Project 1

Robert Relyea

Spring 2019

1 Introduction

Space domain image filtering is useful for augmenting image content as well as reducing noisiness in images. Operations in the frequency domain of images through the Fourier transform are also useful for performing image reconstruction and illustrating the importance of frequency content in the imaging space. In this exercise, several experiments were performed on images in both the space and frequency domains.

2 Filtering in the Space Domain

Space domain image operations are commonly performed using convolutions with kernels specific to the task. These kernels can sharpen and blur images as well as highlight edges within the image content. An example 5x5 high-pass kernel is shown in Figure 1.

-1/25	-1/25	-1/25	-1/25	-1/25
-1/25	-1/25	-1/25	-1/25	-1/25
-1/25	-1/25	24/25	-1/25	-1/25
-1/25	-1/25	-1/25	-1/25	-1/25
-1/25	-1/25	-1/25	-1/25	-1/25

Figure 1: Illustration of a 5x5 high-pass kernel with a scaling factor applied.

The first space domain operation performed was a high-pass filter with the kernel illustrated in Figure 1. The input image utilized for this operation is shown in Figure 2 and the resulting filtered image is shown in Figure 3. The filtered image contains only the high frequency content of the original image including the edges of the fur and the upper eyelids. The high-pass operation was completed in 0.2901 seconds using a convolution operation implemented with Python and NumPy. Edges of the image are padded by continuing the last edge value.

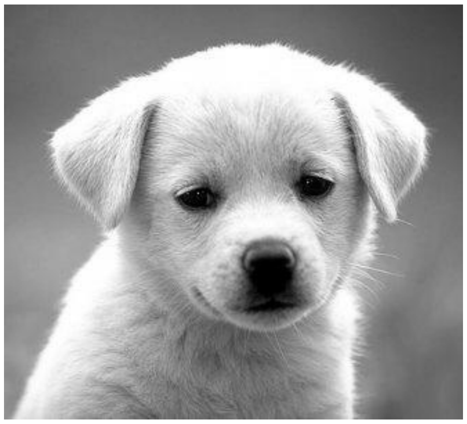

Figure 2: A grayscale image with rich frequency content.

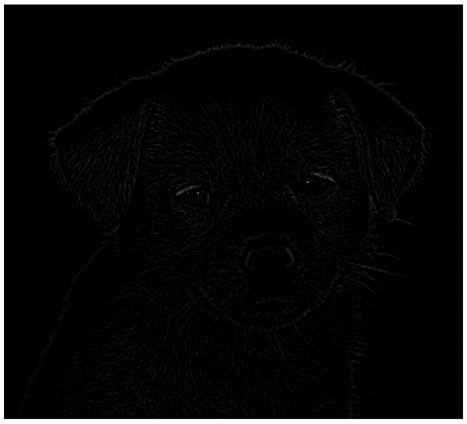

Figure 3: Image from Figure 2 after being passed through a high-pass filter.

A low-pass filter was also applied to the image in Figure 2. The resulting filtered image is shown in Figure 4. The resulting image lacks much of the high-frequency fur detail and the eye reflections are invisible. With subsequent passes through the low-pass filter, the image becomes blurrier. This is shown in Figure 5 where the original image was passed through a low-pass filter three times.

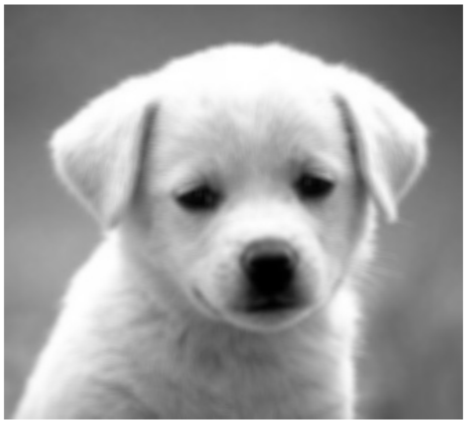

Figure 4: Image from Figure 2 after being passed through a low-pass filter.

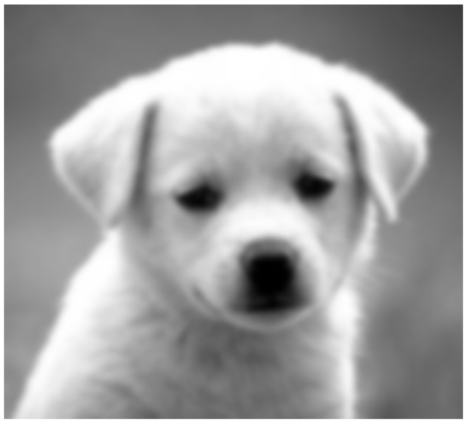

Figure 5: Image from Figure 2 after being passed through a low-pass filter three times.

