Problem Set 3: Motion Magnification and Texture Synthesis

Posted: Thursday, January 28, 2020	**Due:** Tuesday, February 4, 2020

This homework is based on a problem set from the MIT Vision Course [1]. For this problem set, all your work and writeup should be done in the colab notebook. You will not need to submit anything to Gradescope. Your should submit to Canvas a zip file that includes a `.ipynb` file and a video file (for the motion magnification problem).

The starter code can be found at:
https://drive.google.com/open?id=1HVscDzyaHo2xBQ9qRTg3fEXhlYZJNZ54

We recommend editing and running your code in Google Colab, although you are welcome to use your local machine instead.

Problem 3.1 *Motion Magnification*

In this problem we will investigate motion magnification in videos [2]. We'll take a video that has very tiny, subtle motions and increase their size, thereby making them easier for you to see.

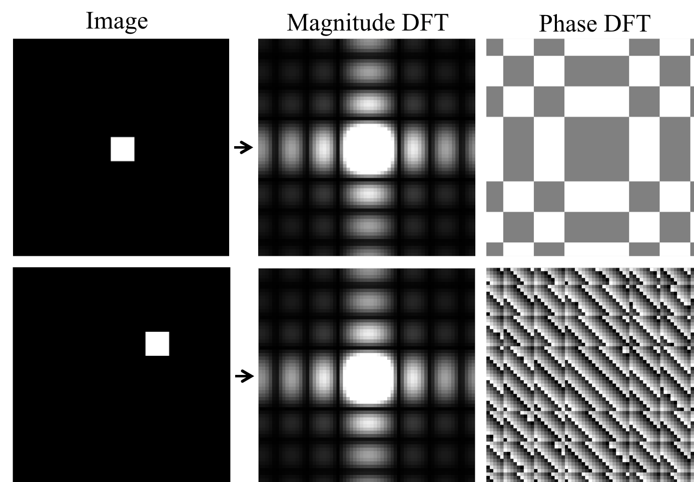

Figure 1: Shifts of an image only produce changes on the phase of the DFT

Recall that position shifts in image space correspond to phase shifts in the frequency domain of the Fourier transform. This means that for two images, we can compare the Fourier transform of the two images to find the phase shift between the images. Amplifying the phase shift by a

fixed factor in the Fourier transform frequency domain will amplify the position shift by the same factor in the image domain after we perform the inverse Fourier transform. We will use this idea to exaggerate the motions in videos.

(a) For a purely horizontal offset of an impulse signal, magnifying the phase shift will result in a magnified horizontal offset after the inverse transform. Please fill in `TODO1` and `TODO2` in `magnifyChange`. You should find the phase shift between the two input images and magnify it by the specified `magnificationFactor`. When complete, the function `magnifyChange` should return an image showing what image 2 would look like with the magnified offset. Please run `Problem 3.a` and generate a plot.

(b) If there is motion in more than one direction between two images, we will see that naively magnifying the phase shift of the whole images will not work. In `Problem 3.b`, we have set up a vertical offset of an impulse signal as well as the horizontal one from part a. Please run `Problem 3.b` and generated a plot, then explain why the two offsets were not properly magnified (you can either add a text cell for this or write your answer in the same cell).

(c) One strategy we can use if there are multiple motions between two images is to do a localized Fourier transform by independently magnifying the offsets on small windows of the images and aggregating the results across the windows. When we restrict our window of consideration, it is more likely for everything in the window to be moving the same way. We will use Gaussian filters to mask small windows of the image and perform magnification on each window independently. In `Problem 3.c`, please fill in the Gaussian filter in `TODO3` and the appropriately windowed input images in `TODO4`. Since we are working with images, we will use the discrete Gaussian filter rather than the continuous one. Run `Problem 3.c` to confirm that the two motions were properly magnified and generate a plot.

(d) We are now ready to apply motion magnification to videos. In particular, we will demonstrate the method using the original video from the paper, which features one of the authors (Bill Freeman) himself. We will use the same approach as in part c of magnifying Gaussian windowed regions of the video frames. Rather than directly finding the phase shifts between consecutive video frames, we will keep a moving average of the Fourier transform phases and compare each new frame's DFT phase with the current moving average of phase. The moving average is an IIR low-pass filter, averaging 0.5 times the previous average with 0.5 times the current phase. For simplicity, each of the RGB channels are processed independently and identically. In `Problem 3.d`, you will need to fill in the Gaussian filter in `TODO5`, the DFT phase of the magnified window in `TODO6`, and the DFT of the magnified window in `TODO7`. Please run `Problem 3.d` and submit the generated video (`bill_magnified.avi`) in a zip file with the ipynb notebook to Canvas. Note that the code may take some time to run - you can temporarily modify `sigma` to decrease the number of windowed regions to process.

