HW3_AD

October 16, 2024

[1]:
```python
import numpy as np
from skimage import io, filters
from skimage.color import rgb2gray
from scipy.ndimage import convolve
import matplotlib.pyplot as plt
```

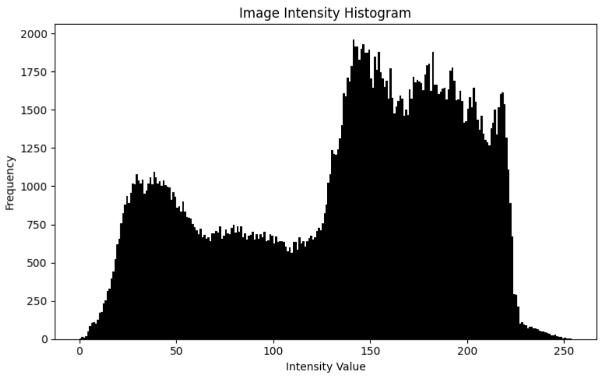
1) Convolution and filtering

Find an image and make it grayscale (2D) if it isn't already using whatever means you like. Also make an 11×11 square array in which all the elements are the same. Perform a local averaging of the image by convolving it with the 11×11 array. In Python use convolve in the scipy.ndimage package. In MATLAB use imfilter; this is similar to conv2 but allows clearer statements about what to do at the borders. (There are other subtle differences between the two.) Try at least two different choices for what the convolution does near and beyond the image borders, one of which is zero-padding. Submit a "Zoomed in" image for each choice, and comment on whether they are doing what you expect.

Suggestion: Pick an image that isn't too large (maybe 600x600?), so that you can see the effects of the filtering more clearly. Or you can make a larger kernel – not 11×11, but something like 51×51, but be warned that this may be slow. Comment: The "Zoomed in" image should zoom in to the edge, of course, so you can see the difference between edge options. Comment: Recall that "zero padding" means filling all the "new" pixels beyond the edge with zeros.

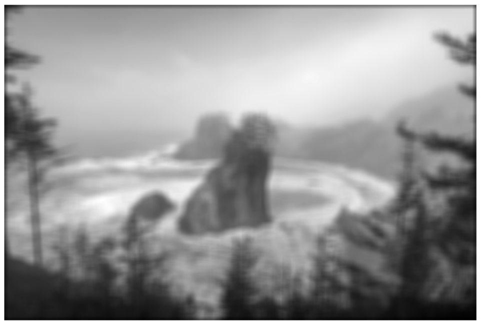
[2]:
```python
def showim(im_array, figsize = (11,5), show_hist = False, cmap = 'gray', vmin =␣
 ↪None, vmax = None):

    if show_hist:
        im_flattened = im_array.ravel()
        number_of_bins = (np.max(im_flattened) - np.min(im_flattened))

        plt.figure(figsize=figsize)
        plt.subplot(1,2,1)
        plt.imshow(im_array, cmap=cmap, vmin=vmin, vmax=vmax)
        plt.axis('off')

        plt.subplot(1,2,2)
        plt.hist(im_flattened, bins=number_of_bins, color='black')
        plt.xlabel('Intensity Value')
        plt.ylabel('Frequency')
        plt.title('Image Intensity Histogram')

        plt.tight_layout()

    else:
        plt.figure(figsize=figsize)
        plt.imshow(im_array, cmap=cmap, vmin=vmin, vmax=vmax)
        plt.axis('off')

def thresh_otsu(im_array, show_results = False):

    if show_results:

        plt.figure(figsize=(15,7))

        otsu_thresh = filters.threshold_otsu(im_array)
        image_otsu_thresh = (im_array > otsu_thresh).astype(np.uint8)
        plt.subplot(1,2,1)
        plt.imshow(image_otsu_thresh, cmap='gray')
        plt.axis('off')

        im_array_flattened = im_array.ravel()

        plt.subplot(1,2,2)
        plt.hist(im_array_flattened, bins=256, color='black')
        plt.axvline(otsu_thresh, color='red', linestyle='dashed', linewidth=2)
        plt.title('Image Intensity Histogram')
        plt.xlabel('Pixel Intensity')
        plt.ylabel('Frequency')

    else:
        otsu_thresh = filters.threshold_otsu(im_array)
        image_otsu_thresh = (im_array > otsu_thresh).astype(np.uint8)

    return image_otsu_thresh
```

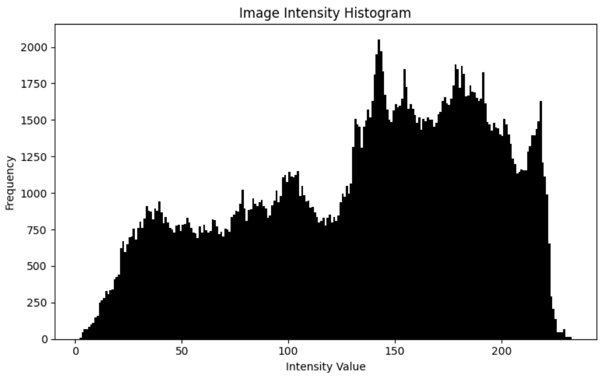
[3]:
```
coast_image = (rgb2gray(io.imread('/home/apd/Projects/ImageAnalysis/HW3/
 ↪Oregon-Coast-1694.jpg'))*255).astype(np.uint8)
showim(coast_image, figsize=(15,5),show_hist=True)
```

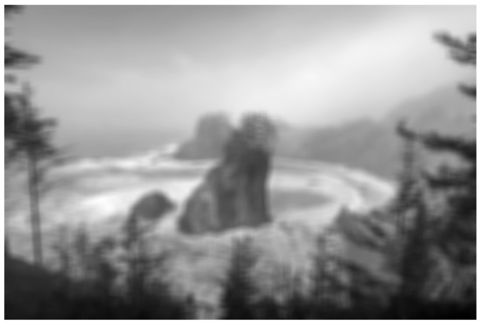


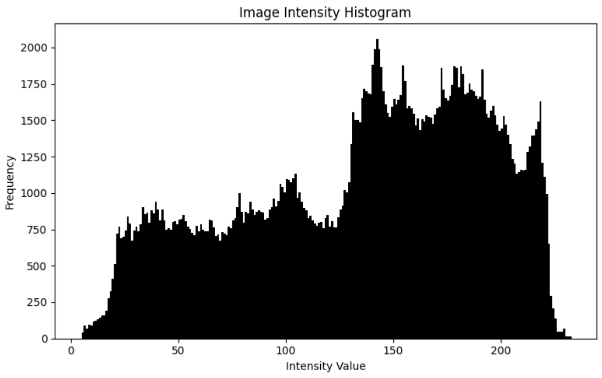
[4]:
```
kernel_size = 11
square_kernel = np.ones((kernel_size, kernel_size)) / (kernel_size**2)

conv_zero_pad = convolve(coast_image, square_kernel, mode='constant', cval=0)
conv_reflect = convolve(coast_image, square_kernel, mode='reflect')
```

[5]:
```
showim(conv_zero_pad, figsize=(15,5), show_hist=True)
showim(conv_reflect, figsize=(15,5), show_hist=True)
```

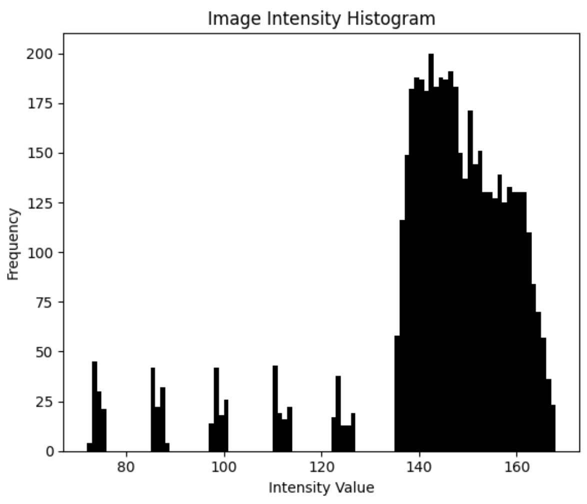


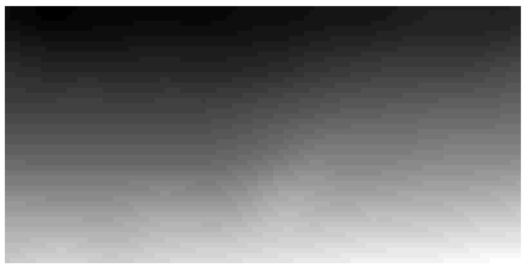


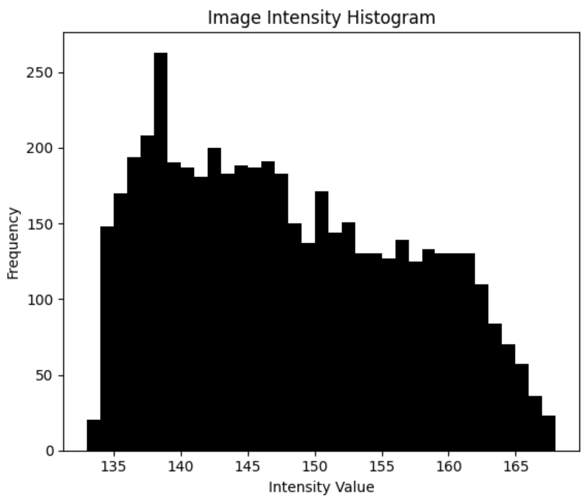




[6]:
```
zoom_ROI_1 = (slice(0, 50), slice(100, 200))
zoom_1 = conv_zero_pad[zoom_ROI_1]
showim(zoom_1, show_hist=True)

zoom_ROI_2 = (slice(0, 50), slice(100, 200))
zoom_2 = conv_reflect[zoom_ROI_2]
showim(zoom_2, show_hist=True)
```