A Sobel edge detection filter was also applied to the image in Figure 2. The resulting edges in the x and y directions after thresholding are shown in Figure 6, and the sum of x and y edges are shown in Figure 7.

3 Filtering a Noisy Image

Spatial domain operations on images can be used to reduce noise and improve the overall quality of the image. To illustrate this capability, salt and pepper and Gaussian noise were added to the image shown in Figure 2. The signal to

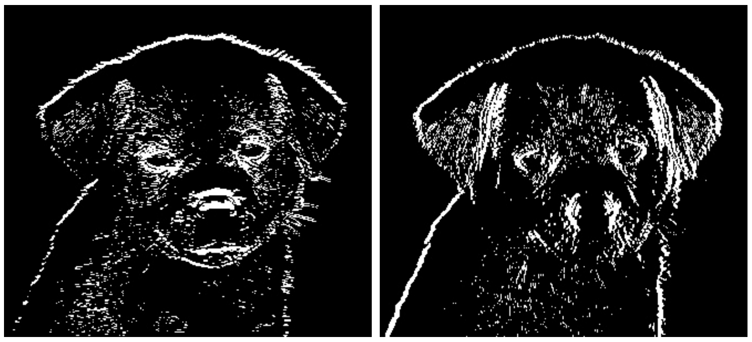

Figure 6: Image from Figure 2 after a Sobel x-axis (left) and y-axis (right) edge detection filter was applied.

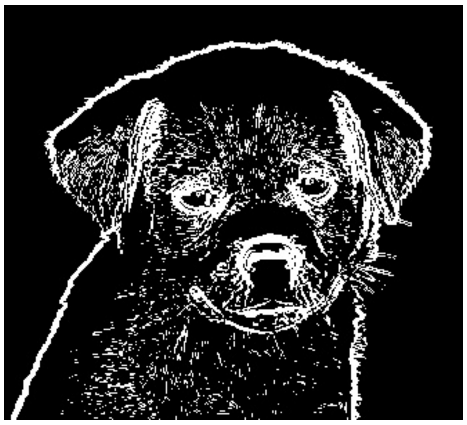

Figure 7: Sum of Sobel edges shown in Figure 6.

noise ratio for the image with added salt and pepper noise shown in Figure 8 was -0.8288 dB. The signal to noise ratio for the image with added Gaussian noise shown in Figure 8 was -0.4196 dB.

Two common approaches for reducing noise in images include low-pass and median filtering. Low-pass or average filtering is effective for uniformly distributed noise while median filtering is better suited for filtering out impulse noise. The results of low-pass and median filtering on the image with salt and pepper noise are shown in Figure 9. The results of low-pass and median filtering on the image with Gaussian noise are shown in Figure 10.

To highlight the effects of the added noise shown in Figure 8 Sobel edge detection was performed. The resulting images are shown in Figure 11. The salt and pepper noise causes a significant amount of new edges to be detected. The Gaussian noise dominates the output of the edge detector. To produce stronger edge detections in noisy images, one of the two explored filtering methods should

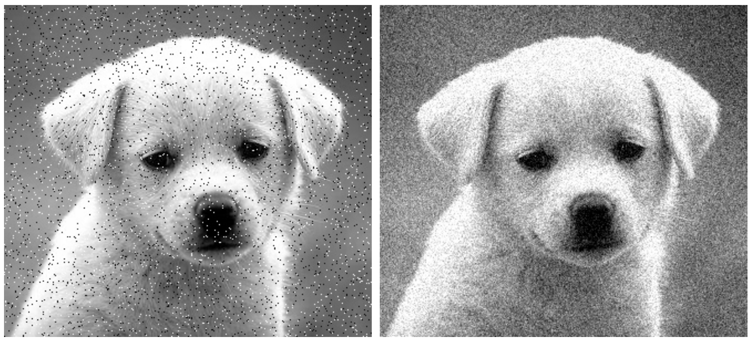

Figure 8: Image from Figure 2 after salt and pepper noise was added (left) and Gaussian noise was added (right).

