prob 1.

Matrix Inversion for Kernel (filter)

This step include flipping of the kernel, rows followed by a flip along its column.

$$\begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \end{bmatrix} \quad h$$

$$\Downarrow$$

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 1 & 0 \end{bmatrix} \quad h' \qquad \Rightarrow \qquad \begin{bmatrix} -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix} \quad h''$$

① h'': $-1, 0, 1$ / $0, 1, 2$ — f: $1, 2$ / $3, 4$ / $2, 1$

~~y=E=0~~

$y[0,0] = 2$

② h'': $-1, 0, 1$ / $0, 1, 2$ — f: $1, 2$ / $3, 4$ / $2, 1$

$y[0,1] = 5$

③ h'': $-1, 0, 1$ / $0, 1, 2$ — f: $1, 2$ / $3, 4$ / $2, 1$

$y[0,2] = 2$

④ h'': $-1, 0, 1$ / $0, 1, 2$ — f: $1, 2$ / $3, 4$ / $2, 1$

$y[0,3] = 0.$

PAGE
DATE

⑤ $y[1,0] = 7$

⑥ $y[1,1] = 13$

⑦ $y[1,2] = 3$

⑧ $y[1,3] = -2$

⑨ $y[2,0] = 7$

⑩ $y[2,1] = 8$

⑪ $y[2,2] = -2$

⑫ $y[2,3] = -4$

⑬ $y[3,0] = 2$

⑭ $y[3,1] = 1$

⑮ $y[3,2] = -2$

⑯ $y[3,3] = -1$

PAGE
DATE

output = [Input_length + kernel_length − 1]

[3 × 2] [2×3] = [4 × 4]

input kernel output.

$$\therefore \text{output} = \begin{bmatrix} 2 & 5 & 2 & 0 \\ 7 & 13 & 3 & -2 \\ 7 & 8 & -2 & -4 \\ 2 & 1 & -2 & -1 \end{bmatrix}$$

prob 2.

$$h = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \qquad I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$\vec{I} = [1, 2, 3, 4, 5, 6, 7, 8, 9]^T$

Find the H (matrix) which is 25×9.

$I * h = HI^T$

First, output size $= (I_m + h_m - 1) \times (I_n + h_n - 1)$

$= 5 \times 5$. (output size)

Then, we should pad zeros in h matrix (kernel).

$$h = \begin{bmatrix} 1 & 0 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

We can use the toeplitz matrix.

$$T_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \qquad T_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

$$T_3 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \qquad T_4 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad T_5 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Then, we can get and compute doubly toeplitz matrix from those toeplitz matrices.

(~~Column~~ no of column = no of rows of image (input))

$$\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
-1 & 0 & 1 & -1 & 0 & 1 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
-1 & 0 & 1 & -1 & 0 & 1 & -1 & 0 & 1 \\
0 & -1 & 0 & 0 & -1 & 0 & 0 & -1 & 0 \\
0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & -1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & -1 & 0 & 1 & -1 & 0 & 1 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1
\end{bmatrix}$$

25×9

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{bmatrix}$$

9×1

Then, we multiply these two matrices to get the 25x1 output vector.

output vector = [1 2 2 −2 −3 5 7 4 −7 −9
12 15 6 −15 −18 11 13 −4 −13 −15
7 8 2 −8 −9]

∴
$$\begin{bmatrix} 1 & 2 & 2 & -2 & -3 \\ 5 & 7 & 4 & -7 & -9 \\ 12 & 15 & 6 & -15 & -18 \\ 11 & 13 & 4 & -13 & -15 \\ 7 & 8 & 2 & -8 & -9 \end{bmatrix}$$

Problem 2 (b)

Code:

```
import numpy as np
from scipy import signal
import cv2
import time

'''
###################################
####          Steps            ####
###################################
1. Zero-padding
2. Compute toeplitz matrix
3. Compute doubly toeplitz matrix
'''

def zero_pad(kernel, output):
    #input: filter and output
    #shape of kernel and output
    height, width = kernel.shape[:2]
    height_output, width_output = output.shape[:2]
    # zero padding at bottom
    bot = ((height + height_output) - 1) - height_output
    # zero padding at right
    right = ((width + width_output) - 1) - width_output
    # zero padding by kernel
    padded = np.pad(kernel, ((0, bot), (0, right)), 'constant', constant_values=0)
    return padded


def toeplitz_matrices(kernel, row):
    # input: kernel , row
    kernel_height, kernel_width = kernel.shape[:2]

    toeplitz_matrix = [] # empty list to append matrices
    for i in range(kernel_height):
        output = np.zeros((kernel.shape[0], row))
        for k in range(kernel.shape[0]):
            for j in range(row):
                if k >= j:
                    output[k][j] = kernel[i][k - j]
        # create toeplitz matrix and add the list
        toeplitz_matrix.append(output)
```

```python
def doubly(toeplitz_matrix):
    # This function computes the doubly-blocked-toeplitz-matrix
    # input is from the above function toeplitz_matrices; input: toeplitz matrix
    height = toeplitz_matrix.shape[0] * toeplitz_matrix.shape[1]
    width = toeplitz_matrix.shape[2] * toeplitz_matrix.shape[2]
    doubly_block = [height, width]
    #initialize doubly block matrix
    output = np.zeros(doubly_block)
    for i in range(toeplitz_matrix.shape[1]):
        for j in range(toeplitz_matrix.shape[2]):
            if i >= j:
                # create the doubly_blocked_toeplitz_matrix with toeplitz matrices
                output[i * toeplitz_matrix.shape[1]:(i + 1) * toeplitz_matrix.shape[1],
                j * toeplitz_matrix.shape[2]:(j + 1) * toeplitz_matrix.shape[2]] = toeplitz_matrix[i - j]
    return output
def convolution_matrix(I, kernel):
    # shape of kernel and Input
    height, width = I.shape[:2]
    kernel_height, kernel_width = kernel.shape[:2]
    #compute output size
    output_height, output_width = height + kernel_height -1, width + kernel_width - 1
    # compute toeplitz matrices
    toeplitz_mat = toeplitz_matrices(zero_pad(kernel, I), width)
    # compute doubly blocked toeplitz matrix
    doubly_toeplitz = doubly(toeplitz_mat)
    #vectorize Input I
    vectorize_I = I.reshape(height * width)
    # multiply doubly toeplitz matrix and vectorized I and the output should reshape to be 5*5
    final_result = np.matmul(doubly_toeplitz, vectorize_I).reshape(output_height, output_width)
    return final_result
```

```python
############ problem2 (b) #################
def conv2dmatrix(I, H):
    # timer
    start = time.time()
    conv_matmul = convolution_matrix(I, H) #Convolution as matrix multiplication
    # get run time
    T = round(time.time() - start, 3)
    # compute convolution from scipy library
    conv_scipy = signal.convolve2d(I, H, 'full')
    # calculate mse between two convolution results
    mse = np.mean((conv_matmul - conv_scipy) ** 2)
    return conv_matmul, T, mse

if __name__ == '__main__':
    I = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])
    H = np.array([[1, 0, -1],
                  [1, 0, -1],
                  [1, 0, -1]])

    result_from_conv2d, T, mse = conv2dmatrix(I, H)
    print('convolution\n', result_from_conv2d)
    print('run time\n', T)
    print('mse score\n', mse.item())
```

Output:

```
convolution
 [[  1.   2.   2.  -2.  -3.]
 [  5.   7.   4.  -7.  -9.]
 [ 12.  15.   6. -15. -18.]
 [ 11.  13.   4. -13. -15.]
 [  7.   8.   2.  -8.  -9.]]
run time
 0.0
mse score
 0.0
```

