Contents

1 Basic Test Results

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

  Disclaimer
  ----------
  The purpose of this script is to make sure that your code is compliant
  with the exercise API and some of the requirements
  The script does not test the quality of your results.
  Don't assume that passing this script will guarantee that you will get
  a high grade in the exercise

login:  ITAMAKATZ

submitted files:


==== README for ex1 ===



List of submitted files:



README - this file

answer_qt1.txt - Answer to question Q1

answer_qt2.txt - Answer to question Q2

answer_qt3.txt - Answer to question Q3

sol2.py - python3 code



answer to q1:
Answer to question Q1:

Using a kernel in the spatial dimension is just an approximation
because we use a perticular kernel, while in the frequency domain we
are not restrained to choise of kernel but instead we use the more global
derivatives of the frequency domain.


answer to q2:
Answer to question Q2:

Nothing to serious :)

(spoiler alert - the proof below is pretty cool)

What happens is that the image is broken into four as if we didn't
Use the ifftshift but the intersection of the four blocks is not
```

Necessarily in the middle. This happens because like we learned In class, to be able to use fft, we assume that the image is periodic.

For the more technical explanation, one of the properties of the DFT is that by shifting in the spatial domain we in fact multiply the shifting Amount on the exponent of the W notation. That makes a lot of sense Because shifting is the main operation when computation convolution, and the representation in the frequency domain is a multiplication.

Back to our subject - since the shifting is a multiplication in the Frequency domain and after that we multiply the filter and the image, you could easily argue that the one that was shifted in the first place was not the kernel but the image!!

In 1D it would look something like this (sorry for the informality but its a bit hard to write equations like this..) :

```
conv(x[n], ker[l - m])  <->  X[k] * KER[k] * W ^ (-m)  <->  conv(x[n - m], ker[l])
```

I personally think this is unbelievable! Hot stuff.

To sum up - this means that by shifting the center of he kernel, it simply Moves the center of the image we are blurring.

M.A.S.H.A.L.
answer to q3:
Answer to question Q3:

Well, the main difference is in the complexity. By using fft we can get a complexity of O(N^2 * NlogN) as opposed to O(N^3) using convolution.

Another difference is that by padding the kernel in the spatial domain, we are in fact interpolating the result of the kernel in the frequency domain to match the dimensions. Since interpolation may not be an exact representation, could lose precision of the kernel.

Beside that, due to the padding done in the spatial domain, the resulting image has darker edges since at those location most of thematrix multiplications consist of zeros.

Lastly, there may be a difference in the result due to normalization standards of the procedure. Both in the transformations as well as with the weights of the kernel in the transformation.

```
section 1.1
DFT and IDFT
section 1.2
2D DFT and IDFT
section 2.1
derivative using convolution
Section 2.2
derivative using convolution
Section 3.1
blur spatial
Section 3.1
blur fourier
all tests Passed.
- Pre-submission script done.

  Please go over the output and verify that there are no failures/warnings.
  Remember that this script tested only some basic technical aspects of your implementation
  It is your responsibility to make sure your results are actually correct and not only
  technically valid.
```

2 README

```
ITAMAKATZ

==== README for ex1 ===

List of submitted files:

README - this file
answer_qt1.txt - Answer to question Q1
answer_qt2.txt - Answer to question Q2
answer_qt3.txt - Answer to question Q3
sol2.py - python3 code
```

3 answer q1.txt

```
Answer to question Q1:

Using a kernel in the spatial dimension is just an approximation
because we use a perticular kernel, while in the frequency domain we
are not restrained to choise of kernel but instead we use the more global
derivatives of the frequency domain.
```

4 answer q2.txt

```
Answer to question Q2:

Nothing to serious :)

(spoiler alert - the proof below is pretty cool)

What happens is that the image is broken into four as if we didn't
Use the ifftshift but the intersection of the four blocks is not
Necessarily in the middle. This happens because like we learned
In class, to be able to use fft, we assume that the image is periodic.

For the more technical explanation, one of the properties of the DFT is
that by shifting in the spatial domain we in fact multiply the shifting
Amount on the exponent of the W notation. That makes a lot of sense
Because shifting is the main operation when computation convolution,
and the representation in the frequency domain is a multiplication.

Back to our subject - since the shifting is a multiplication in the
Frequency domain and after that we multiply the filter and the image,
you could easily argue that the one that was shifted in the first
place was not the kernel but the image!!

In 1D it would look something like this (sorry for the informality but
its a bit hard to write equations like this..) :


conv(x[n], ker[l - m])  <->  X[k] * KER[k] * W ^ (-m)  <->  conv(x[n - m], ker[l])

I personally think this is unbelievable! Hot stuff.

To sum up - this means that by shifting the center of he kernel, it simply
Moves the center of the image we are blurring.

M.A.S.H.A.L.
```

5 answer q3.txt

```
Answer to question Q3:

Well, the main difference is in the complexity. By using fft
we can get a complexity of O(N^2 * NlogN) as opposed to O(N^3)
using convolution.

Another difference is that by padding the kernel in the spatial
domain, we are in fact interpolating the result of the kernel
in the frequency domain to match the dimensions. Since
interpolation may not be an exact representation, could lose
precision of the kernel.

Beside that, due to the padding done in the spatial domain, the
resulting image has darker edges since at those location
most of thematrix multiplications consist of zeros.

Lastly, there may be a difference in the result due to
normalization standards of the procedure. Both in the
transformations as well as with the weights of the kernel in
the transformation.
```

6 sol2.py

```
import numpy as np
import scipy.special
from scipy.misc import imread
from skimage.color import rgb2gray
from scipy.signal import convolve2d

# numerical precision tu truncate using the round function
NUMERIC_ERROR = 13

def read_image(filename, representation):
    # filename - file to open as image
    # representation - is it a B&W or color image

    im = imread(filename)
    # check if it is a B&W image
    if(representation == 1):
        im = rgb2gray(im)
    # convert to float and normalize
    return im.astype(np.float32) / 255

def General_DFT(x, mult):
    # x - array to transform
    # mult - 1 or -1 to define if DFT or IDFT
    # return transform rounded of numerical error

    # compute vander matrix
    vander = np.vander((np.exp(mult * 2 * np.pi * 1j * np.arange(x.shape[0]) / x.shape[0])), increasing=True)
    # return vander.dot(x)
    return vander.dot(x)

def General_DFT2(x, mult):
    # x - array to transform
    # mult - 1 or -1 to define if DFT or IDFT
    # return transform rounded of numerical error

    # compute vander matrix of both dims to perform v_M*x*v_N
    vander_M = np.vander((np.exp(mult * 2 * np.pi * 1j * np.arange(x.shape[0]) / x.shape[0])), increasing=True)
    vander_N = np.vander((np.exp(mult * 2 * np.pi * 1j * np.arange(x.shape[1]) / x.shape[1])), increasing=True)
    return np.around(vander_M.dot(x.dot(vander_N)), NUMERIC_ERROR)

def DFT(signal):
    return General_DFT(signal, -1)

def IDFT(fourier_signal):
    # round of numerical error and normalize
    return np.around(General_DFT(fourier_signal, 1), NUMERIC_ERROR) / fourier_signal.shape[0]


def DFT2(image):
    return General_DFT2(image, -1)

def IDFT2(image):
     # normalize by both dimensions
    return General_DFT2(image, 1) / (image.shape[0] * image.shape[1])

def conv_der(im):
    # normalize and compute conv with padding
    im = im / 255
    xDer = convolve2d(im, np.array([[1, 0, -1]]), mode='same')
```

```python
    yDer = convolve2d(im, np.array([[1],[0],[-1]]), mode='same')

    # compute power
    return np.sqrt(np.abs(xDer)**2 + np.abs(yDer)**2)

def fourier_der(im):

    # compute the frequency derivatives for multiplication
    u, v = np.meshgrid(np.arange(-im.shape[1] / 2, im.shape[1] / 2 - 1 * im.shape[1] % 2),
                       np.arange(-im.shape[0] / 2, im.shape[0] / 2 - 1 * im.shape[0] % 2))

    # no need to normalize since using func General_DFT2
    xDer = 2 * np.pi * 1j * General_DFT2(u * np.fft.fftshift(General_DFT2(im, -1)), 1)
    yDer = 2 * np.pi * 1j * General_DFT2(v * np.fft.fftshift(General_DFT2(im, -1)), 1)

    # compute power
    return np.sqrt(np.abs(xDer) ** 2 + np.abs(yDer)**2)

def create_ker(kernel_size):
    # kernel_size - odd integer
    # returns a binomial kernel of size kernel_size X kernel_size approximating a gausian
    bin = scipy.special.binom(kernel_size - 1, np.arange(kernel_size)).astype(np.int64)
    kernel = convolve2d(bin[np.newaxis, :], bin[:, np.newaxis])
    return kernel / np.sum(kernel)

def blur_spatial (im, kernel_size):
    return convolve2d(im, create_ker(kernel_size), mode='same')

def blur_fourier (im, kernel_size):
    ker = create_ker(kernel_size)

    center = (int(np.floor(im.shape[0] / 2)), int(np.floor(im.shape[1] / 2)))

    # def ker_f to be the size of im and add it the ker
    ker_f = im * 0
    ker_f[np.meshgrid(
        np.arange(center[0] - int((kernel_size - 1) / 2), center[0] + int((kernel_size - 1) / 2) + 1),
        np.arange(center[1] - int((kernel_size - 1) / 2), center[1] + int((kernel_size - 1) / 2) + 1))] = ker

    ker_f = np.fft.ifftshift(ker_f)
    ker_f = DFT2(np.copy(ker_f))
    im_f = DFT2(im)
    # return of type float32
    return (abs(IDFT2(im_f * ker_f))).astype(np.float32)
```