

Problem Set 5: Texture Synthesis

Posted: Thursday, October 3, 2013

Due: Midnight, Thursday, October 10, 2013

You should submit a hard copy of your work in class, and upload your code (and all files needed to run it, images, etc) to Stellar. If you're not done by class, please upload everything to Stellar by midnight.

Your report should include images and plots showing your results, as well as pieces of your code that you find relevant.

Problem 1 *Texture synthesis*

In this problem you will implement the Efros and Leung algorithm for texture synthesis discussed in Section 9.3 of Forsyth and Ponce. In addition to reading the textbook you may also find it helpful to visit Efros' texture synthesis website:

<http://graphics.cs.cmu.edu/people/efros/research/synthesis.html>,

in which many of the implementation details described below can be found. As discussed in class, the Efros and Leung algorithm synthesizes a new texture by performing an exhaustive search of a source texture for each synthesized pixel in the target image, in which sum-of-squared differences (SSD) is used to associate similar image patches in the source image with that of the target. The algorithm is initialized by randomly selecting a 3×3 patch from the source texture and placing it in the center of the target texture. The boundaries of this patch are then recursively filled until all pixels in the target image have been considered. Implement the Efros and Leung algorithm as the following MATLAB function:

```
synthIm = SynthTexture(sample, w, s)
```

where `sample` is the source texture image, `w` is the width of the search window, and `s=[ht, wt]` specifies the height and width of the target image `synthIm`. As described above, this algorithm will create a new target texture image, initialized with a 3×3 patch from the source image. It will then grow this patch to fill the entire image. As discussed in the textbook, when growing the image un-filled pixels along the boundary of the block of synthesized values are considered at each iteration of the algorithm. A useful technique for recovering the location of these pixels in MATLAB is using *dilation*, a morphological operation that expands image regions.

Use MATLAB's `imdilate` and `find` routines to recover the un-filled pixel locations along the boundary of the synthesized block in the target image.

In addition to the above function we ask you to write a subroutine that for a given pixel in the target image, returns a list of possible candidate matches in the source texture along with their corresponding SSD errors. This function should have the following syntax:

```
[bestMatches, errors] = FindMatches(template, sample, G)
```

where `bestMatches` is the list of possible candidate matches with corresponding SSD errors specified by `errors`. `template` is the $w \times w$ image template associated with a pixel of the target image, `sample` is the source texture image, and `G` is a 2D Gaussian mask discussed below. This routine is called by `SynthTexture` and a pixel value is randomly selected from `bestMatches` to synthesize a pixel of the target image. To form `bestMatches` accept all pixel locations whose SSD error values are 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 use threshold values of $\epsilon = 0.1$ 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:

```
Mask = G .* validMask
```

where `validMask` is a square mask of width w that is 1 where the template is filled, 0 otherwise and `G` is a 2D zero-mean Gaussian with variance $\sigma = w/6.4$ sampled on a $w \times w$ grid centered about its mean. `G` can be pre-computed using MATLAB's `fspecial` 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.

Test and run your implementation using the grayscale source texture image `rings.jpg` available on the class website, with window widths of $w = 5, 7, 13$, $s = [100, 100]$ and an initial starting seed of $(x, y) = (4, 32)$. 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?

In your report, include the synthesized textures that correspond to each window size along with answers to the above questions.