Problem 2 (c)

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
import cv2
import math
import time
from convolution_as_matrix import toeplitz_matrices, doubly


def elephant_convolution2D(I, kernel):
    # Generalize the code and
    # the this function can still compute the convolution as a matrix multiplication
    s = time.time()
    #Before dive into the algorithm,  H is too large so cut up the image 100x 100
    Cut_the_img_size = np.zeros((100, 100))
    # find the shape of Input and kernel
    height, width = I.shape[:2]
    # define the kernel height and width
    kernel_height, kernel_width = kernel.shape[:2]
    # define the output size( Im + Ik -1 *Im + Ik -1)
    output_height, output_width = height + kernel_height - 1, width + kernel_width - 1
    # declare the empty 100x100 matrix as cut image size
    output = np.zeros((output_height, output_width))
    # final image size
    final_output_cut = Cut_the_img_size.shape[0] + kernel_height - 1, Cut_the_img_size.shape[1] + kernel_width - 1
    # for the zero padding
    a = final_output_cut[0] - kernel.shape[0]
    # for the zero padding
    b = final_output_cut[1] - kernel.shape[1]
```

```
    # zero padding kernel
    k_zero_pad = np.pad(kernel, ((0, a), (0, b)), 'constant', constant_values=0)
    # compute the toeplitz_matrices
    toeplitz_matrix = toeplitz_matrices(k_zero_pad, Cut_the_img_size.shape[1])
    # compute doubly-blocked-toeplitz-matrix
    doubly_blocked = doubly(toeplitz_matrix)
    cut_height = output_height // Cut_the_img_size.shape[0]
    cut_width = output_width // Cut_the_img_size.shape[1]
    for i in range(cut_height):
        for j in range(cut_width):

            final_cutting = I[i * 100:(i + 1) * 100, j * 100: (j + 1) * 100]
            # vectorize the cut input
            vectorized_cut = final_cutting.reshape(Cut_the_img_size.shape[0] * Cut_the_img_size.shape[1])
            # multiply doubly-blocked-toeplitz matrix and vectorized_cut_img
            cut_t_doubly = np.matmul(doubly_blocked, vectorized_cut)
            # reshape result vector to matrix
            cut_convolution2D = cut_t_doubly.reshape((final_output_cut[0], final_output_cut[1]))
            #output = np.zeros((100, 100))
            output[i * 100:(i + 1) * 100, j * 100: (j + 1) * 100] = cut_convolution2D[1:-1, 1:-1]

    conv_net = signal.convolve2d(I, kernel, 'full')
    mse_score = np.mean((output - conv_net) ** 2).item()
    T = time.time() - s
    return output, T, mse_score
```

```
if __name__ == '__main__':
    # read elephant image
    image = cv2.imread('data/elephant.jpeg', 0)
    H = np.array([[1, 0, -1],
                  [1, 0, -1],
                  [1, 0, -1]])

    # elephant convolution as matrix multiplication
    conv_net, T, mse = elephant_convolution2D(image, H)
    print('convolution output\n', conv_net)
    print('run time \n', T)
    print('mse score \n', mse)

    # display convolution elephant image
    plt.imshow(conv_net, cmap='gray')
    plt.show()

    # save the image
    cv2.imwrite('data/elephant_conv.png', conv_net)
```

Result:

```
convolution output
 [[ 180.    4.    2. ...  -83.    0.    0.]
 [ 274.    7.    3. ... -127.    0.    0.]
 [ 281.    8.    4. ... -134.    0.    0.]
 ...
 [ 275.  -16.  -15. ... -325.    0.    0.]
 [   0.    0.    0. ...    0.    0.    0.]
 [   0.    0.    0. ...    0.    0.    0.]]
run time
 4.70986795425415
mse score
 3324.403406901244
```

Output (edge filtered image):

Problem 3 (Fourier Domain Fun)

I coded utilize functions (Fourier transform, inverse transform and swap phases) for solving the following problems.

```python
import cv2
import numpy as np
import math
import matplotlib.pyplot as plt

def fourier_trans(image):

    dis_fourier = cv2.dft(image, flags=cv2.DFT_COMPLEX_OUTPUT)
    dft_shift = np.fft.fftshift(dis_fourier)

    return dft_shift

def inv_trans(fshift):
    # Inverse transformation
    f_ishift = np.fft.ifftshift(fshift)
    img_back = cv2.idft(f_ishift)
    img_back = cv2.magnitude(img_back[:, :, 0], img_back[:, :, 1])
    img_back = cv2.normalize(img_back, img_back, 0, 1.0, cv2.NORM_MINMAX)
    return img_back

def swap_phases(f_shift, phase):
    return np.multiply(np.abs(f_shift), np.exp(1j * phase))

def inverse_transform(fshift):
    # inverse transform
    f_ishift = np.fft.ifftshift(fshift)
    img_back = np.fft.ifft2(f_ishift)
    img_back = np.abs(img_back)
    return img_back
```

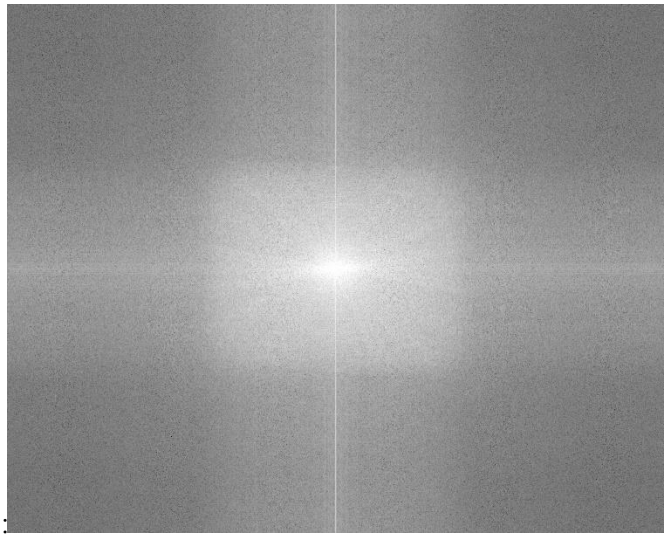
(a) Read in the elephant picture. Plot the Fourier transform (magnitude and phase) of the picture.

Code:

```python
import cv2
import numpy as np



def main():
    # read image
    img_el = cv2.imread('data/elephant.jpeg', 0)

    # get magnitude and phase using fourier transform
    f = np.fft.fft2(img_el)
    fshift = np.fft.fftshift(f)
    magnitude = 20 * np.log(np.abs(fshift))
    phase = np.angle(fshift)

    # plot show magnitude and phase
    cv2.imwrite('data/elephant_magnitude.png', magnitude)
    cv2.imwrite('data/elephant_phase.png', phase)


if __name__ == '__main__':
    main()
```

Output:

Magnitdue:



:

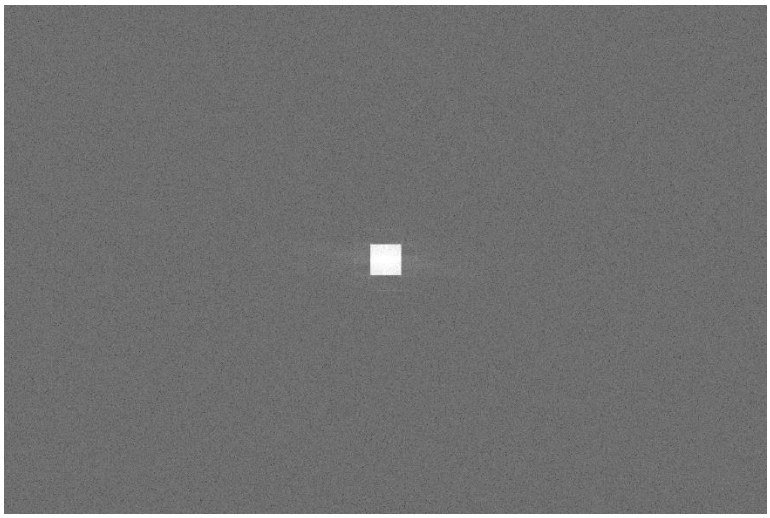
Phase:

(b) low-pass filter

Code:

```python
import cv2
import numpy as np
import math
import matplotlib.pyplot as plt
from inv_transform import fourier_trans, inv_trans

def low_pass_filter():
    # this is the low_pass filter
    # load image
    image = cv2.imread('data/elephant.jpeg', 0)
    #convert to float 32
    img_float32 = np.float32(image)

    rows, cols = image.shape  # 1500 * 1000 image size
    crow, ccol = rows // 2, cols // 2  # 750 x 500 image size
    fft_img = fourier_trans(img_float32)
    # create the mask
    mask = np.zeros((rows, cols, 2), np.uint8)
    mask[crow - 30: crow + 30, ccol - 30: ccol + 30] = 1
    # magnitude after filtering
    fshift = fft_img * mask
    # inverse transform
    final = inv_trans(fshift)
    final1 = np.uint8(final * 255)
    # check the magnitude
    f = np.fft.fft2(final1)
    fshift = np.fft.fftshift(f)
    magnitude = 20 * np.log(np.abs(fshift))
    # save the image output
    cv2.imwrite("data/elephant_low.png", np.uint8(final * 255))
    cv2.imwrite("data/elephant_low_mag.png", magnitude)
if __name__ == '__main__':
    low_pass_filter()
```

Plot the Fourier magnitudes of the image after low pass filtering.

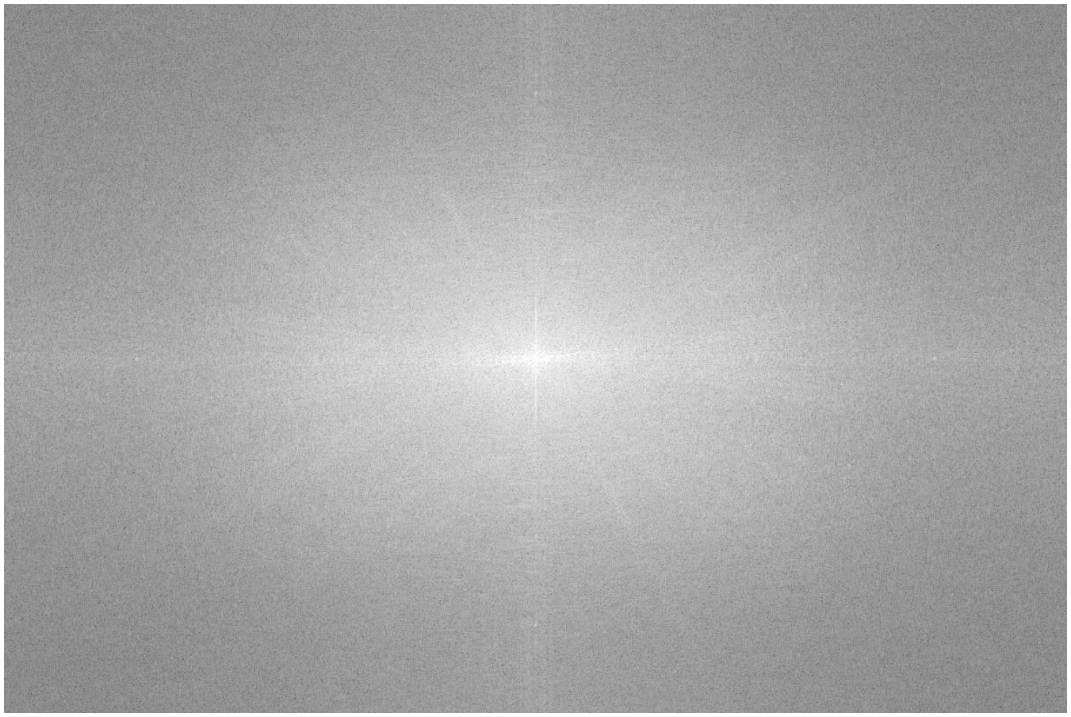


(b) High-pass-filter

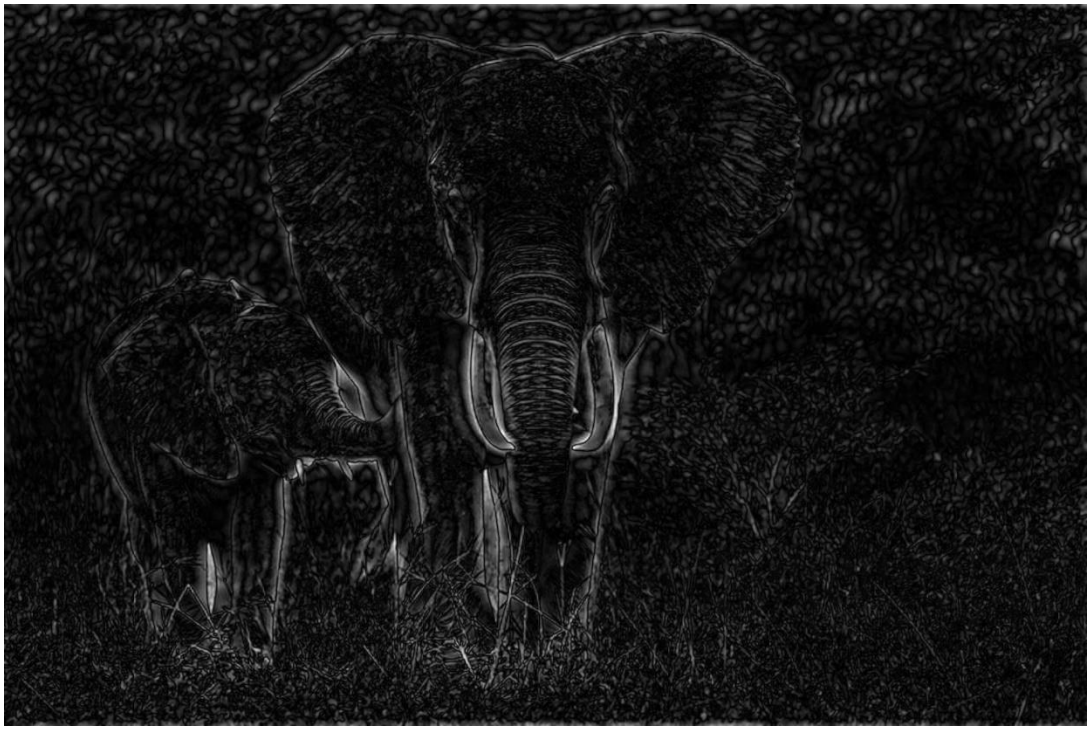
Code:

```python
import cv2
import numpy as np
import math
import matplotlib.pyplot as plt
from inv_transform import fourier_trans, inv_trans

def high_pass_filter():
    #image = cv.imread ( "D:/images/test1.png", cv.IMREAD_GRAYSCALE)
    image = cv2.imread('data/elephant.jpeg', 0)
    img_float32 = np.float32(image)

    rows, cols = image.shape  # 1500 * 1000 image size
    crow, ccol = rows // 2, cols // 2  # 750 x 500 image size
    fft_img = fourier_trans(img_float32)