Problem 3.2 *Texture Synthesis*

In this problem, you will implement the Efros and Leung algorithm [3] for texture synthesis. Texture synthesis, as discussed in class, is a method used for generating new textures from an initial sample texture. It has a broad range of applications in graphics, hole filling, image denoising, etc. You may find it helpful to refer to the pseudo-code at this link: https://people.eecs.berkeley.edu/ efros/research/NPS/alg.html

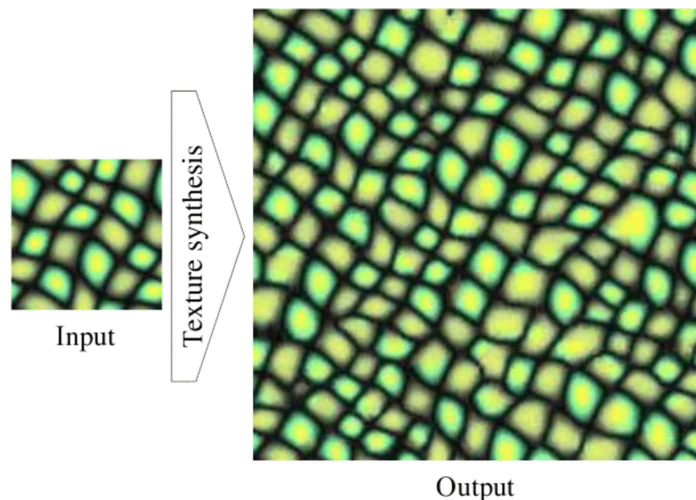

Figure 2: Example of texture synthesis output.

In this algorithm, we will build up a new texture pixel by pixel. We start by sampling a small patch from the source texture and use it to initialize the center of the new texture. We then recursively fill in the boundaries of this patch until all the pixels in the target have been considered. For each pixel, we take a window and do an exhaustive search over the source texture to find the closest window. There are many distances metrics we could use for comparison, but for simplicity, we will use the sum-of-squared differences (SSD). We take the center pixel of this window and copy its color into the target texture. We also want to give pixels near the center of the window more weight because they are probably more important in finding a well-matched window. We do this by weighing the window with a Gaussian kernel. This has the effect such that the closer a pixel is to the center, the more it contributes to the distance metric.

We ask you to implement the function `synthesize_texture(sample, window_size, s)` where `sample` is the source texture image and `s` is a tuple of the new image height and width.

In addition, we also ask you to write a helper function `find_matches(template, sample, gauss_mask)` for returning a list of candidate matches in the source texture along with their corresponding SSD errors. In this function, `sample` is the source texture image, `template` is a $w \times w$ window from the target image where w is the window size, `gauss_mask` is a 2D Gaussian mask.

This function returns a list of `best_matches` and we randomly select one to synthesize a pixel in the final image. To form the `best_matches` list, accept all pixels locations whose SSD error is less than the minimum SSD value times $(1 + \epsilon)$. To avoid randomly selecting a match with unusually large error, also check that the error of the randomly selected match is below a threshold δ. Efros and Leung uses threshold values $\epsilon = 0.1$ and $\delta = 0.3$.

Note that template can have values that have not yet been filled in by the image growing routine. Mask the template image such that these values are not considered when computing SSD. Efros and Leung suggest using the following image mask:

$$\texttt{mask} = \texttt{gauss_mask} \odot \texttt{valid_mask}$$

where `valid_mask` is a square mask of width w that is 1 where the template is filled, 0 otherwise, `gauss_mask` is a 2D zero-mean Gaussian with variance $\sigma = w/6.4$ sampled on a $w \times w$ grid centered about its mean, and $\odot$ is pointwise multiplication. G can be pre-computed using `scipy.ndimage.gaussian_filter` routine. The purpose of the Gaussian is to down-weight pixels that are farther from the center of the template. Also, make sure to normalize the mask such that its elements sum to 1.

Run your implementation using the source texture image `rings.jpg` provided in the homework with window sizes of $w = 5, 7, 13$, $s = [100, 100]$ with an initial starting seed at $(x, y) = (4, 32)$. Explain the algorithm's performance with respect to window size. For a given window size, if you re-run the algorithm with the same starting seed do you get the same result? Why or why not? Is this true for all window sizes? Also run your implementation using real color image `texture.jpg` and see how window size affects the performance.

For fun, we ask you to choose your own sample texture and run the synthesis algorithm on it. Include your sample patch and the output texture in your report. Feel free to be creative; we will be grading this part based on completion.

In total, you should have 3 images for `rings.jpg`, 1 image for `texture.jpg` and one of your own choosing.

References:

[1] : 6.819/6.869: Advances in Computer Vision course taught by Bill Freeman, Antonio Torralba, and Phillip Isola

[2] : http://people.csail.mit.edu/celiu/motionmag/motionmag.html

[3] : https://people.eecs.berkeley.edu/~efros/research/EfrosLeung.html