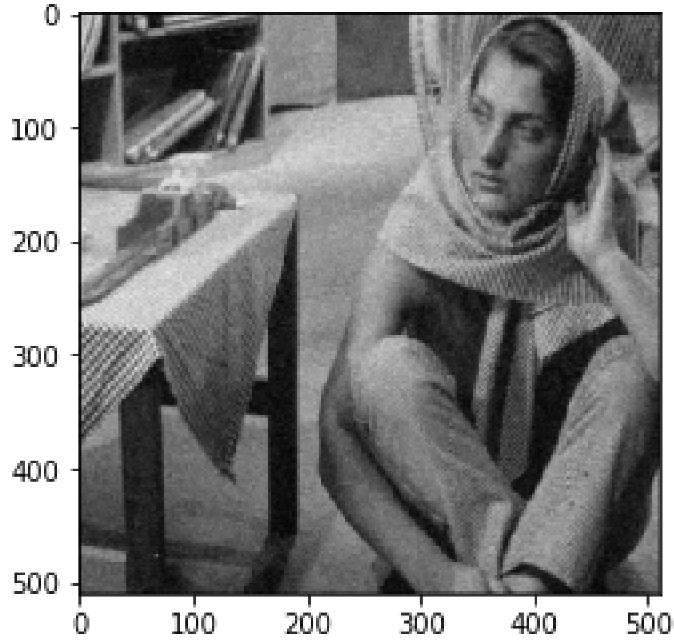
Apply mean filter convolution using your `convolve_image` function and the `mean_filt` matrix.

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



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

```
In [6]: 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.

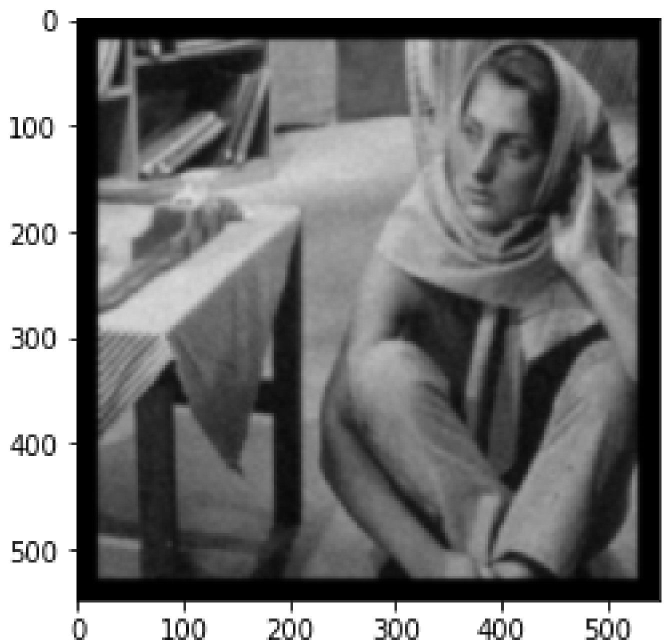
2.5 Gaussian Filter

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.

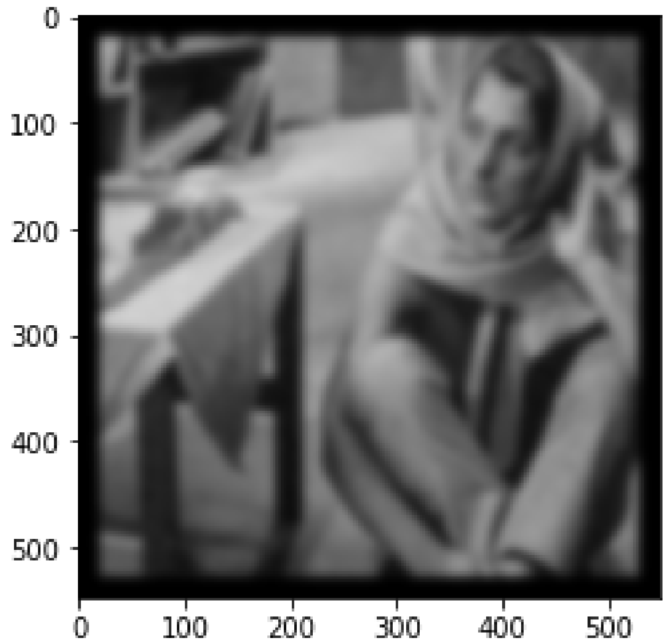
```
In [7]: def gaussian_filter(sigma, k=20):
            '''
            Args:
                sigma: the standard deviation of Gaussian kernel.
                k: controls size of the filter matrix.
            Returns:
                a 2D Gaussian filter matrix of the size (2k+1, 2k+1).

            TODO: Implement a Gaussian filter here.
            '''
            g_filter = np.zeros((2*k+1, 2*k+1))

            for i in range(k+1):
                for j in range(i,k+1):
                    g_filter[k+i, k+j] = 1 / (2*np.pi*sigma*sigma) * np.exp(-(i*i + j*j) / (2*
                    g_filter[k-i, k+j] = g_filter[k+i, k+j]
                    g_filter[k-i, k-j] = g_filter[k+i, k+j]
```

```
                g_filter[k+i, k-j] = g_filter[k+i, k+j]

                g_filter[k+j, k+i] = g_filter[k+i, k+j]
                g_filter[k-j, k+i] = g_filter[k+i, k+j]
                g_filter[k-j, k-i] = g_filter[k+i, k+j]
                g_filter[k+j, k-i] = g_filter[k+i, k+j]



    g_filter = g_filter / np.sum(g_filter)

    return g_filter

show_image(convolve2d(im, gaussian_filter(2)))
```



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.

```
In [8]: show_image(convolve2d(im, gaussian_filter(5)))
```

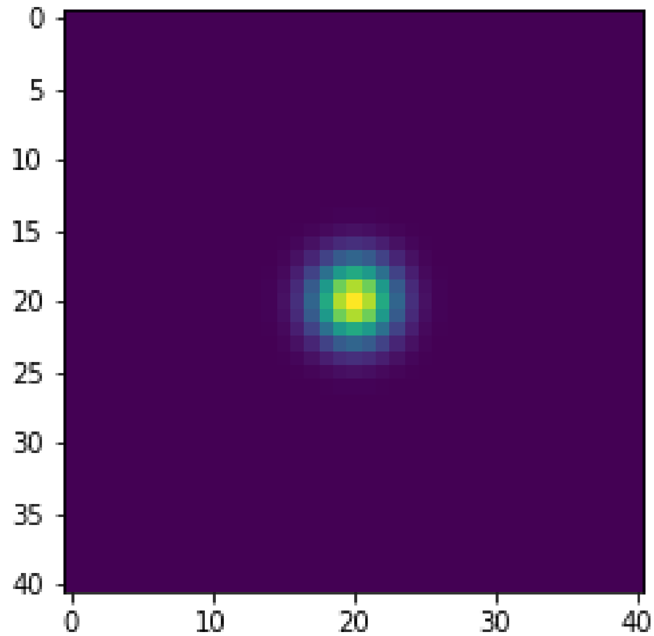


2.6 Visualizing Gaussian Filter

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.

```
In [9]: plt.imshow(gaussian_filter(sigma=2))
```

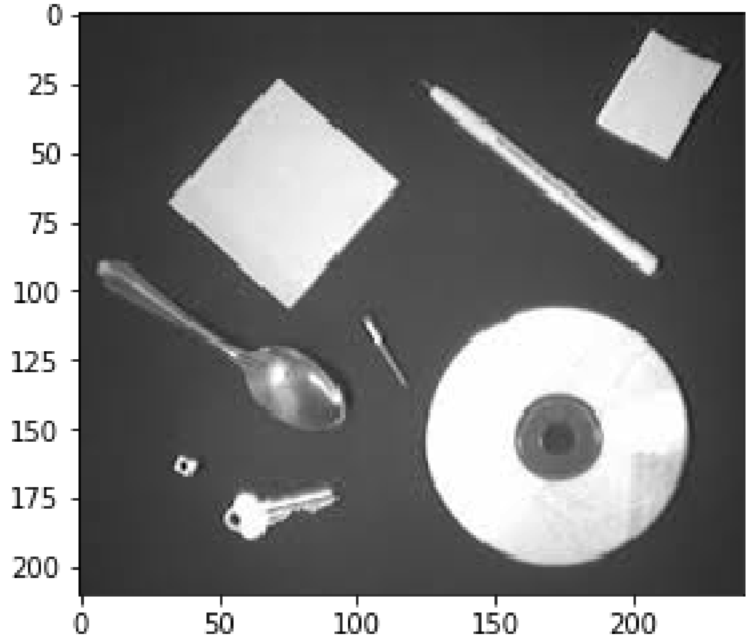
```
Out[9]: <matplotlib.image.AxesImage at 0x28c5d7b9978>
```



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.

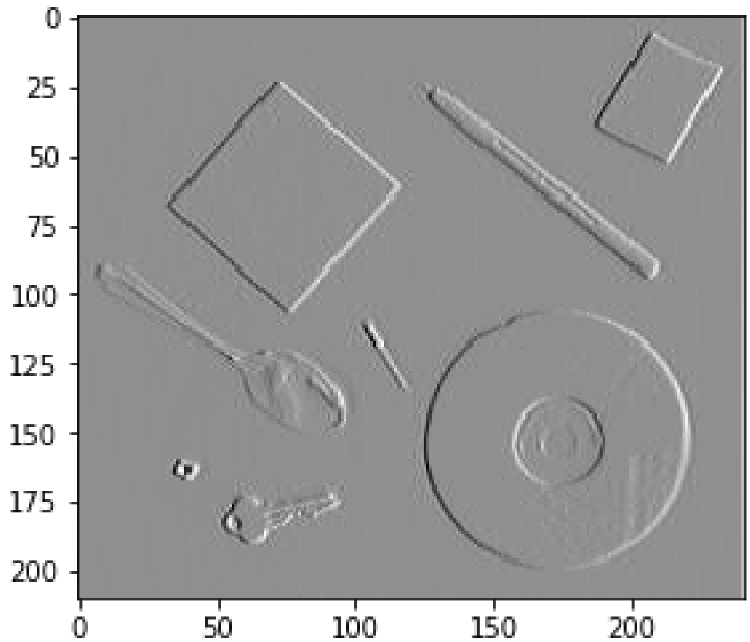
```
In [10]: im = load_image('edge_detection_image.jpg')
         im = im.mean(axis=2) # convert to grayscale
         show_image(im)
```



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.

```
In [11]: delta_filt = np.array([
             [-1,1]
         ])
         ''' TODO: Implement a delta filter here. '''

         plt.imshow(convolve2d(im, delta_filt), cmap='gray')

Out[11]: <matplotlib.image.AxesImage at 0x28c5a039c18>
```

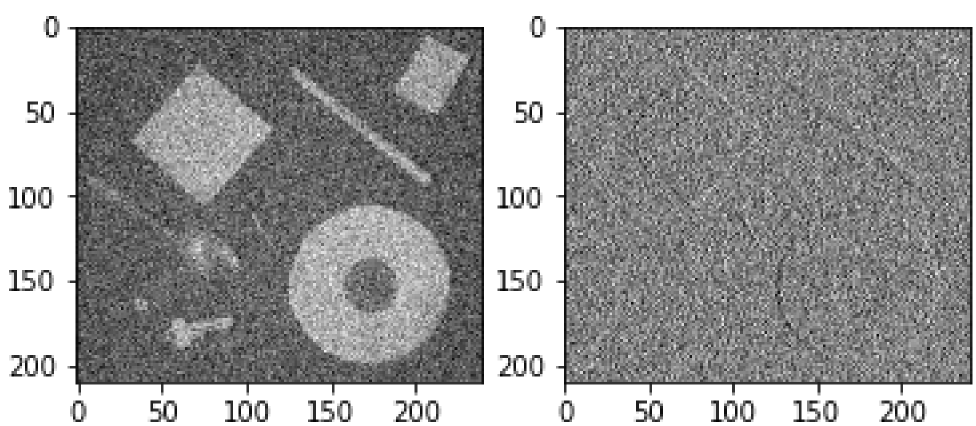


3.2 Noise

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.

In [12]:
```
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')
```

Out[12]: <matplotlib.image.AxesImage at 0x28c5d175438>

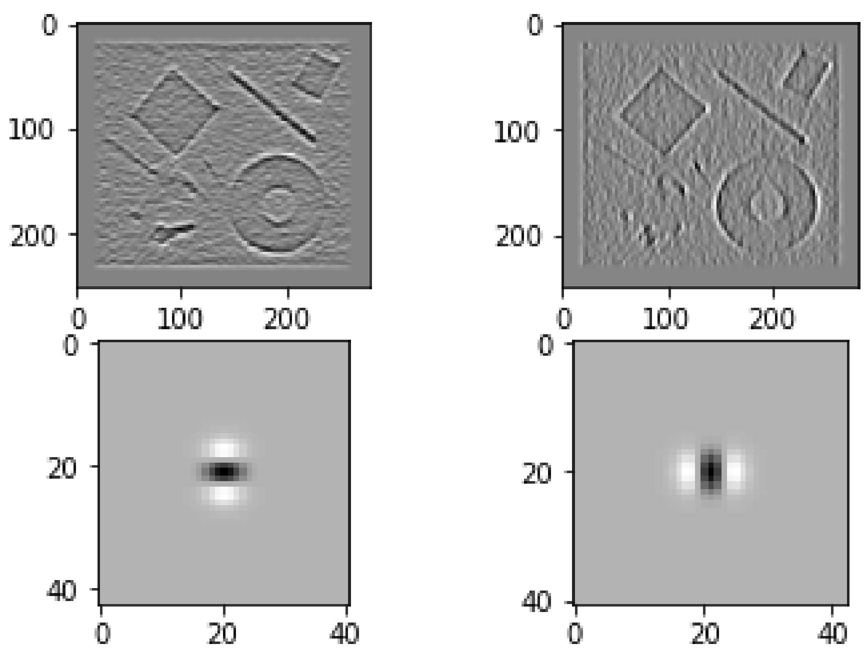


3.3 Laplacian Filters

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.

In [13]:
```
x_delta = np.array(
    [[-1,1]]
)

y_delta = np.array(
    [[-1],[1]]
)

lap_x_filt = convolve2d(convolve2d(gaussian_filter(sigma=2),x_delta),x_delta)
''' TODO: Implement a Laplacian filter for horizontal edges. '''
lap_y_filt = convolve2d(convolve2d(gaussian_filter(sigma=2),y_delta),y_delta)
''' TODO: Implement a Laplacian filter for vertical edges. '''

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')
```

Out[13]: <matplotlib.image.AxesImage at 0x28c5d4e88d0>

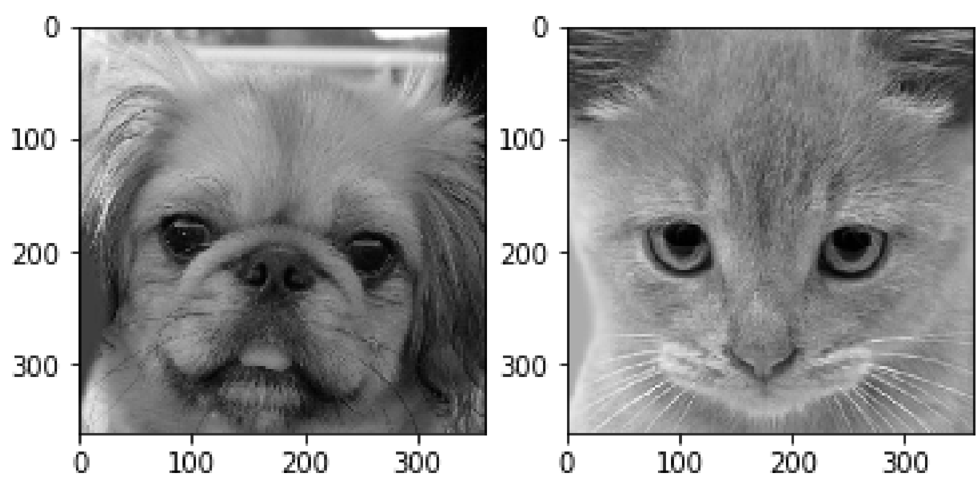


4 Problem 3: Hybrid Images

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.

```
In [14]: 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')
```

```
Out[14]: <matplotlib.image.AxesImage at 0x28c5d59ec88>
```



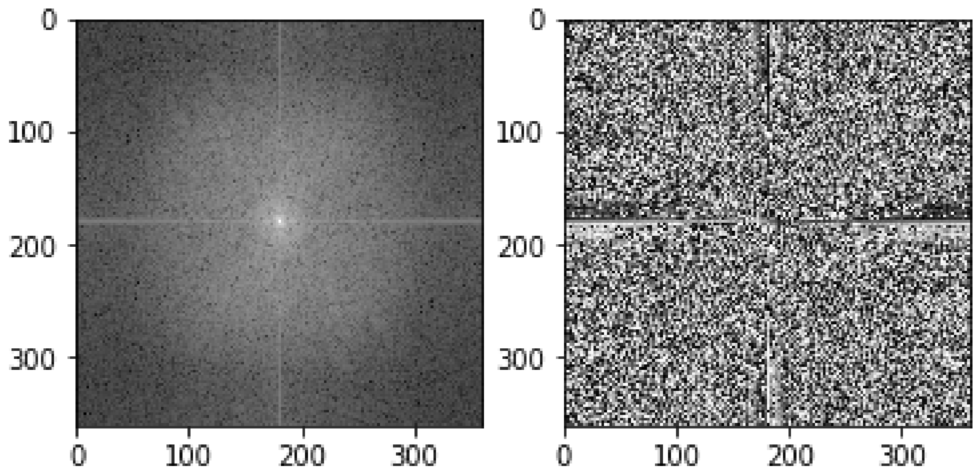
4.1 Fourier Transform

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.

```
In [15]: cat_fft = fft2(cat)
         ''' TODO: compute the Fourier transform of the cat. '''
         dog_fft = fft2(dog)
```

```
''' TODO: compute the Fourier transform of the dog. '''

cat_fft = fftshift(cat_fft)
dog_fft = fftshift(dog_fft)


# Visualize the magnitude and phase of cat_fft. This is a complex number, so we visua
# the magnitude and angle of the complex number.
# Curious fact: most of the information for natural images is stored in the phase (an
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')
```

Out[15]: <matplotlib.image.AxesImage at 0x28c5d634e80>



4.2 Low and High Pass Filters

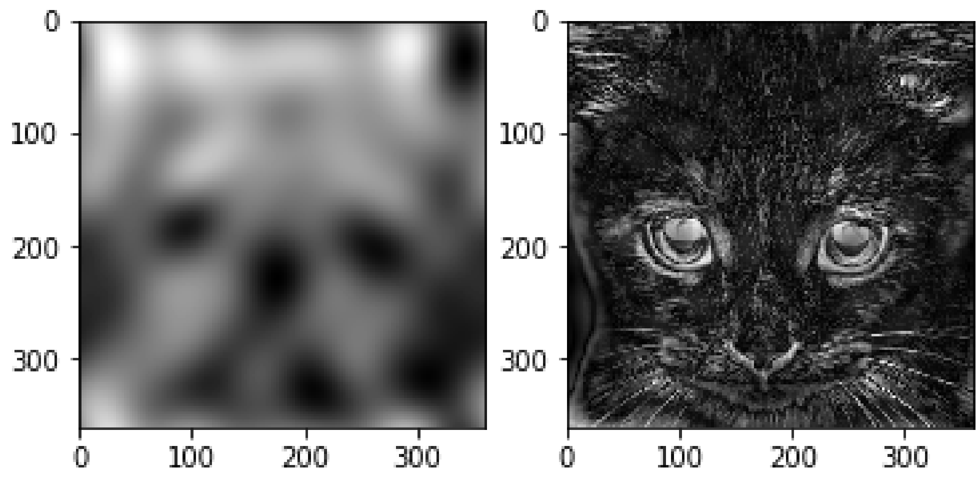
By masking the Fourier transform, you can compute both low and high pass of the images. In Fourier space, write code below to create the mask for a high pass filter of the cat, and the mask for a low pass filter of the dog. Then, convert back to image space and visualize these images.

You may need to use the functions ifft2 and ifftshift.

In [16]:
```
''' TODO: Create the mask for a high pass filter of the cat. '''
cat_center = (cat.shape[0] // 2 + 1, cat.shape[1] // 2 + 1)


threshold = 5
high_mask = np.ones_like(cat)
high_mask[cat_center[0]-threshold:cat_center[0]+threshold,cat_center[1]-threshold:cat_

''' TODO: Create the mask for a low pass filter of the dog. '''
```

```
dog_center = (dog.shape[0] // 2 + 1, dog.shape[1] // 2 + 1)


low_mask = np.zeros_like(dog)
low_mask[dog_center[0]-threshold:dog_center[0]+threshold,dog_center[1]-threshold:dog_c

''' TODO: Apply the high pass filter on the cat and convert back to image space. '''

cat_filtered = np.multiply(cat_fft,high_mask)
cat_filtered = ifftshift(cat_filtered)
cat_filtered = np.abs(ifft2(cat_filtered))
''' TODO: Apply the low pass filter on the dog and convert back to image space. '''

dog_filtered = np.multiply(dog_fft,low_mask)
dog_filtered = ifftshift(dog_filtered)
dog_filtered = np.abs(ifft2(dog_filtered))


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

```
Out[16]: <matplotlib.image.AxesImage at 0x28c5d6cf978>
```

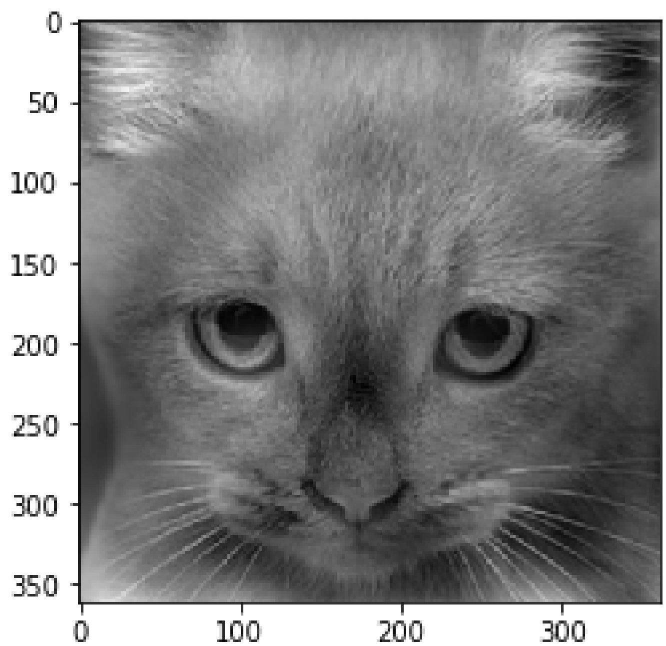


4.3 Hybrid Image Results

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.

```
In [17]: ''' TODO: Compute the hybrid image here. '''

         hybrid = cat_fft * high_mask + dog_fft * low_mask
         hybrid = np.abs(ifft2(ifftshift(hybrid)))

         plt.imshow(hybrid, cmap='gray')

Out[17]: <matplotlib.image.AxesImage at 0x28c5d739e10>
```



4.4 Acknowledgements

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