Contents

1 Basic Test Results

```
Archive:  /tmp/bodek.lfWqDh/impr/ex2/oferaz/presubmission/submission
  inflating: current/README
  inflating: current/sol2.py
  inflating: current/answer_q1.txt
  inflating: current/answer_q2.txt
  inflating: current/answer_q3.txt
```

2 README

```
oferaz
sol2.py
answer_q1.txt
answer_q2.txt
answer_q3.txt
```

3 answer q1.txt

```
Answer to q1:
The difference occur because the image derivation is done
by finding the difference between on pixel to the ones around him,
While in the Fourier domain we are giving more weight to the
high frequency that represent the edge of a shape so this are two different methods so we get two different results,
although quiet similar.
```

4 answer q2.txt

```
Answer to q2:
If the center of the Gaussian will not be in the center of the image than after the dft transform it will not
 be in the center of the image, then it will not affect the high frequencies that responsible for the edging but
 other frequencies, and then after the multiplying - when we will return to the image domain some
 random frequencies will get corrupted
and it will ruin our image instead of blurring it.
```

5 answer q3.txt

```
The differences are:
1. The Computational Complexity of the Fourier domain is much
 better than the one of the image domain because we are using multiplying against convolution, so using fft the all method
 will run in NlogN against N^2
2. The Gaussian is different ? in the Fourier domain we will
 transform him using fft.
```

6 sol2.py

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.misc import imread as imread
from skimage.color import rgb2gray
from scipy.signal import  convolve2d
'''
    :param signal: an array of dtype float32 with shape (N,1).
    :return: fourier_signal -  an array of dtype complex128 with shape (N,1) after dft transform.
    '''
def DFT(signal):
    signal = signal.astype(np.complex128)
    N = signal.shape[0]
    n = np.arange(N)
    k = n.reshape((N, 1))
    kn = k*n
    # calculate the transform matrix.
    dft_matrix = np.exp((-2j * np.pi * kn) / N)
    dftSignal = np.dot(dft_matrix, signal)
    return dftSignal

'''
    :param fourier_signal: : fourier_signal is an array of dtype complex128 with shape (N,1)
    :return: signal -  an array of dtype complex128 with shape (N,1) after dft transform.
    '''
def IDFT(fourier_signal):
    fourier_signal = fourier_signal.astype(np.complex128)
    N = fourier_signal.shape[0]
    n = np.arange(N)
    k = n.reshape((N, 1))
    kn = k*n
    idft_matrix = np.exp((2j * np.pi * kn) / N)
    signal = np.dot(idft_matrix, fourier_signal)
    return signal

'''

    :param image: a grayscale image of dtype float32.
    :return: fourier_image is a 2D array of dtype complex128.
    '''
def DFT2(image):
    image = image.astype(np.complex128)
    N = image.shape[1]
    rows = image.shape[0]
    cols = image.shape[1]
    rows_dft = np.zeros((rows, cols), dtype=np.complex128)
    rows_col_dft = np.zeros((rows, cols), dtype=np.complex128)
    for x in range(rows):
        rows_dft[x,:] = DFT(image[x,:])
    for x in range(cols):
        rows_col_dft[:,x] = DFT(rows_dft[:,x])
    return rows_col_dft

'''

    :param fourier_image: a 2D array of dtype complex128.
    :return: image - a 2D array of dtype complex128.
    '''
def IDFT2(fourier_image):
    fourier_image = fourier_image.astype(np.complex128)
```

```python
    N = fourier_image.shape[1]
    rows = fourier_image.shape[0]
    cols = fourier_image.shape[1]
    rows_dft = np.zeros((rows, cols), dtype=np.complex128)
    rows_col_dft = np.zeros((rows, cols), dtype=np.complex128)
    for x in range(rows):
        rows_dft[x,:] = IDFT(fourier_image[x,:])
    for x in range(cols):
        rows_col_dft[:,x] = IDFT(rows_dft[:,x])

    return rows_col_dft

'''

    :param im: a grayscale image of dtype float32.
    :return magnitude: a magnitude grayscale image of dtype float32.
    '''
def conv_der(im):
    im = im.astype(np.float32)
    conv_matrix_horizontal = np.array([[1,0,-1]])
    #calculate the horizontal derive
    horizontal_derive = convolve2d(im, conv_matrix_horizontal, mode='same')
    conv_matrix_vertical = conv_matrix_horizontal.reshape((3,1))
    # calculate the vertical derive
    vertical_derive = convolve2d(im, conv_matrix_vertical, mode='same')
    return np.sqrt((vertical_derive**2)+(horizontal_derive**2))

'''

    :param im: a grayscale image of dtype float32.
    :return: a magnitude grayscale image of dtype float32.
    '''
def fourier_der(im):
    im.astype(np.float32)
    N = im.shape[0]
    M = im.shape[1]
    V = (np.arange(N) - N/2).reshape((N,1))*(-1)
    U = (np.arange(M) - M/2)
    dft = DFT2(im)

    dft_shifted = np.fft.fftshift(dft)
    A = dft_shifted *np.abs( U)
    B = dft_shifted * np.abs( V)
    a = np.fft.ifftshift(A)
    b = np.fft.ifftshift(B)

    dx = ((2j*np.pi)/N) * IDFT2(a)
    dy = ((2j * np.pi) / M) * IDFT2(b)
    magnitude = (np.sqrt (np.abs(dx)**2 + np.abs(dy)**2))
    return magnitude


'''
    This is a helper function to create kernel gaussian matrix.
    :param kernel_size: The suze of the matrix(size X size).
    :return: gaussian matrix.
    '''
def kernel_maker(kernel_size):
    kernel_row = np.ones((np.floor(kernel_size/2)+1))
    kernel_row = np.convolve(kernel_row,kernel_row,mode='full')

    kernel_col = kernel_row.reshape(kernel_row.size,1)
    kernel = np.zeros((kernel_size, kernel_size))
    kernel[np.floor(kernel_size/2),:] = kernel_row
    kernel = convolve2d(kernel, kernel_col, mode='same')
    kernel /= np.sum(kernel)
    return kernel

```

```python
'''
This function is used for bluring the image using convolution.
    :param im: a greyscale picture.
    :param kernel_size: The suze of the kernel matrix(size X size).
    :return: blured image.
    '''
def blur_spatial(im, kernel_size):
    if kernel_size % 2 == 0:
        return -1 #error for even number
    im = im.astype(np.float32)
    kernel = kernel_maker(kernel_size)
    blur = convolve2d(im,kernel, mode='same')
    return blur

'''
This function is used for bluring the image using dft.
    :param im: a greyscale picture.
    :param kernel_size: The suze of the kernel matrix(size X size).
    :return: blured image.
    '''
def blur_fourier (im, kernel_size):
    if kernel_size % 2 == 0:
        return -1 #error for even number
    im = im.astype(np.float32)
    index_of_middle_row = np.floor(im.shape[1]/2)
    index_of_middle_col = np.floor(im.shape[0]/2)
    kernel = kernel_maker(kernel_size)#getting the kernel filter.
    #lacating the kernel in the center of an im size matrix of zeros.
    kernel_filter = np.zeros((im.shape))
    start_row = index_of_middle_row - np.floor(kernel_size/2)
    start_col = index_of_middle_col - np.floor(kernel_size/2)
    kernel_filter[start_col:start_col+kernel_size, start_row:start_row+kernel_size] = kernel
    #shifting the middle of the kernel to the (0,0) cell.
    kernel_filter = np.fft.ifftshift(kernel_filter)
    kernel_filter_dft = DFT2(kernel_filter)
    im_dft = DFT2(im)
    blur_dft = im_dft * kernel_filter_dft
    blur  = np.fft.ifft2(blur_dft)
    return np.real(blur)

'''
:param fileame: The picture address tou want to open.
:param representation: 1 - for grey, 2 - for colour.
:return: The wanted picture in the wanted format as float representation between 0 to 1.
'''
def read_image(fileame, representation):
    im = imread(fileame)
    im_float = im.astype(np.float32)
    im_float /= 255
    im_g = rgb2gray(im)
    if representation == 1:
        return im_g
    return im_float
```