    # Create a high-pass filter
    mask = np.ones((rows, cols, 2), np.uint8)
    mask[crow-30: crow + 30, ccol-30: ccol + 30] = 0

    # Filter
    fshift = fft_img * mask
    final = inv_trans(fshift)
    final1 = np.uint8(final * 255)
    #check the magnitude
    f = np.fft.fft2(final1)
    fshift = np.fft.fftshift(f)
    magnitude = 20 * np.log(np.abs(fshift))

    cv2.imwrite("data/elephant_high.png", np.uint8(final * 255))
    cv2.imwrite("data/elephant_high_mag.png", magnitude)
if __name__ == '__main__':
    high_pass_filter()
```

Plot the Fourier magnitudes of the image after high pass filtering.



The resulting filtered images:

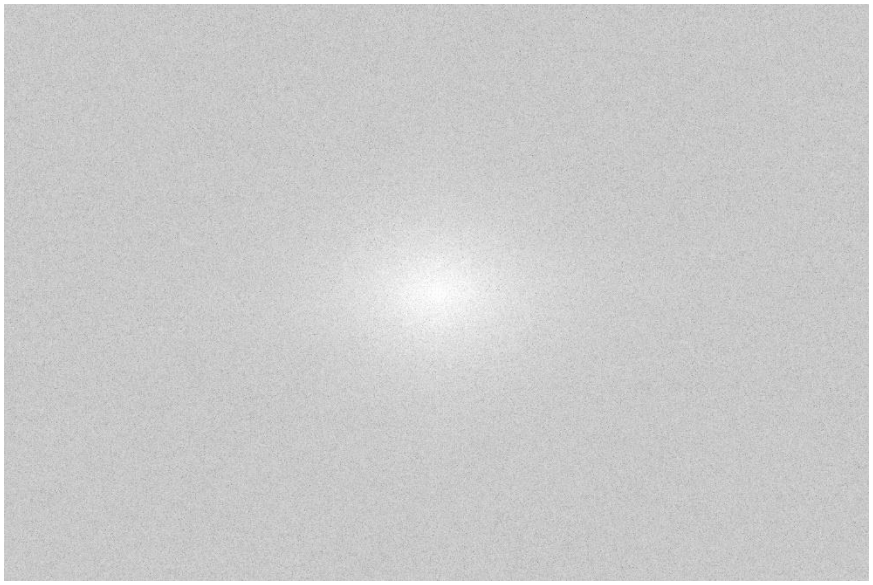


(b) Bandpass filter

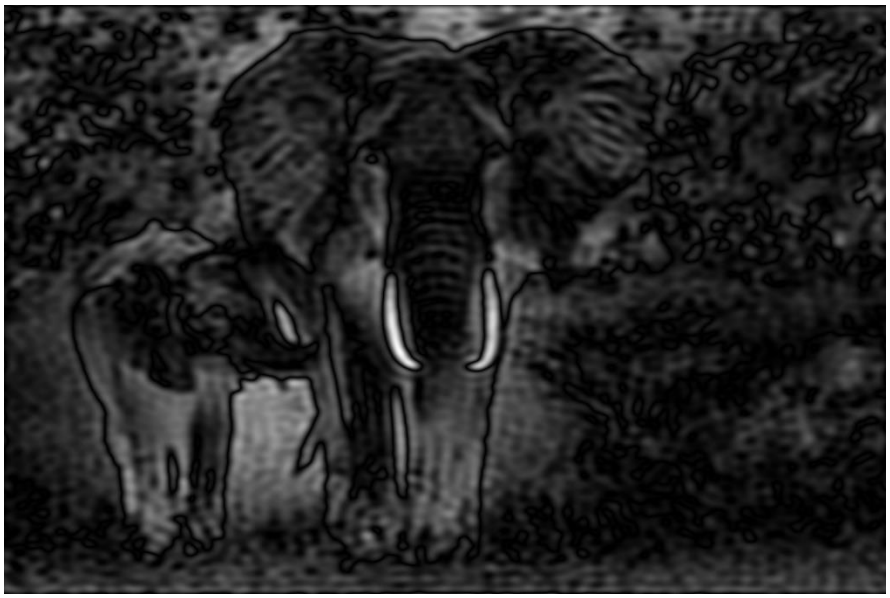
Code:

```
import cv2
import numpy as np
import math
import matplotlib.pyplot as plt

def bandpass_filter():
    #this function implement the bandpass filter
    image = cv2.imread('data/elephant.jpeg', 0)
    #discrete fourier transform
    dis_fourier = cv2.dft(np.float32(image), flags=cv2.DFT_COMPLEX_OUTPUT)
    fshift = np.fft.fftshift(dis_fourier)
    # set the paramters
    rows, cols = image.shape
    crow, ccol = rows // 2, cols // 2
    bandwidth = 56
    radius = 30
    # create the mask
    mask = np.ones((rows, cols, 2), np.uint8)
    for i in range(0, rows):
        for j in range(0, cols):
            # distance from i,j to the center point
            distance = math.sqrt(pow(i - crow, 2) + pow(j-ccol, 2))
            if radius -bandwidth/2 < distance < radius + bandwidth /2:
                mask[i,j,0]= mask[i,j,1] = 1
            else:
                mask[i,j,0] = mask[i,j,1] = 0
    # Filter
    new_f = fshift * mask
    # apply the inverse shift
    ishift = np.fft.ifftshift(new_f)
    image_back = cv2.idft(ishift) #Calculates the inverse Discrete Fourier Transform of a 1D or 2D array.
    image_back = cv2.magnitude(image_back[:, :, 0], image_back[:, :, 1])
    final1 = np.uint8 (image_back * 255) #check the magnitude after filtering
    f = np.fft.fft2(final1)
    fshift = np.fft.fftshift(f)
    magnitude = 20 * np.log(np.abs(fshift))
    cv2.normalize(image_back, image_back, 0, 1, cv2.NORM_MINMAX)
    cv2.imwrite( "data/elephant_band.png", np.uint8 (image_back * 255))
    cv2.imwrite("data/elephant_bandpass_mag.png", magnitude)
if __name__ == '__main__':
    bandpass_filter()
```

Plot the Fourier magnitudes of the image after Bandpass filtering.



The resulting filtered images:



Problem 3 (c) – Phase Swapping

Select two images (of my choice)

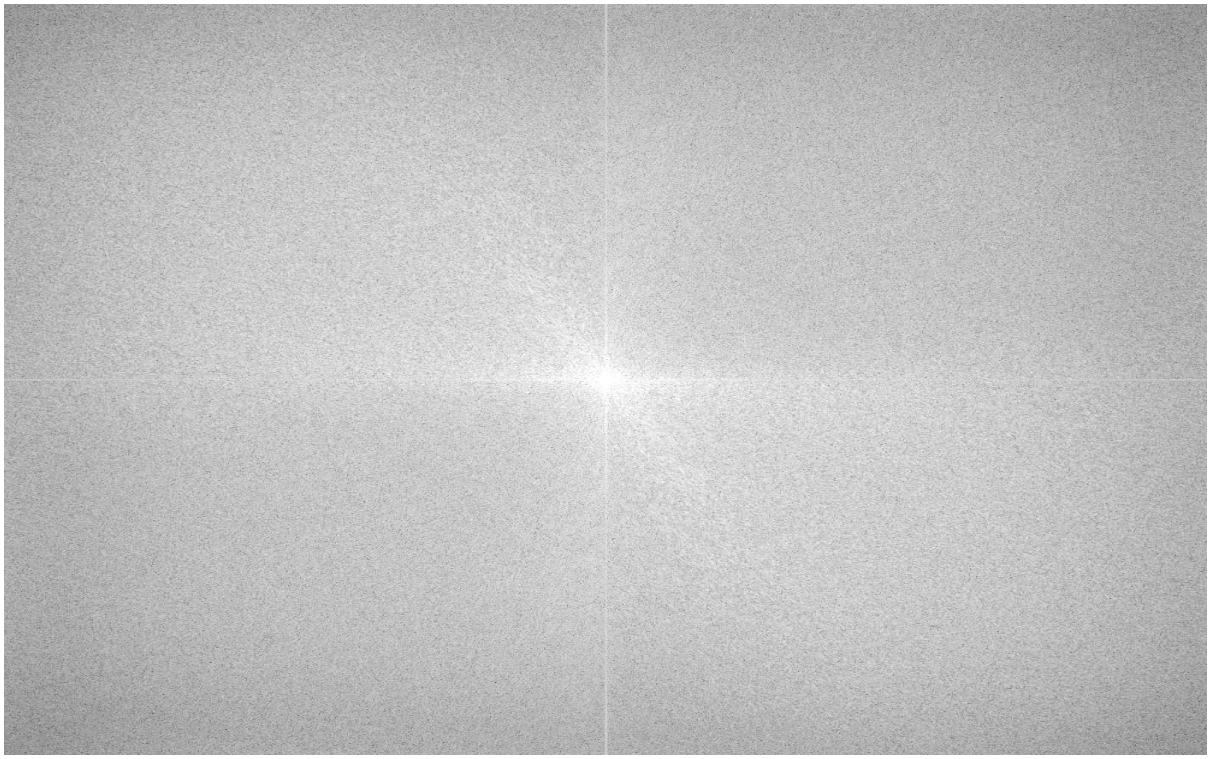
Image1

<From Google: 1000x1000 Grand Canyon >

Image2

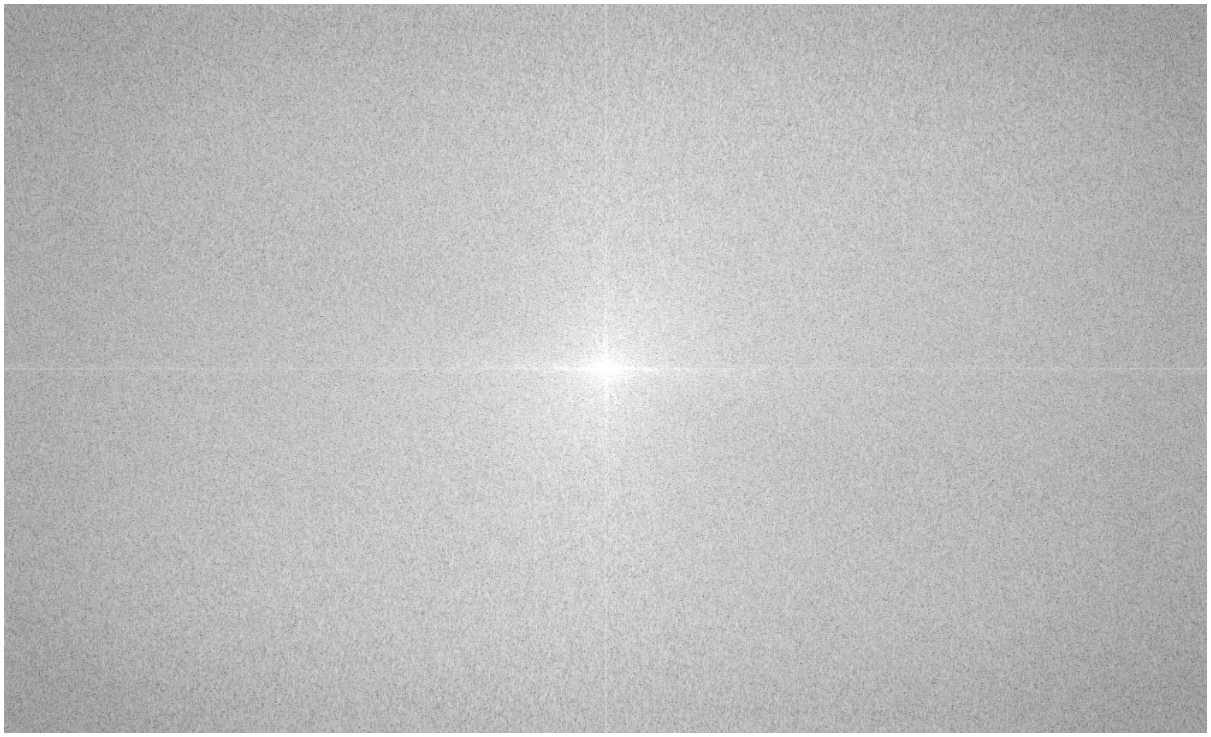
<From Google: 1000x1000 Iceland>

Magnitude of Image1(Grand Canyon):



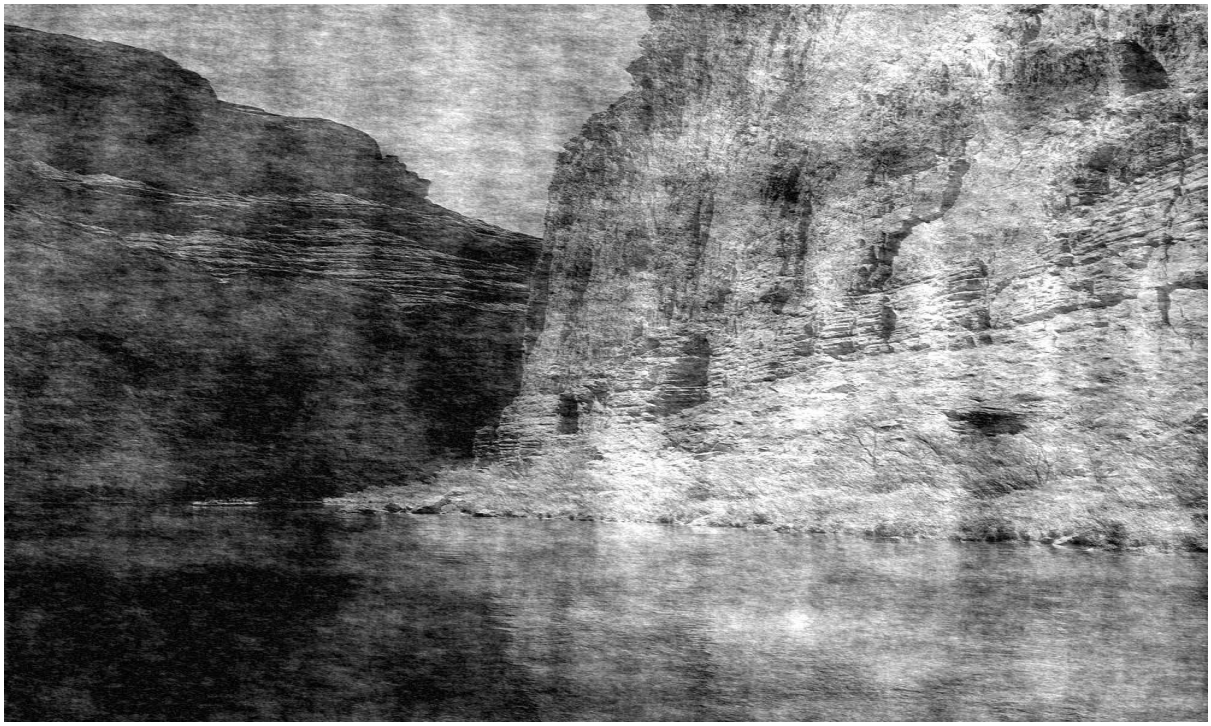
Phase of Image1(Grand Canyon):

Magnitude of Image2(Iceland):

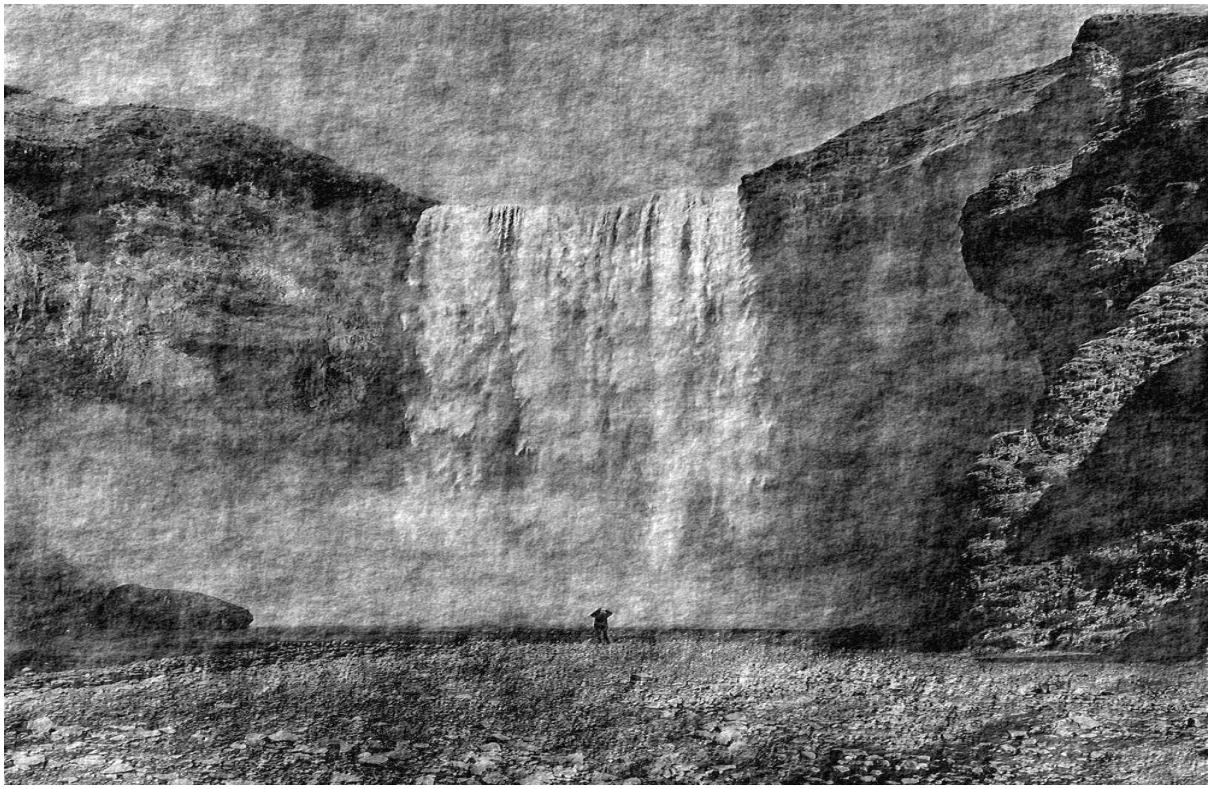


Phase of Image2(Iceland):

Result from swapped their phase images, perform the inverse Fourier transform and reconstruct the images.