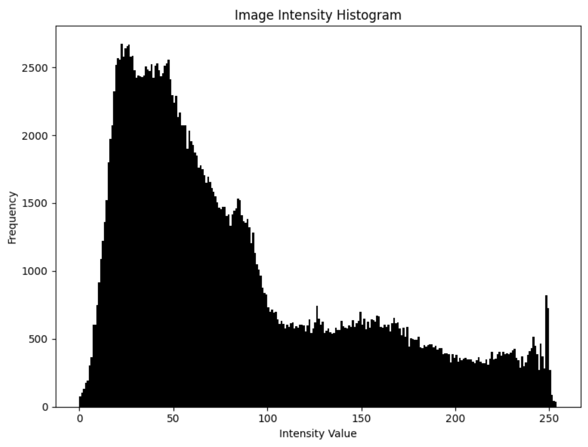
Both of these zooms are taken from the top left of the image, showing the top border. With zero padding, substitute values are just 0, which is apparent when looking at a histogram of the values near the border (we have discrete little peaks representing each horizontal bar as the kernel shifts upwards and more zeros need to be sibstituted in). The reflect seems to just reflect back the nearest pixel values, giving a much smoother distribution.

2 Gaussian filtering

Find a grayscale image or make a color image grayscale however you like; choose an image that has some fairly sharp edges. We'll perform Gaussian filtering of this image. Note that a Gaussian function is exp(– (^2) / 2(^2)) where r is the distance to the center.

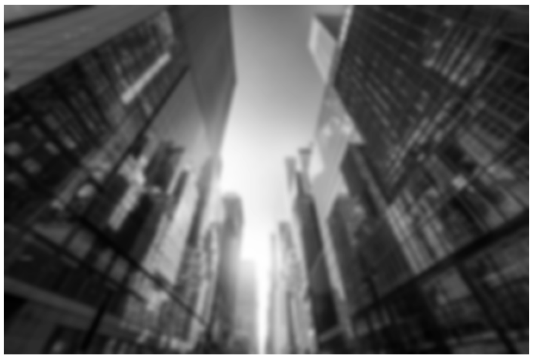
a) Gaussian-filter the image by explicitly creating a Gaussian kernel and convolving it with your image as in Problem 1. Show your code for making the kernel. (Hint: use meshgrid for x and y positions, then figure out "r".) Show the original image, an image of your kernel and the output of the convolution for two different values of the Gaussian width, .

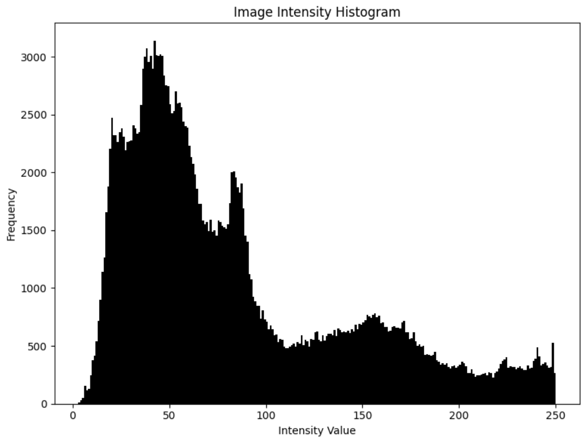
```
[7]: street_image = (rgb2gray(io.imread('/home/apd/Projects/ImageAnalysis/HW3/
       ↪street2.jpg'))*255).astype(np.uint8)
     showim(street_image, show_hist=True, figsize=(15,6))
```

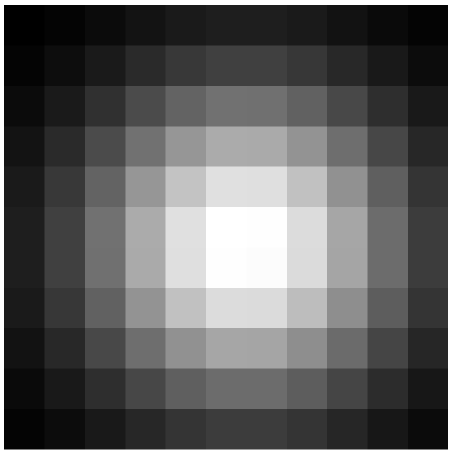


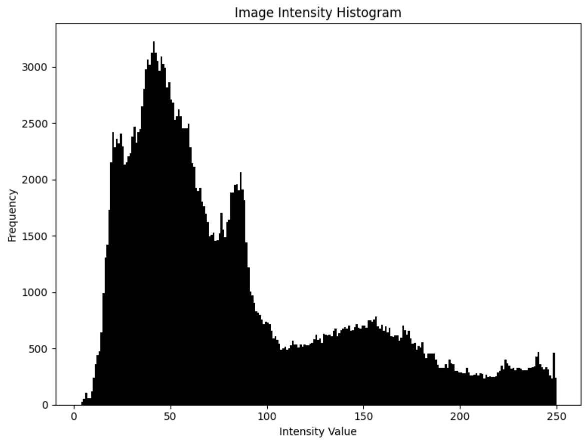
```
[8]: def create_gauss_kernel(kernel_size, sigma):

         length = np.linspace(-kernel_size//2, kernel_size//2, kernel_size)

         x, y = np.meshgrid(length, length)
         r = np.sqrt(x**2 + y**2)
         kernel = np.exp(-(r**2) / (2 * sigma**2))
         return kernel / kernel.sum()

     small_gauss_kernel = create_gauss_kernel(11,3)
     street_conv_1 = convolve(street_image, small_gauss_kernel)
     showim(street_conv_1, show_hist=True, figsize=(15,6))
     showim(small_gauss_kernel, figsize=(3,3))

     wider_gauss_kernel = create_gauss_kernel(11,9)
```

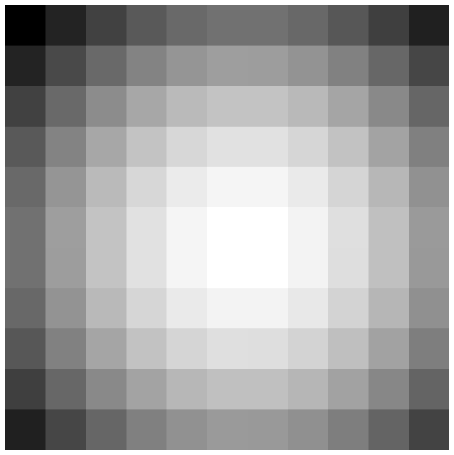
```python
street_conv_2 = convolve(street_image, wider_gauss_kernel)
showim(street_conv_2, show_hist=True, figsize=(15,6))
showim(wider_gauss_kernel, figsize=(3,3))
```



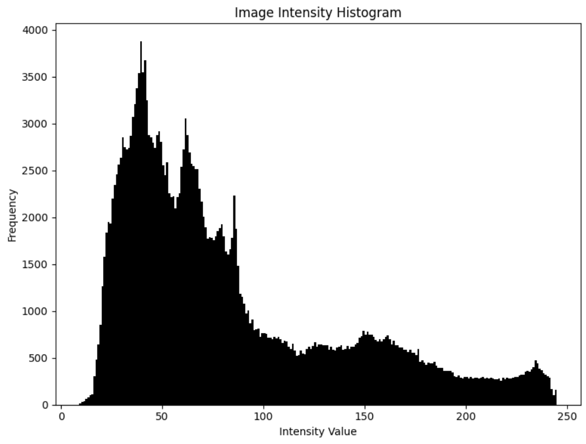




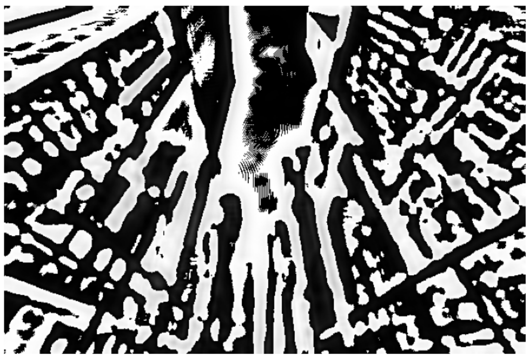


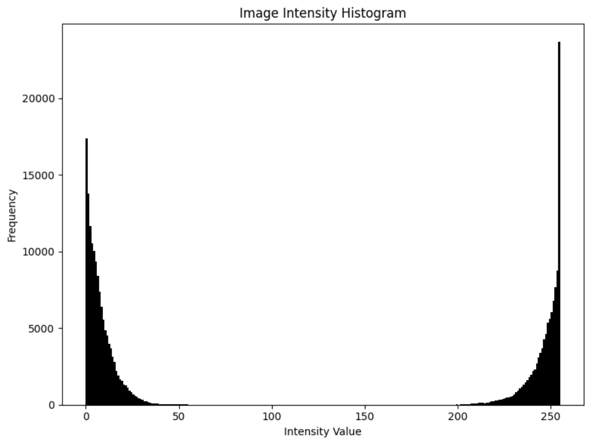


```
[9]: from skimage.filters import gaussian
     new_gauss = (gaussian(street_image, sigma=9)*255).astype(np.uint8)
     showim(new_gauss, show_hist=True, figsize=(15,6))
```



```
[10]: difference = new_gauss-street_conv_2
      showim(difference, show_hist=True, figsize=(15,6))
```