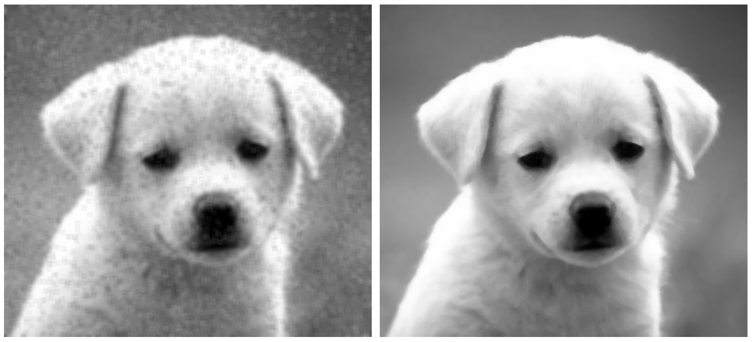

Figure 9: Image with salt and pepper noise from Figure 8 after being passed through a low-pass filter (left) and a median filter (right).

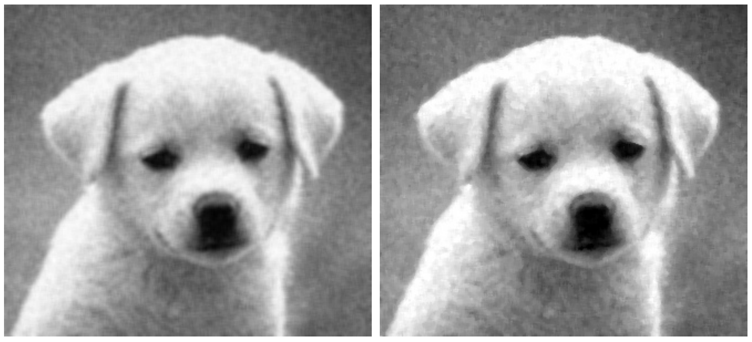

Figure 10: Image with Gaussian noise from Figure 8 after being passed through a low-pass filter (left) and a median filter (right).

be applied. For salt and pepper noise a median filter is ideal and a low-pass filter for Gaussian noise. The Sobel edge detection results from appropriate filtering for the noisy images are shown in Figure 12. The results show a drastic reduction in detected edges contributed by noise as well as a loss of edges compared to the original image shown in Figure 7.

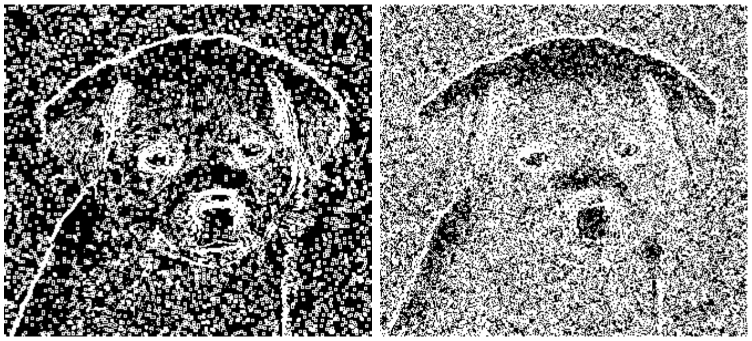

Figure 11: The image with salt and pepper noise (left) and Gaussian noise (right) from Figure 8 after being passed through a Sobel edge detector.

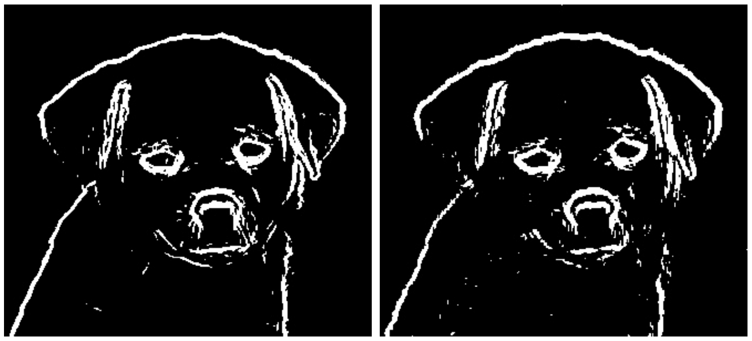

Figure 12: Sobel edge detection on pre-filtered noisy images. An image with salt and pepper noise reduced by median filtering and passed through a Sobel edge detector is shown on the left. An image with Gaussian noise reduced by low-pass filtering and passed through a Sobel edge detector is shown on the right.

4 The 2-D Fourier Transform

The frequency domain exposes additional features about the content of an image. The frequency spectrum of an image can be obtained by performing a Fourier transform. The fast Fourier transform (FFT) can be calculated through NumPy by utilizing the np.fft.fft2 command on an image as documented at [1]. The magnitude of the resulting spectral information of the image from Figure 2 is shown in Figure 13. The origin is located in the corners of the image after the np.fft.fftshift function call. The np.fft.fftshift command shifts the origin into the center of the frame and performing the log operation improves the visibility of the magnitudes across the spectrum. These adjusted magnitude visualizations are shown in Figure 14.

Figure 13: Visualization of the frequency magnitude of the image in Figure 2 after performing a FFT.

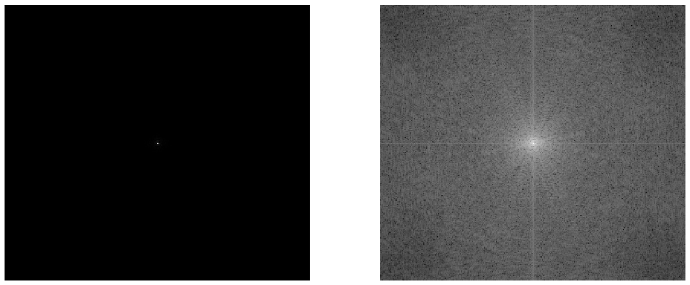

Figure 14: Visualization of the spectral magnitudes shown in Figure 13 after applying np.fft.fftshift (left) and the log operation (right).

Reconstruction of the original image after performing the FFT is possible through the inverse fast Fourier transform (IFFT). The NumPy np.fft.ifft2 command exposes this functionality. The resulting image after applying the IFFT to the previously calculated FFT is shown in Figure 15. Further reconstructions were performed after zeroing out frequencies outside a certain radius. Mean squared error results for all reconstructions are shown in Table 1. Reconstructed images after zeroing magnitudes outside a radius of N/3 and N/16 are shown in Figure 16. These images show a degradation in high frequency content compared to the original image in Figure 2.

Figure 15: Reconstructed image utilizing the IFFT on the FFT of the image shown in Figure 2.

Table 1: Mean squared error results for IFFT image reconstruction after altering the frequency magnitude of the image shown in Figure 2.

Zeroing Radius	**Mean Squared Error**
Unaltered	9.637e-30
N/3	4.092e-04
N/4	5.967e-04
N/8	1.040e-03
N/16	2.093e-03

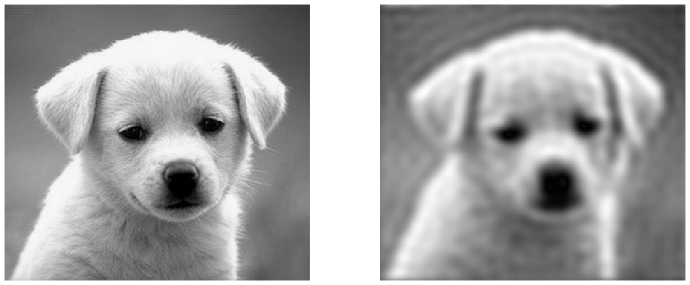

Figure 16: Reconstructed image utilizing the IFFT on the FFT of the image shown in Figure 2 after zeroing frequency magnitudes outside radii N/3 (left) and N/16 (right).

5 The Magnitude and Phase of the 2-D DFT

To illustrate the importance of both the real and imaginary components of the FFT when reconstructing an image, two separate images were reconstructed after swapping frequency magnitudes and keeping their phases. The two images used for this experiment are shown in Figure 17 which were obtained from the dataset provided by Khosla et al. [2]. The magnitude and phase of the first image can be seen in Figure 18 and the frequency content of the second image is shown in 19. The resulting reconstructed images are shown in Figure 20. The resulting mean squared reconstruction error of the first image was 0.15336 and the second image was 0.15225. Both reconstructed images contain components of the other image. A difference in overall intensity is the only distinguishable property between the reconstructed images.

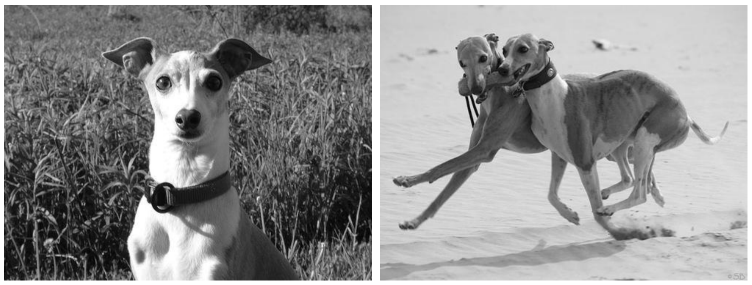

Figure 17: Grayscale images with rich frequency content.

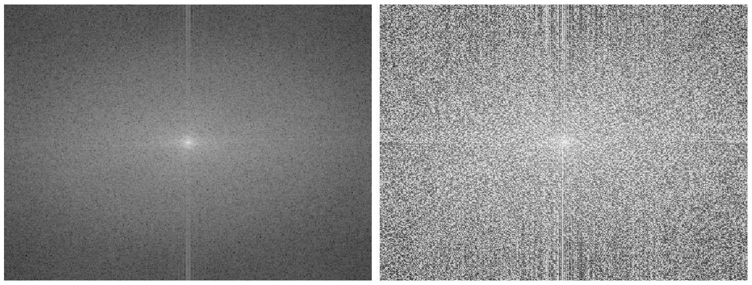

Figure 18: Magnitude (left) and phase (right) of the first image shown in Figure 17.

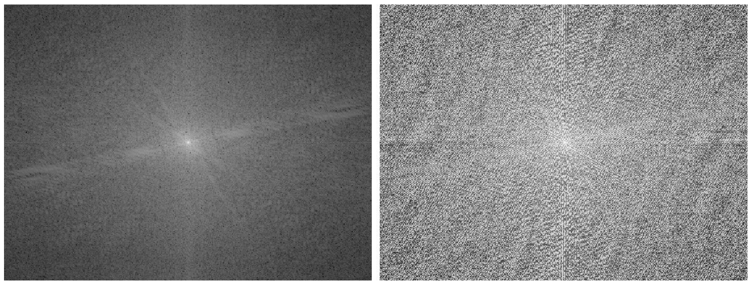

Figure 19: Magnitude (left) and phase (right) of the second image shown in Figure 17.

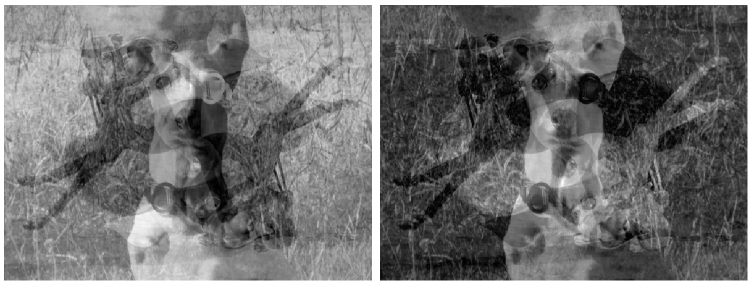

Figure 20: Reconstructions of the first (left) and second (right) images shown in Figure 17 after swapping frequency magnitudes obtained from the FFT.

6 Discussion

Several different concepts were explored in this exercise including operations in both the spatial and frequency domains of different images. High and low frequency content filtering were implemented and their effects after being applied to images were documented. Low-pass and median filtering were utilized to reduce noise in images where salt and pepper and Gaussian noise were artificially introduced. Median filtering proved to be effective for reducing salt and pepper noise as illustrated by the resulting Sobel edge detection output. This was also demonstrated for low-pass filtering on Gaussian noise. The frequency content of images was extracted successfully utilizing FFT operations available in the NumPy fft library. The real and imaginary components were visualized after performing a np.fft.fftshift operation and applying the log operation to the resulting magnitude and phase components. The effects of zeroing out high-frequency magnitudes and performing image reconstruction through the IFFT was observed. Mean squared error results of the varied reconstructions showed an increase in reconstruction error as more high frequency content was removed. Further reconstruction errors were observed when swapping the frequency magnitudes of two separate images and performing reconstruction through the IFFT. The resulting reconstructed images contained content from both images and varied only in image intensity.

References

[1] Fourier transform - opencv 3.0.0-dev documentation.

[2] Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, and Li Fei-Fei. Novel dataset for fine-grained image categorization. In *First Workshop on Fine-Grained Visual Categorization, IEEE Conference on Computer Vision and Pattern Recognition*, Colorado Springs, CO, June 2011.

7 Appendix

7.1 utils.py

```
import numpy as np
from math import log10

# Convolve over image
def conv(image, kernel):
    # Assuming MxM kernel
    M = len(kernel)
    if M < 1:
        return image

    output = np.zeros_like(image)

    # Apply edge padding
    # Duplicates the last element in each axis by padding amount
    original_shape = image.shape
    image = np.pad(image, [(M/2,M/2) ,(M/2,M/2)] , 'edge')

    # Convolve filter
    kernel = np.flip(kernel, axis=0)
    kernel = np.flip(kernel, axis=1)

    # Operate over entire image
    for x in range(original_shape[0]):
        for y in range(original_shape[1]):
            mult = np.multiply(kernel, image[x:x+M, y:y+M])
            output[x,y] = np.sum(mult, axis=(0,1))

    return output

def median(image, M):
    if M < 1:
        return image

    output = np.zeros_like(image)

    # Apply edge padding
    # Duplicates the last element in each axis by padding amount
    original_shape = image.shape
    image = np.pad(image, [(M/2,M/2) ,(M/2,M/2)] , 'edge')

    # Operate over entire image
    for x in range(original_shape[0]):
        for y in range(original_shape[1]):
            output[x,y] = np.median(image[x:x+M, y:y+M])

    return output

def snr(original, noisy):
    original_var = np.var(original)
    noise_var = np.var(noisy)
    return 10 * log10(original_var/noise_var)

def mse(original, new):
    (n, m) = original.shape
    diff = original - new
    error = np.sum(np.square(diff))
    return error / float(m * n)

def circle_mask(img, radius):
    (n, m) = img.shape
    # Find center of image
    center = (m/2, n/2)
    # Generate indices for all coordinates in the image
    idy, idx = np.ogrid[:n, :m]
    # Calculate distances between indices and the image center
    distances = np.sqrt((idx - center[0])**2 + (idy - center[1])**2)
    # Create a mask for indices within the given radius
    mask = distances <= (n / radius)
    # Apply mask to the given image and return
    return np.multiply(mask, img)

def threshold(img, thresh):
    img[img < thresh] = 0
    img[img >= thresh] = 255
    return img
```

7.2 part1.py

```
import numpy as np
import cv2
import time
from utils import conv, threshold

EXPORT_IMAGES = True

image = cv2.imread("../data/pup.jpg", cv2.IMREAD_GRAYSCALE) / 255.0

# 5x5 averaging filter kernel (low pass)
```

```
avg_kernel = np.ones((5,5)) / 25.0

# 5x5 high pass filter
high_pass_kernel = [[-1, -1, -1, -1, -1],
                    [-1, -1, -1, -1, -1],
                    [-1, -1, 24, -1, -1],
                    [-1, -1, -1, -1, -1],
                    [-1, -1, -1, -1, -1]]
high_pass_kernel = np.array(high_pass_kernel) / 25.0

# Sobel edge detection filters
sx_kernel = [[-1,0,1],
             [-2,0,2],
             [-1,0,1]]

sy_kernel = [[1,2,1],
             [0,0,0],
             [-1,-2,-1]]

# High pass filter
conv_start = time.time()
high = conv(image, high_pass_kernel)
conv_end = time.time()
print("Highpass␣execution␣time␣(s):␣" + str(conv_end - conv_start))

# Low pass filter
low = conv(image, avg_kernel)
low2 = conv(low, avg_kernel)
low3 = conv(low2, avg_kernel)


# Sobel edge filter
sx = np.abs(conv(image, sx_kernel))
sx = threshold(sx, 0.3)
sy = np.abs(conv(image, sy_kernel))
sy = threshold(sy, 0.3)
sobel = sx + sy
sobel[sobel > 255] = 255

if EXPORT_IMAGES:
    cv2.imwrite("../figures/part1/high_pup.jpg", high * 255)
    cv2.imwrite("../figures/part1/low_pup.jpg", low * 255)
    cv2.imwrite("../figures/part1/low2_pup.jpg", low2 * 255)
    cv2.imwrite("../figures/part1/low3_pup.jpg", low3 * 255)
    cv2.imwrite("../figures/part1/edgex_pup.jpg", sx * 255)
    cv2.imwrite("../figures/part1/edgey_pup.jpg", sy * 255)
    cv2.imwrite("../figures/part1/edge_pup.jpg", sobel * 255)
```

7.3 part2.py

```
import numpy as np
import cv2
from skimage.util import random_noise
import matplotlib
from utils import conv, median, snr, threshold

EXPORT_IMAGES = True

image = cv2.imread("../data/pup.jpg", cv2.IMREAD_GRAYSCALE) / 255.0

# Generate salt and pepper noise
seasoned_image = random_noise(image, mode='s&p', seed=0)
seasoned_snr = snr(image, seasoned_image)
print("Salt and pepper SNR: " + str(seasoned_snr) + " dB")

# Generate Gaussian noise
gaussed_image = random_noise(image, mode='gaussian', seed=0)
gaussed_snr = snr(image, gaussed_image)
print("Gaussian SNR: " + str(gaussed_snr) + " dB")

# Apply a median filter over image
# 5x5 averaging filter kernel (low pass)
avg_kernel = np.ones((5,5)) / 25.0
averaged_simage = conv(seasoned_image, avg_kernel)
averaged_gimage = conv(gaussed_image, avg_kernel)


# Apply a median filter over image
median_simage = median(seasoned_image, 5)
median_gimage = median(gaussed_image, 5)

# Sobel edge detection filters
sx_kernel = [[-1,0,1],
             [-2,0,2],
             [-1,0,1]]

sy_kernel = [[1,2,1],
             [0,0,0],
             [-1,-2,-1]]

# Sobel edge filter on noisy images
sx = np.abs(conv(seasoned_image, sx_kernel))
sx = threshold(sx, 0.3)
sy = np.abs(conv(seasoned_image, sy_kernel))
sy = threshold(sy, 0.3)
sobel_simage = sx + sy
sx = np.abs(conv(gaussed_image, sx_kernel))
sx = threshold(sx, 0.3)
sy = np.abs(conv(gaussed_image, sy_kernel))
sy = threshold(sy, 0.3)
sobel_gimage = sx + sy

# Sobel edge filter on pre-filtered noisy images
sx = np.abs(conv(median_simage, sx_kernel))
sx = threshold(sx, 0.3)
sy = np.abs(conv(median_simage, sy_kernel))
sy = threshold(sy, 0.3)
sobel_msimage = sx + sy
sx = np.abs(conv(averaged_gimage, sx_kernel))
sx = threshold(sx, 0.3)
sy = np.abs(conv(averaged_gimage, sy_kernel))
sy = threshold(sy, 0.3)
sobel_lgimage = sx + sy

if EXPORT_IMAGES:
    cv2.imwrite("../figures/part2/seasoned_pup.jpg", seasoned_image * 255)
    cv2.imwrite("../figures/part2/gaussed_pup.jpg", gaussed_image * 255)
    cv2.imwrite("../figures/part2/averaged_spup.jpg", averaged_simage * 255)
    cv2.imwrite("../figures/part2/averaged_gpup.jpg", averaged_gimage * 255)
    cv2.imwrite("../figures/part2/median_spup.jpg", median_simage * 255)
    cv2.imwrite("../figures/part2/median_gpup.jpg", median_gimage * 255)
    cv2.imwrite("../figures/part2/sobel_spup.jpg", sobel_simage * 255)
    cv2.imwrite("../figures/part2/sobel_gpup.jpg", sobel_gimage * 255)
    cv2.imwrite("../figures/part2/sobel_mspup.jpg", sobel_msimage * 255)
    cv2.imwrite("../figures/part2/sobel_lgpup.jpg", sobel_lgimage * 255)
```

7.4 part3.py

```
import numpy as np
import cv2
from matplotlib import pyplot as plt
import csv
from utils import mse, circle_mask

image = cv2.imread("../data/pup.jpg", cv2.IMREAD_GRAYSCALE) / 255.0
fft = np.fft.fft2(image)

# Remove plt frame
```

```
x = plt.axes([0,0,1,1], frameon=False)
x.get_xaxis().set_visible(False)
x.get_yaxis().set_visible(False)

### A
# Unshifted
plt.imshow(np.abs(fft), cmap = 'gray')
#plt.title('Magnitude'),
plt.xticks([]), plt.yticks([])
plt.savefig("../figures/part3/fft.png", transparent=True)

# # Shifted
fft_shift = np.fft.fftshift(fft)
plt.imshow(np.abs(fft_shift), cmap = 'gray')
#plt.title('Magnitude Shifted'),
plt.xticks([]), plt.yticks([])
plt.savefig("../figures/part3/fft_shift.png", transparent=True)

# Log applied
fft_log = 20*np.log(np.abs(fft_shift))
plt.imshow(fft_log, cmap = 'gray')
#plt.title('Log Magnitude Shifted'),
plt.xticks([]), plt.yticks([])
plt.savefig("../figures/part3/fft_log.png", transparent=True)

### B
# Inverse
reconstructed = np.abs(np.fft.ifft2(fft_shift))
plt.imshow(reconstructed, cmap='gray')
#plt.title('Reconstructed Image'),
plt.xticks([]), plt.yticks([])
plt.savefig("../figures/part3/recon.png", transparent=True)

### C
# Radius of N/3
freq_3 = circle_mask(fft_shift, 3)
recon_3 = np.abs(np.fft.ifft2(freq_3))# + np.abs(fft_shift)))
plt.imshow(recon_3, cmap='gray')
#plt.title('Reconstructed Image (N/3)'),
plt.xticks([]), plt.yticks([])
plt.savefig("../figures/part3/recon_3.png", transparent=True)

# Radius of N/4
freq_4 = circle_mask(fft_shift, 4)
recon_4 = np.abs(np.fft.ifft2(freq_4))# + np.abs(fft_shift)))
plt.imshow(recon_4, cmap='gray')
```

```
#plt.title('Reconstructed Image (N/4)'#plt.title
plt.xticks([]), plt.yticks([])
plt.savefig("../figures/part3/recon_4.png", transparent=True)

# Radius of N/8
freq_8 = circle_mask(fft_shift, 8)
recon_8 = np.abs(np.fft.ifft2(freq_8))# + np.abs(fft_shift)))
plt.imshow(recon_8, cmap='gray')
#plt.title('Reconstructed Image (N/8)'
plt.xticks([]), plt.yticks([])
plt.savefig("../figures/part3/recon_8.png", transparent=True)

# Radius of N/16
freq_16 = circle_mask(fft_shift, 16)
recon_16 = np.abs(np.fft.ifft2(freq_16))# + np.abs(fft_shift)))
plt.imshow(recon_16, cmap='gray')
#plt.title('Reconstructed Image (N/16)'),
plt.xticks([]), plt.yticks([])
plt.savefig("../figures/part3/recon_16.png", transparent=True)

# Compile MSE results
mse_results = []
mse_results.append(("Unaltered",
                    '{:0.3e}'.format(mse(image, reconstructed))))
mse_results.append(("N/3", '{:0.3e}'.format(mse(image, recon_3))))
mse_results.append(("N/4", '{:0.3e}'.format(mse(image, recon_4))))
mse_results.append(("N/8", '{:0.3e}'.format(mse(image, recon_8))))
mse_results.append(("N/16", '{:0.3e}'.format(mse(image, recon_16))))

# Export results to csv
with open('../figures/part3/results.csv', 'wb') as f:
    writer = csv.writer(f)
    writer.writerow(['reconstruction', 'mse'])
    for (recon, value) in mse_results:
        writer.writerow([recon, value])
```

7.5 part4.py

```
import numpy as np
import cv2
from matplotlib import pyplot as plt
from utils import mse

# Remove plt frame
x = plt.axes([0,0,1,1], frameon=False)
x.get_xaxis().set_visible(False)
```

```
x.get_yaxis().set_visible(False)

image1 = cv2.imread("../data/n02091032_561.jpg",
                    cv2.IMREAD_GRAYSCALE) / 255.0
fft1 = np.fft.fft2(image1)
fft1_shift = np.fft.fftshift(fft1)
fft1_real_log = 20*np.log(np.abs(fft1_shift))
fft1_imag_log = 20*np.log(fft1_shift.imag)


image2 = cv2.imread("../data/n02091134_755.jpg",
                    cv2.IMREAD_GRAYSCALE) / 255.0
fft2 = np.fft.fft2(image2)
fft2_shift = np.fft.fftshift(fft2)
fft2_real_log = 20*np.log(np.abs(fft2_shift))
fft2_imag_log = 20*np.log(fft2_shift.imag)

### A
# Image 1 FFT
plt.imshow(fft1_real_log, cmap = 'gray')
plt.xticks([]), plt.yticks([])
plt.savefig("../figures/part4/fft1_mag.png", transparent=True)

plt.imshow(fft1_imag_log, cmap = 'gray')
plt.xticks([]), plt.yticks([])
plt.savefig("../figures/part4/fft1_phase.png", transparent=True)

# Image 2 FFT
plt.imshow(fft2_real_log, cmap = 'gray')
plt.xticks([]), plt.yticks([])
plt.savefig("../figures/part4/fft2_mag.png", transparent=True)

plt.imshow(fft2_imag_log, cmap = 'gray')
plt.xticks([]), plt.yticks([])
plt.savefig("../figures/part4/fft2_phase.png", transparent=True)

### B
recon_1 = np.abs(np.fft.ifft2(fft2.real + (1j * fft1.imag)))
print("Recon_1_mse: " + str(mse(recon_1, image1)))
recon_2 = np.abs(np.fft.ifft2(fft1.real + (1j * fft2.imag)))
print("Recon_2_mse: " + str(mse(recon_2, image2)))

plt.imshow(recon_1, cmap = 'gray')
plt.xticks([]), plt.yticks([])
plt.savefig("../figures/part4/fft1_recon.png", transparent=True)
plt.imshow(recon_2, cmap = 'gray')
plt.xticks([]), plt.yticks([])
plt.savefig("../figures/part4/fft2_recon.png", transparent=True)
```