< Iceland magnitude and grand phase >



< grand magnitude and Iceland phase >

How this looks perceptually?

Perceptually, it is recognized that it contains more import information of images object based on phase. For example, If the Iceland phase and Grand Canyon magnitude, it seems the result shows more information corresponding to the Iceland image.

The results confirm that the **phase contains more important information** than the magnitude of the image. So, the point is that the phase is more important part in the structure of objects in images.

Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from inv_transform import inv_trans, fourier_trans, inverse_transform
def swap_phases(f_shift, phase):
    return np.multiply(np.abs(f_shift), np.exp(1j * phase))
def main():
    # load images
    img1 = cv2.imread('data/grand.jpg', 0) #grand-canyon image
    img2 = cv2.imread('data/iceland.jpg', 0)#Iceland image
    # get magnitude and phase for image 1
    fourier = np.fft.fft2(img1)
    fshift1 = np.fft.fftshift(fourier)
    mag1 = 20 * np.log(np.abs(fshift1))
    phase1 = np.angle(fshift1)
    # get magnitude and phase for image 2
    fourier = np.fft.fft2(img2)
    fshift2 = np.fft.fftshift(fourier)
    mag2 = 20 * np.log(np.abs(fshift2))
    phase2 = np.angle(fshift2)
    # show magnitude and phase for both images
    cv2.imwrite('data/grand_mag.png', mag1)
    cv2.imwrite('data/iceland_mag.png', mag2)
    cv2.imwrite('data/grand_phase.png', phase1)
    cv2.imwrite('data/iceland_phase_.png', phase2)
    # perform phase swap and inverse transform to get image
    mag1_pha2 = inverse_transform(swap_phases(fshift1, phase2))
    plt.imshow(mag1_pha2, cmap='gray')
    plt.title('grand mag and palace phase')
    plt.show()
    plt.axis('off')
    cv2.imwrite('data/grand mag and iceland phase.png', mag1_pha2) # save swapped phase image
    # perform phase swap and inverse transform to get image
    mag2_pha1 = inverse_transform(swap_phases(fshift2, phase1))
    plt.imshow(mag2_pha1, cmap='gray')
    plt.title('data/grand phase and iceland mag.png')
    plt.show()
    plt.axis('off')
    cv2.imwrite('data/grand mag and iceland phase.png', mag2_pha1)# save swapped phase image
if __name__ == '__main__':
    main()
```

