Image Processing - 67829

Exercise 2: Fourier Transform & Convolution

Due date: 8/12/2016

The purpose of this exercise is to help you understand the concept of the frequency domain by performing simple manipulations on images. This exercise covers:

- Implementing Discrete Fourier Transform (DFT) on 1D and 2D signals
- Performing image derivative
- Convolution theory

1 Discrete Fourier Transform - DFT

1.1 1D DFT

Write two functions that transform a 1D discrete signal to its Fourier representation and vice versa. The two functions should have the following interfaces:

```
fourier_signal = DFT(signal)
```

where: `signal` is an array of dtype float32 with shape (N,1) (technically it's 2D), and `fourier_signal` is an array of dtype complex128 with the same shape.

```
signal = IDFT(fourier_signal)
```

where: `fourier_signal` is an array of dtype **complex128** with shape (N,1), and `signal` has the **same** shape and dtype. Note that when the origin of `fourier_signal` is a transformed real signal you can expect `signal` to be real valued as well, although it may return with a tiny imaginary part. You can ignore the imaginary part (see tips).

note: "IDFT" stands for inverse DFT.

You should use the following formulas:

For DFT transform:

$$F(u) = \sum_{x=0}^{N-1} f(x) e^{-\frac{2\pi i u x}{N}}$$

And for IDFT transform:

$$f(x) = \frac{1}{N} \sum_{u=0}^{N-1} F(u) e^{\frac{2\pi i u x}{N}}$$

Both functions should be implemented without loops.

1.2 2D DFT

Write two functions that convert a 2D discrete signal to its Fourier representation and vice versa. The two functions should have the following interfaces:

```
fourier_image = DFT2(image)
```

where: `image` is a grayscale image of dtype float32, and `fourier_image` is a 2D array of dtype complex128.

```
image = IDFT2(fourier_image)
```

where: `fourier_image` is a 2D array of dtype **complex128**, and `image` has the **same** shape and dtype. Note that when the origin of `fourier_image` is a real image transformed with DFT2 you can expect the returned `image` to be real valued as well, although it may return with a tiny imaginary part. You can ignore the imaginary part (see tips).

You should use the following formulas:

For DFT2:

$$F(u,v) = \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} f(x,y) e^{-2\pi i (\frac{ux}{N} + \frac{vy}{M})}$$

And for IDFT2:

$$f(x,y) = \frac{1}{NM} \sum_{u=0}^{N-1} \sum_{v=0}^{M-1} F(u,v) e^{2\pi i (\frac{ux}{N} + \frac{vy}{M})}$$

Where M and N are the numbers of rows and columns, respectively.

Note: You should implement these 2D transformation using subsequent 1D transformations (section 1.1). You can loop over one dimension and call 1D transformation each time on one vector.

Note that you should not implement the Fast Fourier Transform.

1.2.1 Bonus - 5 points

Make your 1D transformations to be compatible with an input of a matrix, and implement DFT2 and IDFT2 without loops.

2 Image derivatives

2.1 Image derivatives in image space

Write a function that computes the magnitude of image derivatives. You should derive the image in each direction separately (vertical and horizontal) using simple convolution with $[1, 0, -1]$ as a row and column vectors, to get the two image derivatives. Next, use these derivative images to compute the magnitude image. The function should have the following interface:

```
magnitude = conv_der(im)
```

Where the input and the output are grayscale images of type float32.

2.2 Image derivatives in Fourier space

Write a function that computes the magnitude of image derivatives using Fourier transform. Use the formula from class to derive in the x and y directions. You are recommended to use `np.fft.fftshift` in the frequency domain so that the (U,V)=(0,0) frequency will be at the center of the image. The function should have the following interface:

```
magnitude = fourier_der(im)
```

Where the input and the output are float32 grayscale images.
magnitude is the magnitude of the derivative and should be calculated in the following way:

```
magnitude = np.sqrt (np.abs(dx)**2 + np.abs(dy)**2)
```

In both sections (2.1 and 2.2) you should <u>not</u> normalize the magnitude values to be in the range of [0,1], just return the values you get.

Q1: Why did you get two different magnitude images?

3 Convolution theory

In this section we will get familiar with the convolution theory. You should blur an image f in two ways:

- using a convolution with gaussian kernel g in image space
- using a point-wise operation between the fourier image F with the same gaussian kernel, but in fourier space G.

3.1 Blurring in image space

Write a function that performs image blurring using 2D convolution between the image f and a gaussian kernel g. The function should have the following interface:

```
blur_im = blur_spatial (im, kernel_size)
```

where:
`im` - is the input image to be blurred (grayscale float32 image).
`kernel_Size` - is the size of the gaussian kernel in each dimension (an odd integer).
`blur_im` - is the output blurry image (grayscale float32 image).

Comments:

- The gaussian kernel g should contain approximation of the gaussian distribution using the binomial coefficients. A consequent 1D convolutions of [1 1] with itself is an elegant way for deriving a row of the binomial coefficients. Explore how you can get a 2D gaussian approximation using the 1D binomial coefficients. Pay attention that the kernel elements should be summed to 1.
- Once you have a 2D gaussian kernel, you can convolve the image with the kernel using the function `convolve2d` from `scipy.signal`.
- You can handle the border issue as you like. (zero padding, cyclic image...).

3.2 Blurring in Fourier space

Write a function that performs image blurring with gaussian kernel in Fourier space. The function's interface:

```
blur_im = blur_fourier (im, kernel_size)
```

where:
`im` - is the input image to be blurred (grayscale float32 image).
`kernel_size` - is the size of the gaussian in each dimension (an odd integer).

`blur_im` - is the output blurry image (grayscale float32 image).

In this function you should create the 2D gaussian kernel g in the same way you created in section 3.1 and transform it to its Fourier representation G. You should also transform the image f to its Fourier representation F and multiply point-wise $F \cdot G$. Then, you should perform the inverse transform on the result in order to get back to the image space.
Comments:

- The size of the kernel g is $kernel\_sizeXkernel\_size$ pixels. Since its Fourier representation will have the same size it can not be point-wise multiplied with F (which has the same size as the image). Therefore, you should pad the gaussian kernel, g, with zeros to bring it to the same shape as the image while preserving the center of the gaussian at (0,0).

- To do so, you are recommended to build a 2D-gaussian kernel in the center of an image. You should locate the center of the gaussian kernel at $[floor(m/2) + 1 \quad floor(n/2) + 1]$, where (m, n) are the dimensions of the image. The convention for even number of pixels is the same. Then you can use `np.fft.ifftshift` so the center of the gaussian will be at the origin (0,0) of the image.

Q2: What happens if the center of the gaussian (in the space domain) will not be at the (0,0) of the image? Why does it happen?
Q3: What is the difference between the two results (Blurring in image space and blurring in Fourier space)?

4 Important Comments

- Use `convolve2d` with the `'same'` option when you perform the convolution operation (in section 2.1 and 3.1).

- In order to compute the Fourier transform in sections 2.2 and 3.2 you should use **your** 2D implementation from section 1.2.

5 Some tips

- **Fourier centering:** The output of your `DFT2` implementation is a matrix which contains the Fourier coefficients. This matrix is organized s.t. $F(0, 0)$ is located at the origin (0,0) (top left corner). However, visualizing the Fourier coefficients may be easier to do with $F(0, 0)$ shifted to the

center of the matrix. For this purpose use `np.fft.fftshift` which performs a *cyclic translation* of a matrix in both axes (try to shift a grayscale image to understand the behavior of this function). Remember to shift back using `np.fft.ifftshift` before transforming back to the image domain.

- **Fourier display:** To best visualize a Fourier coefficient image `im_f` you should first apply the intensity transformation `np.log(1+np.abs(im_f))` discussed in class. This operation reduces the dynamic range of the coefficient magnitude image and will therefore cause more values to become visible (try to display `im_f` with and without this transformation and compare what you see).

- **Useful functions:**

  - `scipy.signal.convolve2d` 2D convolution – use the `'same'` option when you want the output to have the same size as the input
  - `np.meshgrid` used to create index maps, you can use `np.arange` instead, and perform the same operations via broadcasing
  - `np.complex128` dtype of array with complex numbers.
  - `np.fft.fft2, np.fft.ifft2` 2D discrete Fast Fourier Transform (and inverse). You can use these functions to check your results from section 1.1 and 1.2
  - `np.real` (or `np.real_if_close`) When you return from the frequency domain to the image domain, there might be some very small imaginary part in the matrix elements due to numerical errors. You can ignore them and take only the real part of the matrix.

6 Submission

Submission instructions may be found in the "Exercise Guideline's" document published on the course web page. Please read and follow them carefully. You should answer the questions in sections 2 and 3 in answer files named *answer_q*1.*txt*, *answer_q*2.*txt*, and *answer_q*3.*txt*.

Good luck and enjoy!