It seems like the convolved images mainly differ around the edges of the buildings, or sharp edges. After doing some reading, it seems like skimage doesn't use an actual kernel thats an array but performs efficient mathematical operations on the image, meaning it likely doesn't have to "clip" its gaussian kernel like we do when we choose an explicit size (not sigma) for it. In that way, skimage's gaussian convolution is probably more mathematically accurate as it is able to incorporate more image information into it's calculations.

```
[11]: import time
      manual_to = time.time()
      manual_conv = convolve(street_image, wider_gauss_kernel)
      manual_tf = time.time()

      skimage_to = time.time()
```

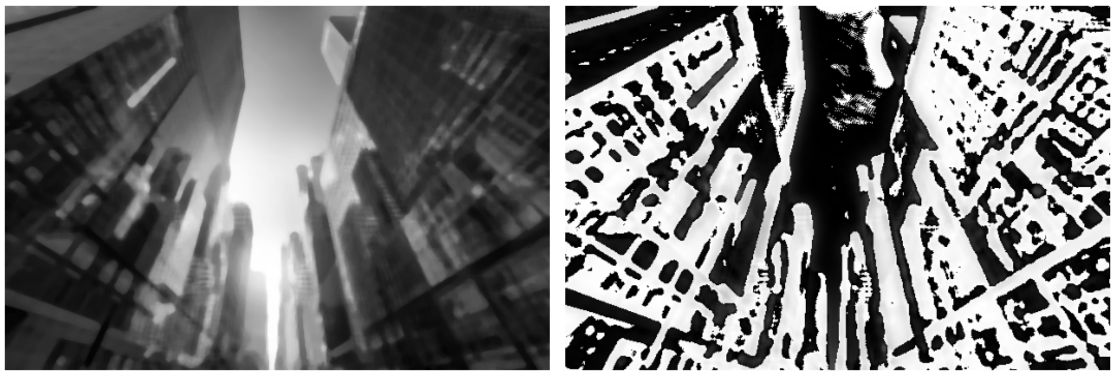
```
skimage_conv = gaussian(street_image, sigma=9)
skimage_tf = time.time()

print(f"manual time: {manual_tf-manual_to}")
print(f"skimage time: {skimage_tf-skimage_to}")
```

```
manual time: 0.033333539962768555
skimage time: 0.014143943786621094
```

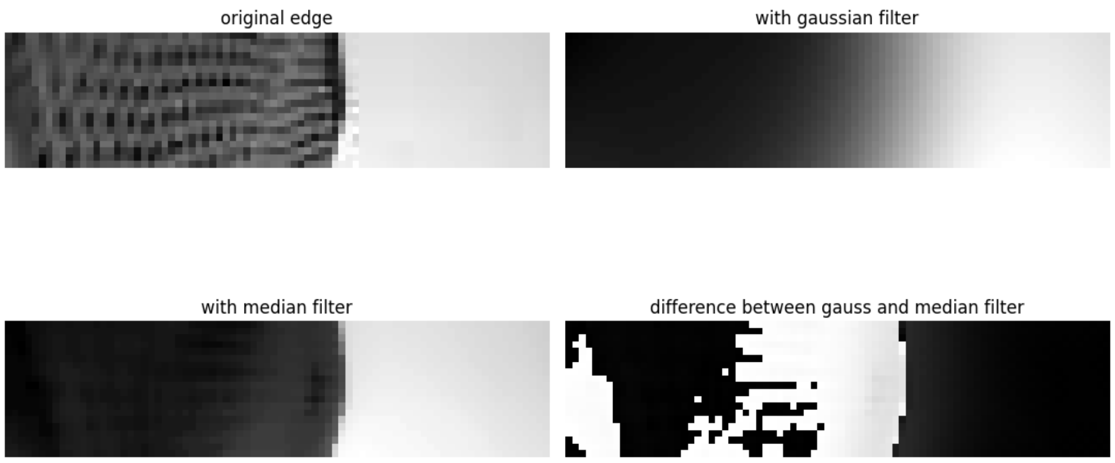
By decomposing a 2-D seperable kernel (matrix) into two 1-D kernels (rows), the results of the application of each of these 1-D kernels put together reconstructs the explicit application of the 2-D convolution with less computations. It scales much better with increasing kernel sizes! In the this problem, skimage likely uses this method to convolve the image, where my manual version is slower as it sticks to explicit computation off of a 2-D kernel.

[12]:
```
from skimage.filters import median

median_t0 = time.time()
median_conv = median(street_image, np.ones((9,9)))
median_tf = time.time()

difference = median_conv - new_gauss

print(f"median time: {median_tf-median_t0}")

plt.figure(figsize=(11,7))
plt.subplot(1,2,1)
plt.imshow(median_conv, cmap = 'gray')
plt.axis('off')
plt.subplot(1,2,2)
plt.imshow(difference, cmap = 'gray')
plt.axis('off')
plt.tight_layout()
```

```
median time: 0.1944108009338379
```



[13]:
```python
street_zoom = (slice(80, 100), slice(220, 300))

plt.figure(figsize=(11,7))

plt.subplot(2,2,1)
plt.imshow(street_image[street_zoom], cmap = 'gray')
plt.title('original edge')
plt.axis('off')

plt.subplot(2,2,2)
plt.imshow(new_gauss[street_zoom], cmap = 'gray')
plt.title('with gaussian filter')
plt.axis('off')

plt.subplot(2,2,3)
plt.imshow(median_conv[street_zoom], cmap = 'gray')
plt.title('with median filter')
plt.axis('off')

plt.subplot(2,2,4)
plt.imshow(difference[street_zoom], cmap = 'gray')
plt.title('difference between gauss and median filter')
plt.axis('off')
plt.tight_layout()
```

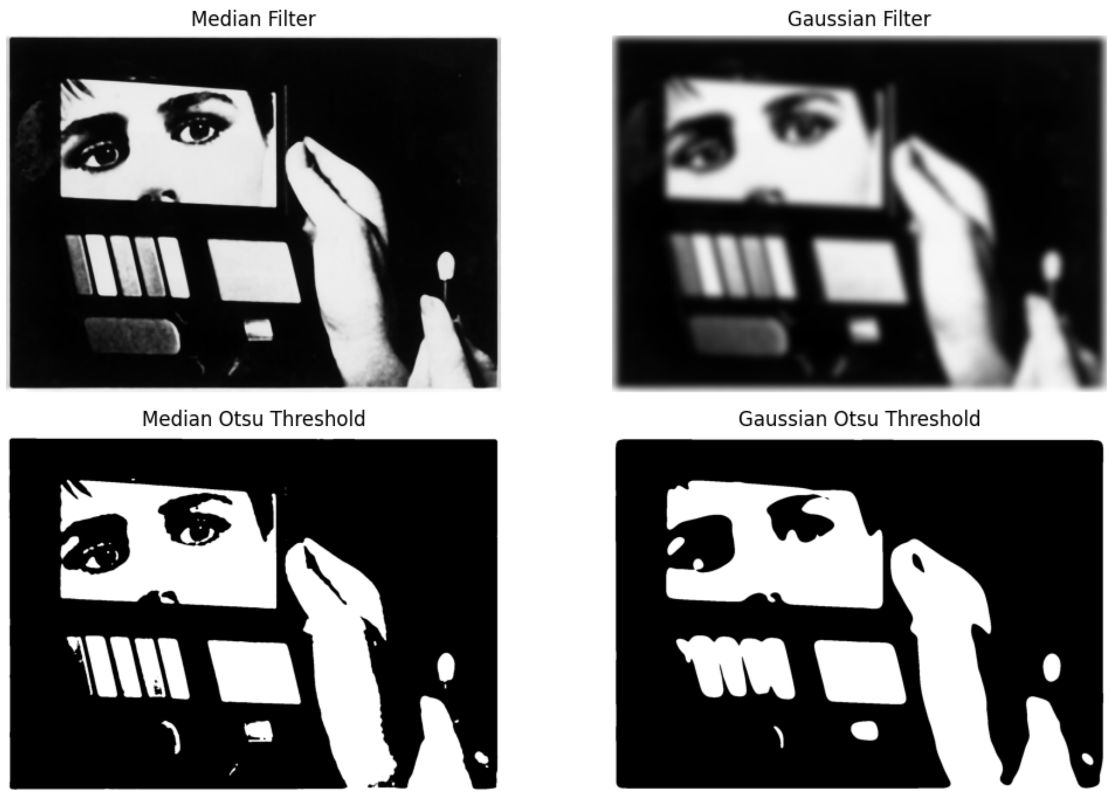


Again the differences seem to be at the edges. The median filter seems to make either side of the edge more pronounced, it's either very dark or light on either side of the edge. The gaussian filter smooths the edge out, essentially blurring it. The median filter also takes much longer, about and order of magnitude for a kernel size of 11x11!

4) Filtering and thresholding

Apply (separately) a Gaussian and a median filter to the photo "MakeUp_RichardPrince_1983_gray.png" (on Canvas), and then threshold the resulting images using Otsu's method. Use a similar median filter size as the Gaussian width. Comment on the appearance of the thresholded images.

[14]:
```python
makeup_image = io.imread('/home/apd/Projects/ImageAnalysis/HW3/
 ↪MakeUp_RichardPrince_1983_gray.png')

makeup_median = median(makeup_image, np.ones((11,11)))
makeup_gauss = gaussian(makeup_image, sigma = 11)

mm_thresh = thresh_otsu(makeup_median)
mg_thresh = thresh_otsu(makeup_gauss)

plt.figure(figsize=(11,7))

plt.subplot(2,2,1)
plt.imshow(makeup_median, cmap = 'gray')
plt.title('Median Filter')
plt.axis('off')

plt.subplot(2,2,2)
plt.imshow(makeup_gauss, cmap = 'gray')
plt.title('Gaussian Filter')
plt.axis('off')

plt.subplot(2,2,3)
plt.imshow(mm_thresh, cmap = 'gray')
plt.title('Median Otsu Threshold')
plt.axis('off')

plt.subplot(2,2,4)
plt.imshow(mg_thresh, cmap = 'gray')
plt.title('Gaussian Otsu Threshold')
plt.axis('off')
plt.tight_layout()
```



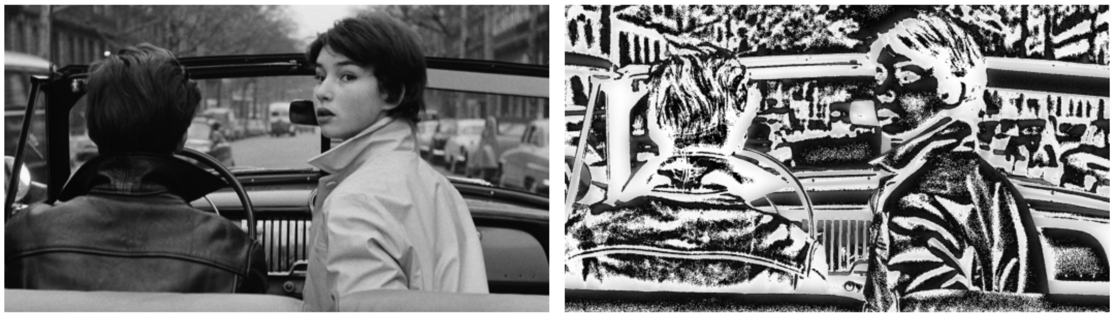
The threshold on the median image seems more defined, while the threshold on the gaussian image seems to deliver a more smooth, abstract image (it almost looks like it just captures the photos). This lines up with the operations of each kernel, and results from the last problem.

5) High-pass filtering

As noted in class, a Gaussian filter is a "low pass" filter, retaining gently-varying (low spatial frequency) changes in intensity and smoothing close-together changes (high spatial frequency). Any image is a combination of low- and high-frequency components. (We can be more exact about this using the Fourier analysis; we may revisit this later.) Therefore, if we subtract a Gaussian-filtered image from the original image, we're left with the high-frequency components. This subtraction (the "identity filter" minus a Gaussian filter) is a high-pass filter. Download "Elevator_to_the_gallows.png" and create a high-pass-filtered image with Gaussian width 21. Submit the image and comment on its appearance. Note the intensity range: if you're subtracting double-precision numbers, you may wish to set negative values equal to zero

[15]:
```
gallows_image = io.imread('/home/apd/Projects/ImageAnalysis/HW3/
  ↪Elevator_to_the_gallows.png')
gallows_gaussian = (gaussian(gallows_image, sigma=21)*255).astype(np.uint8)
#gallows_highpass = (np.ones_like(gallows_gaussian)*255).astype(np.
  ↪uint8)-gallows_gaussian
gallows_highpass = gallows_image - gallows_gaussian
```

```
plt.figure(figsize=(11,7))
plt.subplot(1,2,1)
plt.imshow(gallows_image, cmap = 'gray')
plt.axis('off')
plt.subplot(1,2,2)
plt.imshow(gallows_highpass, cmap = 'gray')
plt.axis('off')
plt.tight_layout()
```



Areas where there isn't much variation, such as the road, areas of the jackets with the same shading, and the sky all don't "pass" through the filter, whereas finer and more varied sections such as hair, trees, and the complex shadows of the jackets, all pass through the filter.

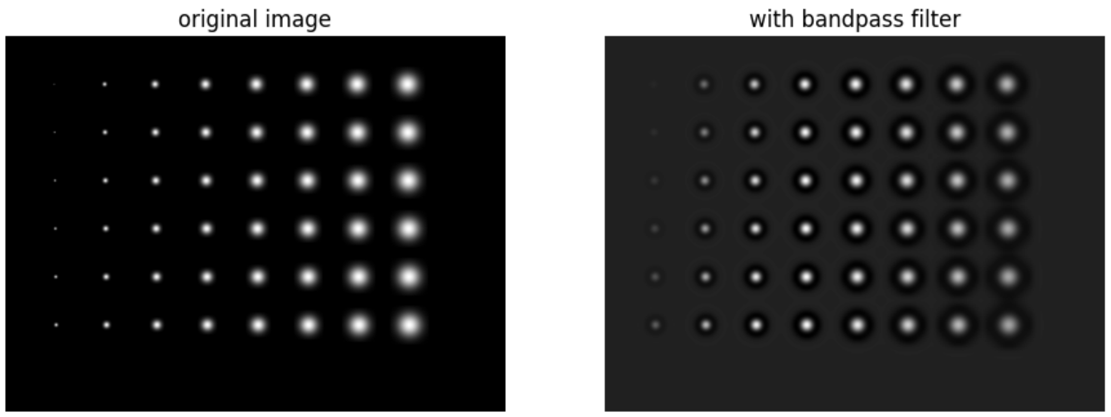
6 Band-pass filtering

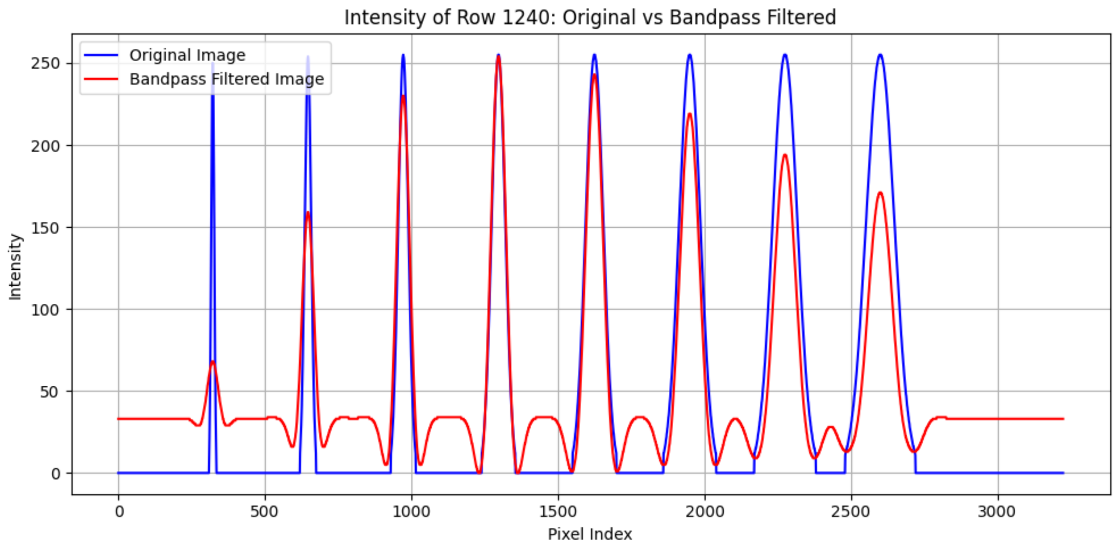
We've now seen low-pass and high-pass filtering. We can combine these to make a band-pass filter which, as you might guess, keeps middle-frequency components of an image. Load the image "gaussians_s2_to_s50_px.tif," which is a grid of Gaussians with varying width, from about = 2 to 50 pixels. Figure out a series of low or high pass filtering and subtraction steps that will diminish the intensity of the "big" and the "small" dots, leaving the "medium" dots relatively intact. ("Relatively" is important: you won't perfectly reject or keep dots, but the middle columns should be attenuated less than the outer ones.) Describe your procedure in words, include your code. Submit an image of the final band-pass filtered image. Also submit a plot of intensity vs. column number (i.e. "x") for row 1240, as indicated below, including both the original and the band- pass-filtered images

[34]:
```
from tqdm.auto import tqdm

def normalize_uint8(array):
    array = array +  abs(np.min(array))
    array = (array / np.max(array)) * 255
    return array.astype(np.uint8)

def invert_array(arr):
    return 255 - arr
```

```python
garray = io.imread('/home/apd/Projects/ImageAnalysis/HW3/gaussians_s2_to_s50_px.
 ↪tif').astype(np.float32)
identity_arr = (np.ones_like(garray)*255).astype(np.float32)

highpass = (gaussian(garray, sigma=30) * 255).astype(np.float32)
lowpass = (gaussian(garray, sigma=20) * 255).astype(np.float32)

'''
garray_highpass = normalize_uint8((garray - highpass))
garray_lowpass = normalize_uint8((lowpass))
garray_bandpass = normalize_uint8(garray- highpass + lowpass)
'''

kernel_list = [25, 25, 25, 40, 30, 20, 20]

garray_bandpass = np.zeros_like(garray)

for kernel_size in tqdm(kernel_list):
    lowpass = (gaussian(garray, sigma=kernel_size) * 255).astype(np.float32)
    bandpass = invert_array(highpass - lowpass)

    garray_bandpass += bandpass

garray_bandpass = normalize_uint8(garray_bandpass)

plt.figure(figsize=(11,7))

plt.subplot(1,2,1)
plt.imshow(garray, cmap = 'gray')
plt.title('original image')
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(garray_bandpass, cmap = 'gray')
plt.title('with bandpass filter')
plt.axis('off')

row_original = garray[1240, :]
row_bandpass = garray_bandpass[1240, :]

# Plot the intensity values
plt.figure(figsize=(10,5))

plt.plot(row_original, label='Original Image', color='blue')
plt.plot(row_bandpass, label='Bandpass Filtered Image', color='red')

plt.title('Intensity of Row 1240: Original vs Bandpass Filtered')
plt.xlabel('Pixel Index')
plt.ylabel('Intensity')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
100%|      | 7/7 [00:08<00:00,  1.24s/it]
```





Since a gaussian convolution acts as a lowpass filter, if we subtract it from the original image we get a highpass filter as a result. Thus, I subtracted a higher larger kernel size convolution from the image (what I'm calling highpass) while also adding a smaller kernel size convolution kernel (what I'm calling lowpass) to the image to effectively pass it through both a low and high pass filter. Looking at the peaks in the plot above, we can see it works somewhat, although I'm still having trouble with filtering out some of the right-most peaks.

7) Signal to Noise Ratio

The first part of this problem is a useful first step towards next week's simulated single molecule images, and the second part revisits the notion of the signal-to- noise ratio.

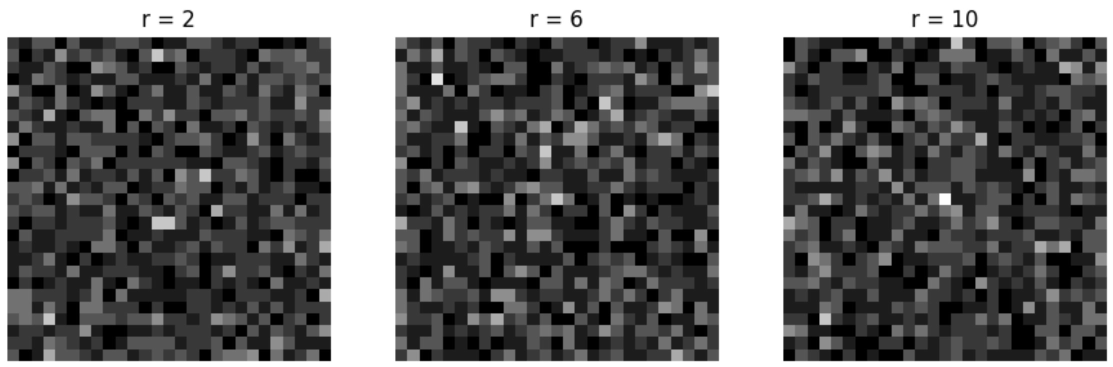
Suppose your background (photons plus dark noise) is on average b = 2 intensity units per millisecond at each pixel, and readout noise is negligible. The signal you care about gives, on average, r intensity units per millisecond, all of which go to the center pixel of a 27 x 27 pixel sensor array. The signal is also variable and is also Poisson distributed; in other words, if r = 4, we expect in t = 2 ms an average intensity of r_t = 8, but the number is actually a random number drawn from a Poisson distribution.

a) Show one example each of the output for r = 2, 6, and 10 units/ms and an exposure time of 1 millisecond. (You can show them each scaled to the full range of display intensity, which is the default for matplotlib.imshow in Python or keep the same range for all by setting vmin and vmax.) .

[18]:
```
from numpy.random import poisson
```

[226]:
```
def generate_simage(r, b = 2, t = 1, sensor_size = (27,27)):

    simage = poisson(lam = b*t, size = sensor_size) # generates background data
    signal_output = poisson(lam=r*t) # single point of signal data
    sensor_center = (sensor_size[0] // 2, sensor_size[1] // 2)
    simage[sensor_center] += signal_output # adding signal to center pixel of␣
 ↪sensor array

    return simage

vmin = 0

simage1 = generate_simage(r=2)
simage2 = generate_simage(r=6)
simage3 = generate_simage(r=10)

vmax = np.max([np.max(simage1), np.max(simage2), np.max(simage3)])

plt.figure(figsize=(11,7))

plt.subplot(1,3,1)
plt.imshow(simage1, cmap = 'gray', vmin=vmin, vmax=vmax)
plt.title('r = 2')
plt.axis('off')

plt.subplot(1,3,2)
plt.imshow(simage2, cmap = 'gray', vmin=vmin, vmax=vmax)
plt.title('r = 6')
plt.axis('off')

plt.subplot(1,3,3)
plt.imshow(simage3, cmap = 'gray', vmin=vmin, vmax=vmax)
plt.title('r = 10')
plt.axis('off');
```

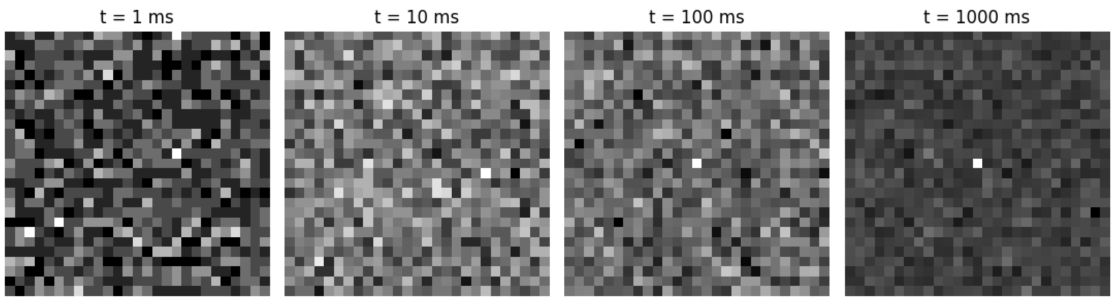
```
9
```

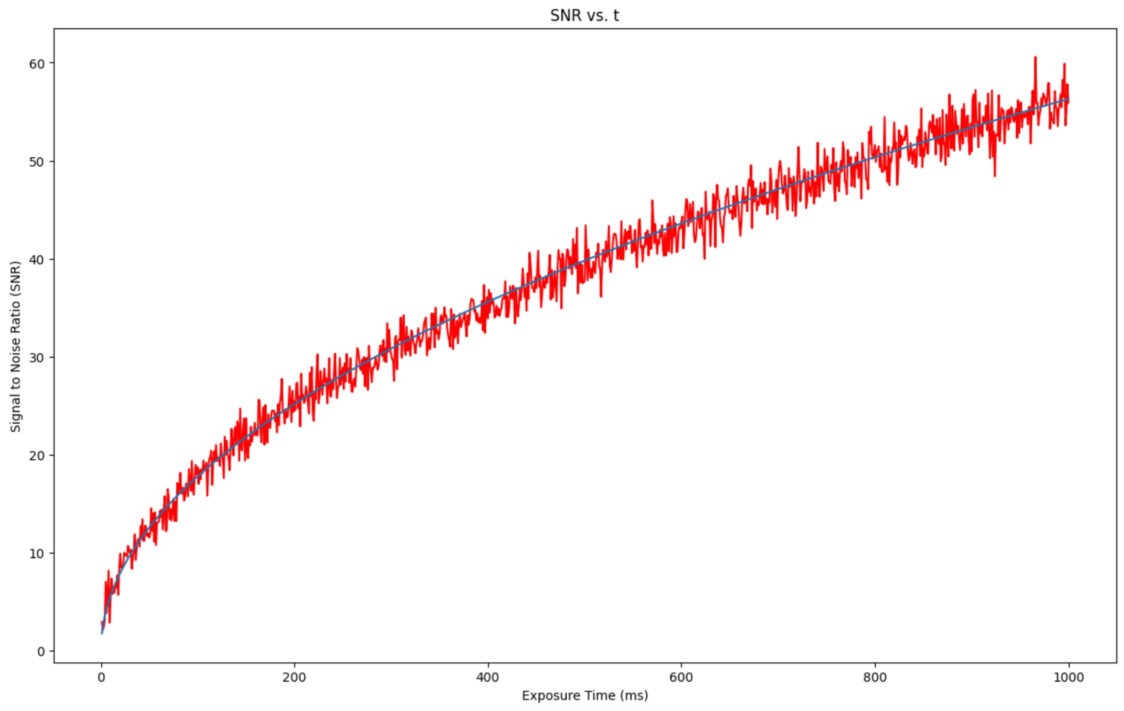


(b) Make a series of simulated images for r = 0.5 units/ms for a series of exposure times t = 1, 2, 3, ..., 300 milliseconds. Just make one simulated image for each t; we'll deal with averaging over images later. From the actual pixel values of the simulated images, calculate the signal-to-noise ratio (SNR). You can crudely use the intensity of the center pixel as an estimate of "signal" and the standard deviation of all the other pixels as your estimate of "noise"" Make a scatter plot of SNR vs t, and comment on whether it has the expected shape. (Hint: you might want to make your image a 1D array to more easily calculate the standard deviation of "all the other pixels." If you had a 3x3 image, you'd calculate the standard deviation of pixels 0, 1, 2, 3, 5, 6, 7, 8; why?)

[246]:
```
r = 0.5
tmax = 1000
t_range = np.linspace(1,tmax,tmax)

simage_list = []
snr_list = []

for t in t_range:

    simage = generate_simage(r=r, t=t)
    signal = simage[(simage.shape[0] // 2, simage.shape[1] // 2)]
    background = np.delete(simage.flatten(), (len(simage.flatten())) // 2)
    snr = signal / np.std(background)

    simage_list.append(simage)
    snr_list.append(snr)
```

```
plt.figure(figsize=(11,7))

plt.subplot(1,4,1)
plt.imshow(simage_list[0], cmap = 'gray')
plt.title('t = 1 ms')
plt.axis('off')

plt.subplot(1,4,2)
plt.imshow(simage_list[9], cmap = 'gray')
plt.title('t = 10 ms')
plt.axis('off')

plt.subplot(1,4,3)
plt.imshow(simage_list[99], cmap = 'gray')
plt.title('t = 100 ms')
plt.axis('off')

plt.subplot(1,4,4)
plt.imshow(simage_list[999], cmap = 'gray')
plt.title('t = 1000 ms')
plt.axis('off')
plt.tight_layout()

plt.figure(figsize=(15,9))
plt.plot(t_range, snr_list, 'r')
plt.xlabel('Exposure Time (ms)')
plt.ylabel('Signal to Noise Ratio (SNR)')
plt.title('SNR vs. t')
plt.plot(t_range, 1.78*np.sqrt(t_range));
```





Since both are poisson distributions, we expect the signal to be increasing faster than the standard deviation of the noise. The signal itself is directly proportional to time, so we expect a linear increase, while the standard deviation of the noise is proportional to the square root of time, thus SNR ~ t / sqrt(t) ~ sqrt(t). To show this I've plotted a sqrt line with a best guess (no fancy fitting yet) coefficent that fits the SNR wrt time.

8) A little bit of Fourier Transforming

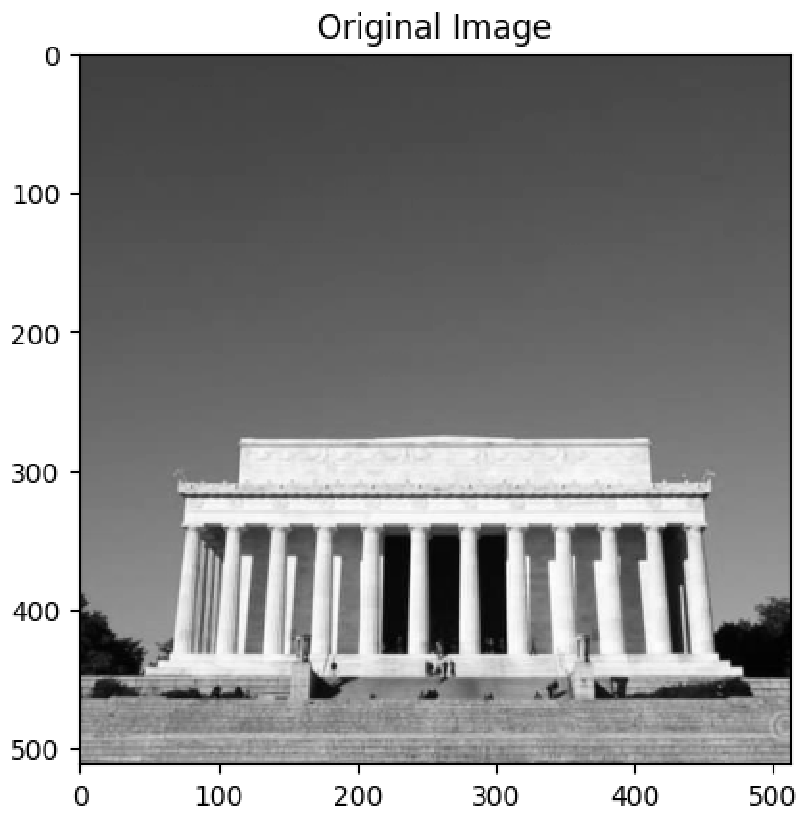
Read the code, run it, and go back and read it again. The program loads an image, performs a Fourier Transform (fft2), shifts it so that zero frequency is at the center, separates the complex numbers into amplitude and phase, displays the amplitude (which I showed in class), and does something interesting to the amplitude array, and Fourier Transforms back into an image. Why does the image look like it does given that the "mask" looks like it does? Explain. (A few sentences is fine.)

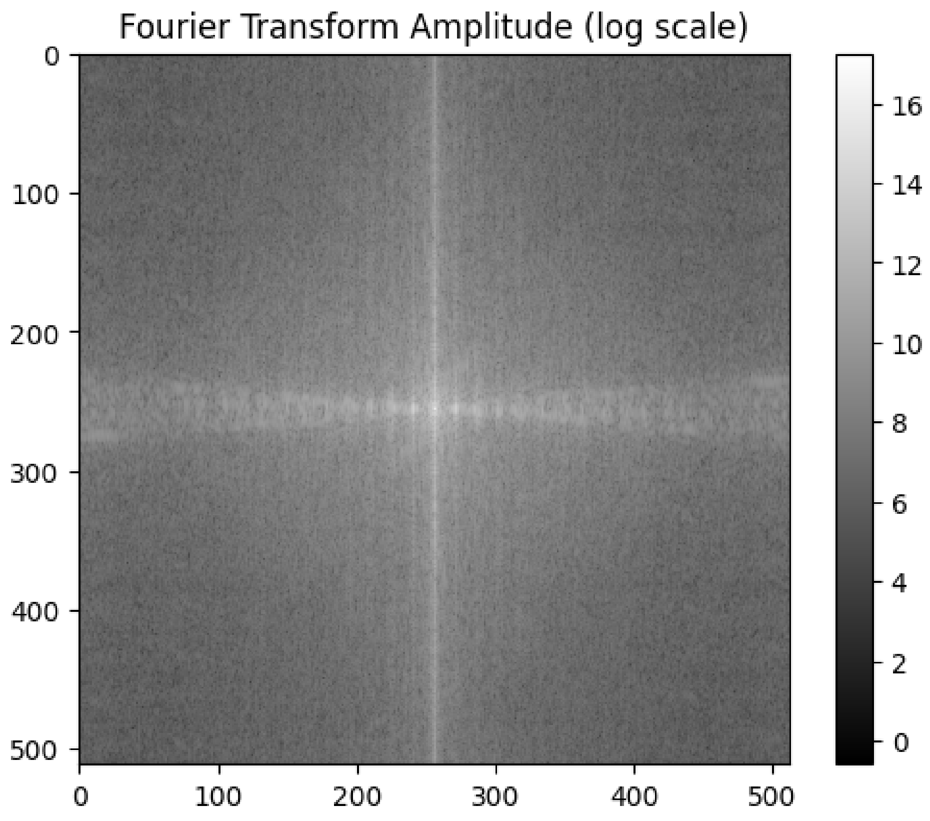
It looks like this code is essentially zeroing out several frequencies of information from the original image. The "fundamental frequency" is essentially just the period, or spacing, between the frequencies that are to be removed from the image. This removal is done by the multiplication of the zeros-mask and amplitude. When inverse transformed, the image is now altered as it's missing spatial information/features that have frequencies that are multiples of the fundamental frequency. This is apparent in the final plotted image, which now has almost a "ghost" of itself overlayed in the spots which were previously gaps between the columns/pillars. If I were to guess, this fundamental frequency lines up with the spacing between the pillars.

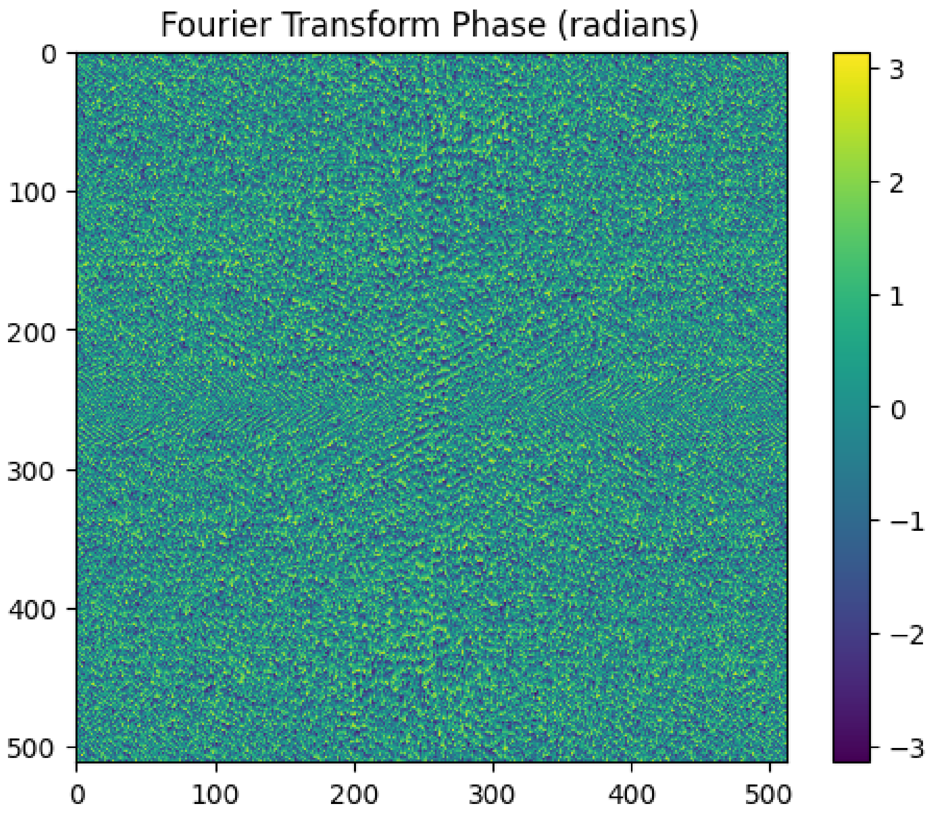
[23]:
```
import numpy as np
import matplotlib.pyplot as plt
```

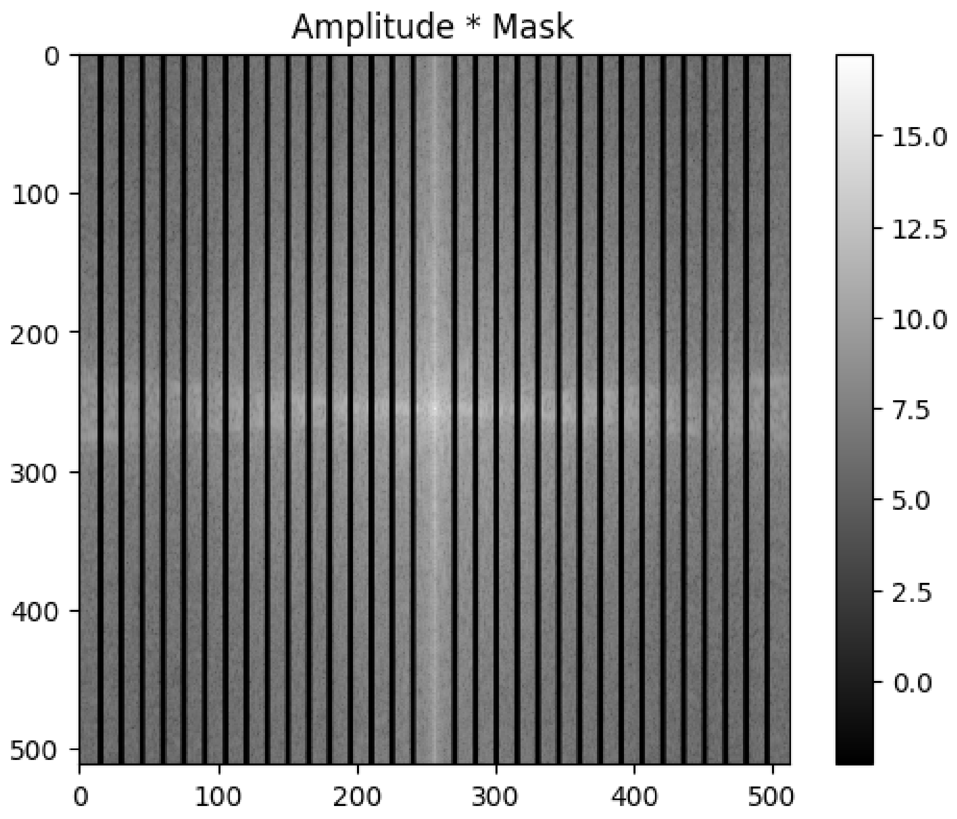
```python
import os
from skimage import io # input output sub-package


#%% Load the image

parentDir = r'/home/apd/Projects/ImageAnalysis/HW3'
fileName = r'Lincoln_Coleman_40-copyright-havecamerawilltravel-com_crop512_gray.
  ↪png'

im = io.imread(os.path.join(parentDir, fileName))

if im.ndim > 2:
    # A bit silly, since I know this is 2D
    im = np.mean(im, axis=2, dtype=im.dtype)

print('Image shape: ', im.shape)
# Image size; I'm not checking if it's square!
# The Lincoln Memorial image is 512x512
N = im.shape[0]

plt.figure()
plt.imshow(im, 'gray')
plt.title('Original Image')

#%% Fourier Transform

# Perform 2D Fourier transform
F = np.fft.fft2(im)  # Fast Fourier Transform
F_shifted = np.fft.fftshift(F)  # Shift so zero frequency is in the center

# Calculate the amplitude and phase
amplitude = np.abs(F_shifted)
phase = np.angle(F_shifted)

# Display amplitude as an image
plt.figure()
plt.title("Fourier Transform Amplitude (log scale)")
# Should maybe add an offset to avoid -Inf,
# but I've tested and there are no zeros.
plt.imshow(np.log(amplitude), cmap='gray')
plt.colorbar()
plt.show()

# Display phase as an image
plt.figure()
plt.title("Fourier Transform Phase (radians)")
plt.imshow(phase)
plt.colorbar()
plt.show()

#%% Masking

# "Fundamental frequency" for the mask
f0 = 15 # I determined this "by hand"
# Full width of the mask -- should be an even number
df = 4

# Create a mask array
mask = np.ones((N, N))
for k in range(1, N//(2*f0)):
    center_f = N/2 + k*f0
    mask[:, int(center_f - df/2):int(center_f + df/2)] = 0
    center_f = N/2 - k*f0
    mask[:, int(center_f - df/2):int(center_f + df/2)] = 0

# Create a new amplitude array that is the original multiplied by this mask
new_amplitude = amplitude * mask

# Display the new amplitude as an image
plt.figure()
plt.title("Amplitude * Mask")
plt.imshow(np.log(new_amplitude + 0.1), cmap='gray') # + 0.1 because of zeros.
plt.colorbar()
plt.show()


# Combine new amplitude with original phase
new_F_shifted = new_amplitude * np.exp(1j * phase)

# Perform the inverse Fourier transform
new_F = np.fft.ifftshift(new_F_shifted)
new_im = np.fft.ifft2(new_F)
new_im = np.abs(new_im)

# Display the resulting image
plt.figure()
plt.title("Image based on Inverse FT")
plt.imshow(new_im, cmap='gray')
plt.colorbar()
plt.show()
```

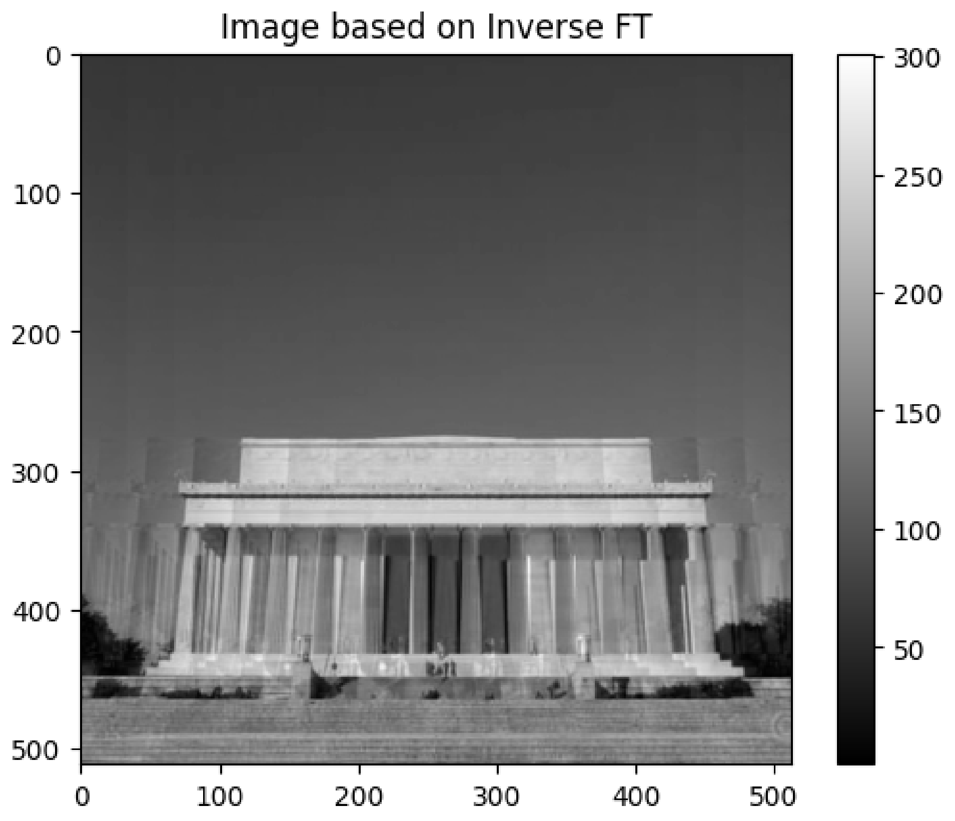
```
Image shape:  (512, 512)
```


Original Image
0
100
200
300
400
500
0
100
200
300
400
500


Fourier Transform Amplitude (log scale)
0
100
200
300
400
500
0
100
200
300
400
500
16
14
12
10
8
6
4
2
0

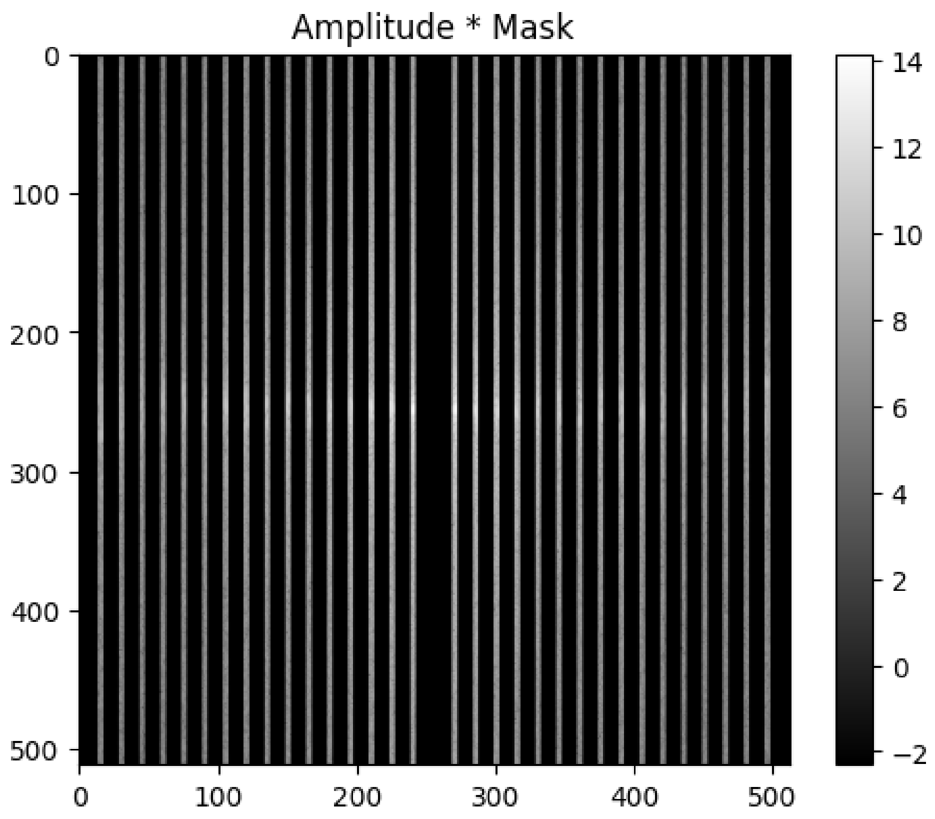

Fourier Transform Phase (radians)
0
100
200
300
400
500
0
100
200
300
400
500
3
2
1
0
−1
−2
−3

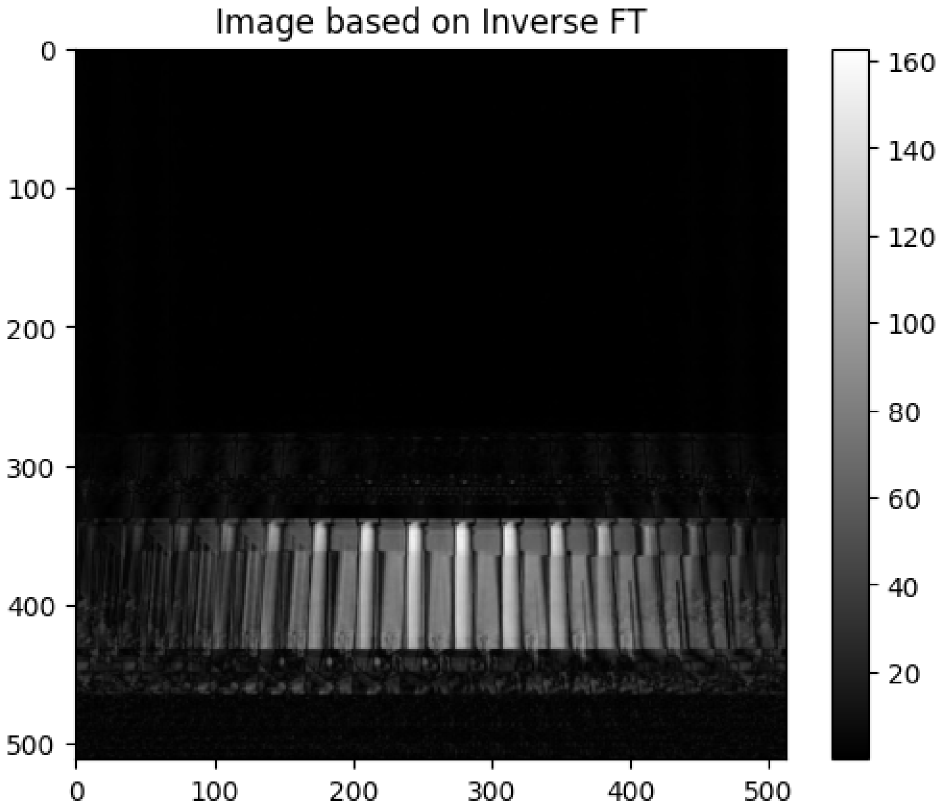

Amplitude * Mask
0
100
200
300
400
500
0
100
200
300
400
500
15.0
12.5
10.0
7.5
5.0
2.5
0.0



[24]:
```
new_amplitude = amplitude * (1-mask)

# Display the new amplitude as an image
plt.figure()
plt.title("Amplitude * Mask")
plt.imshow(np.log(new_amplitude + 0.1), cmap='gray') # + 0.1 because of zeros.
plt.colorbar()
plt.show()



# Combine new amplitude with original phase
new_F_shifted = new_amplitude * np.exp(1j * phase)

# Perform the inverse Fourier transform
new_F = np.fft.ifftshift(new_F_shifted)
new_im = np.fft.ifft2(new_F)
new_im = np.abs(new_im)

# Display the resulting image
plt.figure()
plt.title("Image based on Inverse FT")
```

```python
plt.imshow(new_im, cmap='gray')
plt.colorbar()
plt.show()
```





We can just invert the mask to instead keep, rather than remove, the parts with fundamental frequency before the inverse fft by multiplying by (1 - mask) instead.

9) I'd say this assignment took me 4-5 hours, with the longest part being the bandpass filter. I tried toying with the different gaussians and kernel sizes for a while before landing at where I got, and it still isn't an amazing bandpass filter.