Problem3 (d) Hybrid Images.

Selected Images are same as the above, Grand Canyon & Iceland

Result (Hybrid Image):

Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from inv_transform import fourier_trans, inv_trans
def low_pass_filter(image):# This function from the problem lowpass filter
    img_float32 = np.float32(image)
    rows, cols = image.shape  # 1500 * 1000 image size
    crow, ccol = rows // 2, cols // 2  # 750 x 500 image size
    fft_img = fourier_trans(img_float32)
    mask = np.zeros((rows, cols, 2), np.uint8)
    mask[crow - 30: crow + 30, ccol - 30: ccol + 30] = 1
    fshift = fft_img * mask
    final = inv_trans(fshift)
    return final
def high_pass_filter(image):
    img_float32 = np.float32(image)
    rows, cols = image.shape  # 1500 * 1000 image size
    crow, ccol = rows // 2, cols // 2  # 750 x 500 image size
    fft_img = fourier_trans(img_float32)
    # Create a high-pass filter
    mask = np.ones((rows, cols, 2), np.uint8)
    mask [crow-30: crow + 30, ccol-30: ccol + 30] = 0
    fshift = fft_img * mask
    final_2 = inv_trans(fshift)
    return final_2
def hybrid_image(img_1_back, img_2_back):
    # This function is for generating hybrid image
    # add the two images and return
    final_1 = cv2.add(img_1_back, img_2_back)
    return final_1
def main():
    img_1 = cv2.imread('data/iceland.jpg', 0)# load images takes 2 images; iceland and grand
    img_2 = cv2.imread('data/grand.jpg', 0)
    first_img = low_pass_filter(img_1) ## perform fourier transform on both images
    second_img = high_pass_filter(img_2) ## # perform fourier transform on both images
    hybrid = hybrid_image(first_img, second_img)# generate hybrid image
    plt.imshow(hybrid, cmap='gray')# display hybrid image
    plt.show()
    cv2.imwrite('data/hybrid_image.png', (hybrid*255))# save hybrid image
if __name__ == '__main__':
    main()
```

Problem 4. (Multiresolution Blending using Gaussian/Laplacian Pyramids)

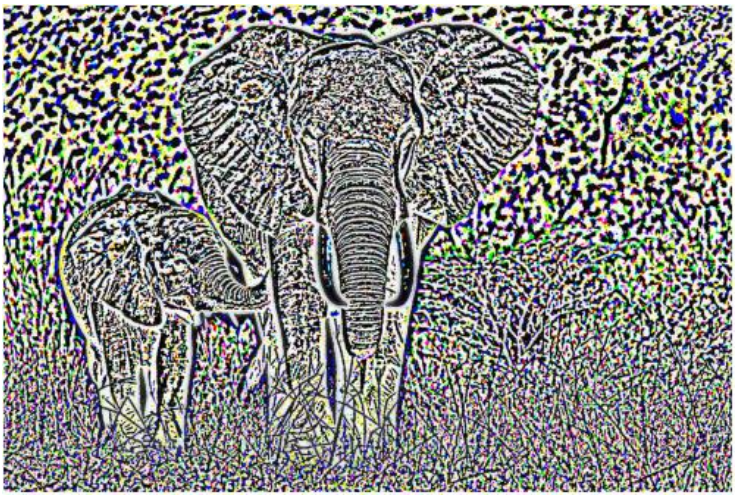
Code for (a) through (c):

```python
import cv2
import matplotlib.pyplot as plt
def pyr_up(image):
    # upsampling the image
    upsampled = cv2.resize(image, None, fx = 2, fy = 2, interpolation = cv2.INTER_CUBIC)
    return upsampled
def pyr_down(image):
    # Gaussian Blur on image
    g_blur = cv2.GaussianBlur(image, (25, 25), 5)
    # Downsampling the blur image
    g_blur_resized = cv2.resize(g_blur, None, fx = 0.5, fy = 0.5, interpolation = cv2.INTER_CUBIC)
    return g_blur_resized
def gaussian_pyramid(image):
    # generate Gaussian pyramid
    copy_image = [image]
    for i in range(5): #define the depth
        image = pyr_down(image)
        copy_image.append(image)
    return copy_image
def laplacian_pyramid(image):
    #generate Laplacian pyramid
    g_blur = cv2.GaussianBlur(image, (25, 25), 5)
    list_laplacian = [image - g_blur]
    for i in range(4): #define the depth
        # downsampling image
        down_pyramid = pyr_down(image)
        # gaussian blur for down pyramid
        blur_down_pyramid = cv2.GaussianBlur(down_pyramid, (25, 25), 5)
        list_laplacian.append(down_pyramid - blur_down_pyramid)
        # change img to downsample gaussian pyr
        image = down_pyramid
    list_laplacian.append(pyr_down(image))
    return list_laplacian
```

```python
def reconstruct(layer):
    # takes the laplacian pyramid and this function is for the reconstruction of image, depth is 5
    image = layer[5]
    for i in range(4, -1, -1):
        upsample = pyr_up(image)
        next_layer = layer[i]
        height = min(upsample.shape[0], next_layer.shape[0])
        width = min(upsample.shape[1], next_layer.shape[1])
        image = upsample[:height, :width, :] + next_layer[:height, :width, :]
    return image
def main():
    # load the image
    image = cv2.imread('data/elephant.jpeg', 1)
    # convert to RGB
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    #Gaussian pyramids
    for pyr in gaussian_pyramid(image):
        plt.axis("off")
        plt.imshow(pyr)
        plt.show()
    #Laplacian pyramids
    for pyr in laplacian_pyramid(image):
        plt.axis("off")
        plt.imshow(pyr)
        plt.show()
    # reconstruct img from laplacian pyramids
    recon = reconstruct(laplacian_pyramid(image))
    # Reconstructed image
    plt.imshow(recon)
    plt.axis("off")
    plt.show()
if __name__ == '__main__':
    main()
```

Gaussian Pyramids of an image:

Laplacian Pyramids of an image:

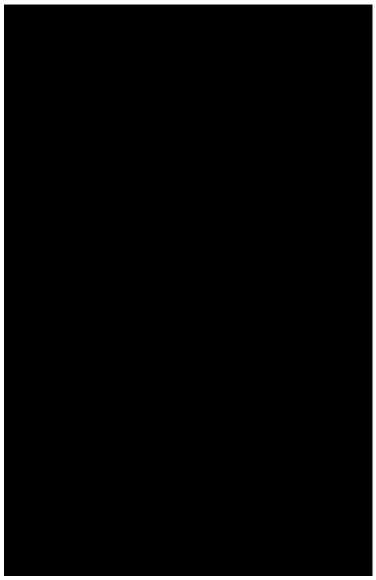


Reconstructed Image:

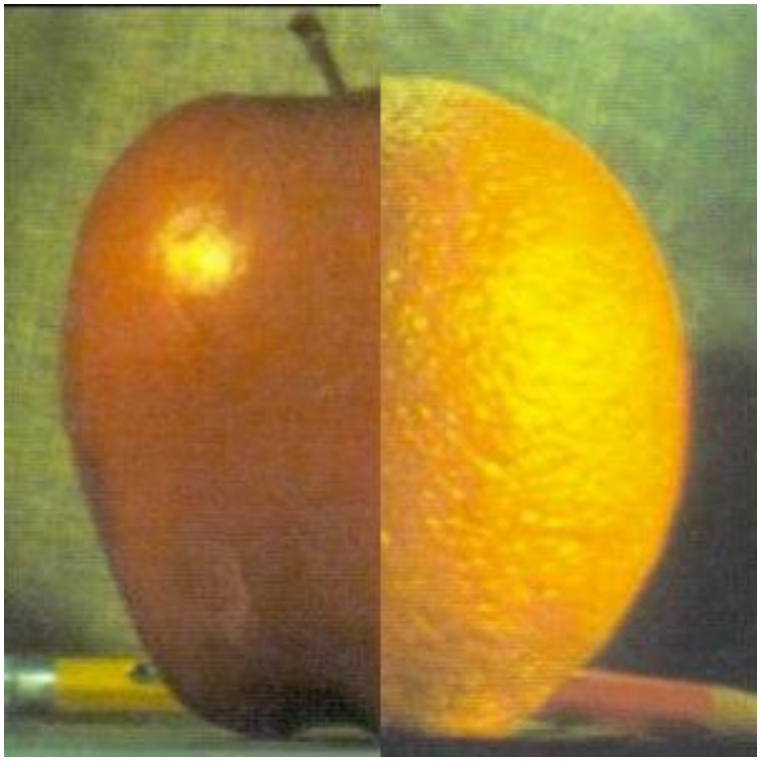
Problem 4

(d) Direct Blending:

Mask: (one half of the image as 1 and the other half at zero)



Result the image:



Code for Direct Blending:

```python
import numpy as np
import matplotlib.pyplot as plt
import cv2
def mask(image):
    #input : image
    # keeping one half of the image as 1 and the other half at zero.
    dst = image.shape
    half_zero = np.zeros((dst[0], dst[1] // 2, 3))
    half_one = np.ones((dst[0], dst[1] - dst[1] // 2, 3))
    src = np.concatenate((half_zero, half_one), axis = 1)
    return src
def direct_blending(mask, apple, orange):
    # direct blending I = (1 - M) * I1 + M * I2
    a = ((1-mask) * apple).astype(np.uint8)
    b = ((mask) * orange).astype(np.uint8)
    return a+b
def main():
    # load apple and orange image
    apple = cv2.imread('data/apple.jpeg')
    apple = cv2.cvtColor(apple, cv2.COLOR_BGR2RGB)
    orange = cv2.imread('data/orange.jpeg')
    orange = cv2.cvtColor(orange, cv2.COLOR_BGR2RGB)
    # double precision
    apple = apple.astype(np.double)
    orange = orange.astype(np.double)
    # create mask
    mask1 = mask(apple)
    # display the mask and the outputs from direct blending
    plt.imshow(mask1)
    plt.axis('off')
    plt.show()
    direct_blending_output = direct_blending(mask1, apple, orange)
    plt.imshow(direct_blending_output)
    plt.axis('off')
    plt.show()
    #save the image
    cv2.imwrite('data/direct_blending.png', cv2.cvtColor(direct_blending_output, cv2.COLOR_RGB2BGR))
if __name__ == '__main__':
    main()
```

Problem 4

(e) Alpha Blending:

Result image:

How does this look like?

It is more look like one fruit than direct blending.

The definition of alpha blending is that the process of overlaying a foreground image with transparency over a background image.

Code for Alpha Blending:

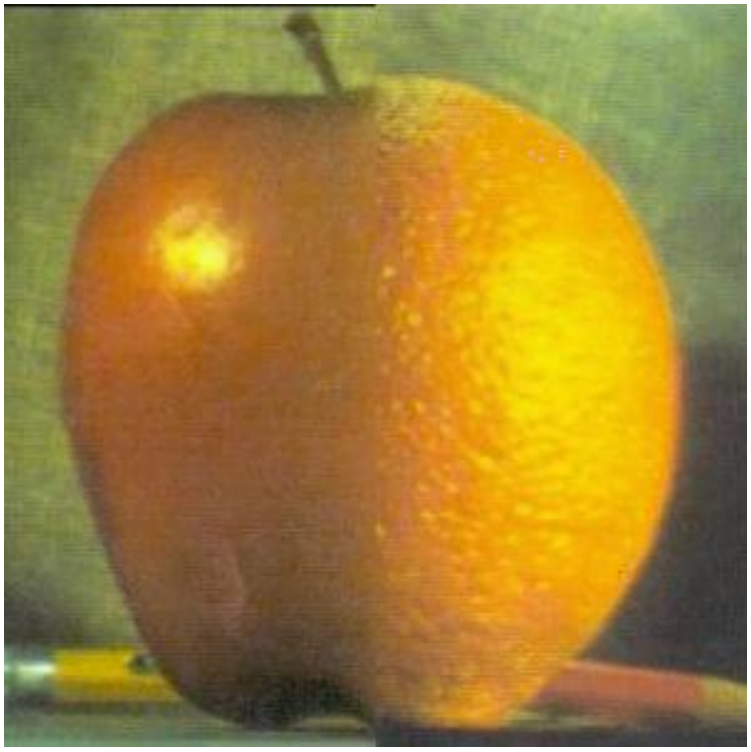
```python
import numpy as np
import matplotlib.pyplot as plt
import cv2
from direct_blending import mask
def mask(image):
    #input : image
    # keeping one half of the image as 1 and the other half at zero.
    dst = image.shape
    half_zero = np.zeros((dst[0], dst[1] // 2, 3))
    half_one = np.ones((dst[0], dst[1] - dst[1] // 2, 3))
    src = np.concatenate((half_zero, half_one), axis = 1)
    return src
def alpha_blending(mask,image1, image2):# alpha blending
    # blur the mask edge with a Gaussian / I = (1-M)*I1+M *I2
    # Gaussian blur
    mask = cv2.GaussianBlur(mask, (31, 31), 0)
    mask1_embedded = ((1 - mask) * image1).astype(np.uint8)
    mask2_embedded = (mask * image2).astype(np.uint8)
    return mask1_embedded + mask2_embedded
def main():
    # load apple and orange image
    apple = cv2.imread('data/apple.jpeg')
    apple = cv2.cvtColor(apple, cv2.COLOR_BGR2RGB)
    orange = cv2.imread('data/orange.jpeg')
    orange = cv2.cvtColor(orange, cv2.COLOR_BGR2RGB)
    # double precision
    apple = apple.astype(np.double)
    orange = orange.astype(np.double)
    # create mask
    mask1 = mask(apple)
    # display the mask and the outputs from direct blending
    plt.imshow(mask1)
    plt.axis('off')
    plt.show()
    alpha_blending_output = alpha_blending(mask1, apple, orange)
    plt.imshow(alpha_blending_output)
    plt.axis('off')
    plt.show()
    cv2.imwrite('data/alpha_blending.png', cv2.cvtColor(alpha_blending_output, cv2.COLOR_RGB2BGR)) #save the image
if __name__ == '__main__':
    main()
```

Problem 4

(f) Multiresolution Blending:

Result of orpple:

Result of appange:



Code for multiresolution blending:

```python
import numpy as np
import matplotlib.pyplot as plt
import cv2
from pyramid import pyr_up,pyr_down, laplacian_pyramid, gaussian_pyramid, reconstruct
from direct_blending import direct_blending, mask
def multiresolution_blending(image1, image2, mask):
    # this is multiresolution blending function
    # Gaussian pyramid of the Mask, depth 5
    # Laplacianpyramids of the images
    laplacian1 = laplacian_pyramid(image1)
    laplacian2 = laplacian_pyramid(image2)
    # blend pyramid
    empty_list = []
    for i in range(5, -1, -1):
        empty_list.append(direct_blending(mask[i],laplacian1[i], laplacian2[i]))
    # reconstruct the image
    final = reconstruct(empty_list[::-1])
    return final
def main():
    # load apple and orange image
    apple = cv2.imread('data/apple.jpeg')
    apple = cv2.cvtColor(apple, cv2.COLOR_BGR2RGB)
    orange = cv2.imread('data/orange.jpeg')
    orange = cv2.cvtColor(orange, cv2.COLOR_BGR2RGB)
    # double precision
    apple = apple.astype(np.double)
    orange = orange.astype(np.double)
    # create mask and input mask in gaussian_pyramid filter.
    mask1 = mask(apple)
    gaus_mask = gaussian_pyramid(mask1)
    # display the mask and the outputs from multiresolution_blending
    plt.imshow(mask1)
    plt.axis('off')
    plt.show()
    multi_blending_output = multiresolution_blending(apple, orange,gaus_mask)
    plt.imshow(multi_blending_output)
    plt.axis('off')
    plt.show()
    #save the image
    cv2.imwrite('data/multiresolution_blending_apponge.png', cv2.cvtColor(multi_blending_output, cv2.COLOR_RGB2BGR))
```

```python
    # another multiresolution blending
    multi_blending_output = multiresolution_blending(orange, apple, gaus_mask)
    plt.imshow(multi_blending_output)
    plt.axis('off')
    plt.show()
    # save the image
    cv2.imwrite('data/multiresolution_blending_orapple.png', cv2.cvtColor(multi_blending_output, cv2.COLOR_RGB2BGR))
if __name__ == '__main__':
    main()
```

My own work for image blending

My images: (Tiger & Lion)

<Tiger & Lion>

Direct Blending:

Output:

Alpha Blending:

Output:

Multiresolution Blending:

Output: