Computer Vision
Mini-project 3
CS 670, Fall 2019

Name: Kunjal Panchal
Student ID: 32126469
Email: kpanchal@umass.edu

October 23, 2019

Contents

1 Problem 1: Image Denoising

Let's take a look at some terms useful for understanding this problem:

- IMAGE DENOISING Images taken with both digital cameras and conventional film cameras will pick up noise from a variety of sources. Further use of these images will often require that the noise be (partially) removed.
- GAUSSIAN NOISE Variations in intensity drawn from a Gaussian normal distribution. [1]
- SALT AND PEPPER NOISE Contains random occurrences of black and white pixels.
- LINEAR SMOOTHING FILTERS One method to remove noise is by convolving the original image with a mask that represents a low-pass filter or smoothing operation. For example, the Gaussian mask comprises elements determined by a Gaussian function. [2]
- NON LINEAR FILTERS A median filter is an example of a non-linear filter and, if properly designed, is very good at preserving image detail.
- NON LOCAL MEANS Another approach for removing noise is based on non-local averaging of all the pixels in an image. In particular, the amount of weighting for a pixel is based on the degree of similarity between a small patch centered on that pixel and the small patch centered on the pixel being de-noised. [2]

1.1 (a) Gaussian Filtering

The idea of Gaussian smoothing is to use this 2-D distribution as a 'point-spread' function, and this is achieved by convolution.

Since the image is stored as a collection of discrete pixels we need to produce a discrete approximation to the Gaussian function before we can perform the convolution.

In theory, the Gaussian distribution is non-zero everywhere, which would require an infinitely large convolution kernel, but in practice it is effectively zero more than about three standard deviations from the mean, and so we can truncate the kernel at this point.

Figure 1 shows a suitable integer-valued convolution kernel that approximates a Gaussian with a σ of 1.0.

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Figure 1: Discrete approximation to Gaussian function with σ=1.0

We convolute all noisy images with gaussian filters first, we also experiment with different σ standard deviations and find out which value gives the optimal results.

NOTE: Matlab support says that the recommended replacement functions, "imgaussfilt" and "imgaussfilt3", are "generally faster and more advanced" with comparison to “fspecial('gaussian',hsize,sigma)”. Also they point out that the documentation for "imgaussfilt" states that the "A" matrix input can be of any dimension, so it should work for N-D images.

Hence, we use “imgaussfilt” instead of combination of “fspecial” (to create a gaussian filter kernel) and “imfilter” (to convolute it with noisy image and get a denoised image).

This is shown in **line 7 and 8** of code snippet 1.1.

Matlab Implementation of Gaussian Filters

```
1 %% Denoising algorithm (Gaussian filtering)
2
3 count = 1;
4 % Experiment with different values of standard
    deviation
```

Image Name	Optimal Standard Deviation σ	Least Error - SSD
saturn-noise1g.png	1.09	148.91
saturn-noise1sp.png	1.55	165.32
saturn-noise2g.png	1.71	556.21
saturn-noise2sp.png	1.90	349.12

Table 1: Optimal values of standard deviation for which we get the most similar denoised image, in comparison to the original image; using Gaussian Filters

```
for i = 1.85:0.01:1.95
    % Give noisy image and std dev as parameters to gaussian filter to get a denoised image
    filtered3 = imgaussfilt(noise1, i);
    filtered4 = imgaussfilt(noise2, i);

    % Compute errors - SSD
    error3 = sum(sum((im - filtered3).^2));
    error4 = sum(sum((im - filtered4).^2));

    % Print results
    fprintf('Gaussian Filter Simga %.2f- Input, Errors : %.2f %.2f\n', i, error3, error4)

    %Display results
    figure(2);
    subplot(5, 8, count); imshow(filtered3); title(sprintf('SE %.2f', error3));
    figure(3);
    subplot(5, 8, count); imshow(filtered4); title(sprintf('SE %.2f', error4));
    count = count + 1;
end
```

We report the optimal error and standard deviation σ here [See Table 1], and show the output denoised images with those optimal parameters, for better comparison with other values.

1.2 (b) Median Filtering

A median filter operates over a window by selecting the median intensity in the window.

The window of a 2D median filter can be of any central symmetric shape, a round disc, a square, a rectangle, or a cross. The pixel at the center will be replaced by the median of all pixel values inside the window.

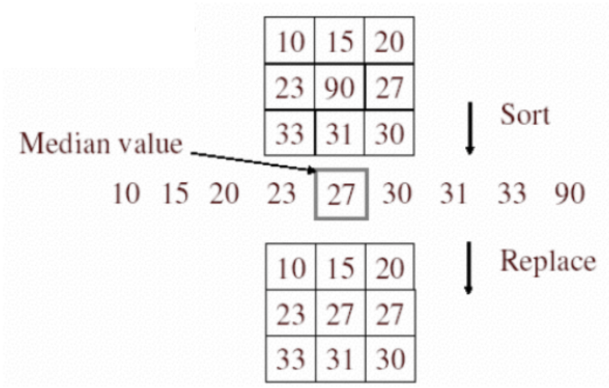


Figure 2: A 3 x 3 median filter and how a pixel value is determined by operating around its neighborhood [1]

We use `medfilt2(I, [m n])`, which performs median filtering, where each output pixel contains the median value in the m-by-n neighborhood around the corresponding pixel in the input image.

This is shown in **line 12 and 13** of code snippet 1.2.

Matlab Implementation of Median Filters

```
1 %% Denoising algorithm (Median filtering)
2
3 % To find the std dev where there is least SSD
4 min5 = Inf; index_x5 = 0; index_y5 = 0;
5 min6 = Inf; index_x6 = 0; index_y6 = 0;
6
7 % Experiment with different filter kernel dimensions
```

```
for i = 1:10
    for j = 1:10

        % Convolute noisy image with i x j dimension median filter kernel
        filtered5 = medfilt2(noise1, [i, j]);
        filtered6 = medfilt2(noise2, [i, j]);

        % Compute errors - SSD
        error5 = sum(sum((im - filtered5).^2));
        error6 = sum(sum((im - filtered6).^2));
        fprintf('Median Filter Neighborhood %d %d-Input, Errors: %.2f %.2f\n', i, j, error5, error6)

        % Save the filter dimensions of the kernel with the least error
        if error5 < min5
            min5 = error5;
            index_x5 = i;
            index_y5 = j;
        end
        if error6 < min6
            min6 = error6;
            index_x6 = i;
            index_y6 = j;
        end
    end
end

% Print the results
fprintf('Median Filter Neighborhood %d %d- image1, Error: %.2f\n', index_x5, index_y5, min5)
fprintf('Median Filter Neighborhood %d %d- image2, Error: %.2f\n', index_x6, index_y6, min6)
```

We report the optimal error and filter kernel dimensions m x n here [See Table 2], and show the output denoised images with those optimal parameters, for better comparison with other values.

Image Name	Optimal Kernel Dimensions mxn	Least Error - SSD
saturn-noise1g.png	5 x 7	90.11
saturn-noise1sp.png	3 x 3	8.78
saturn-noise2g.png	7 x 9	225.28
saturn-noise2sp.png	3 x 3	18.57

Table 2: Optimal values of filter kernel dimensions for which we get the most similar denoised image, in comparison to the original image; using Median Filters

1.3 (c) Non-Local Means Filtering

In this method, for each pixel in the input image the algorithm computes the denoised value as weighted mean of all the pixel values within the window, where the weight is some inverse function of the distance between patches centered at the pixels.

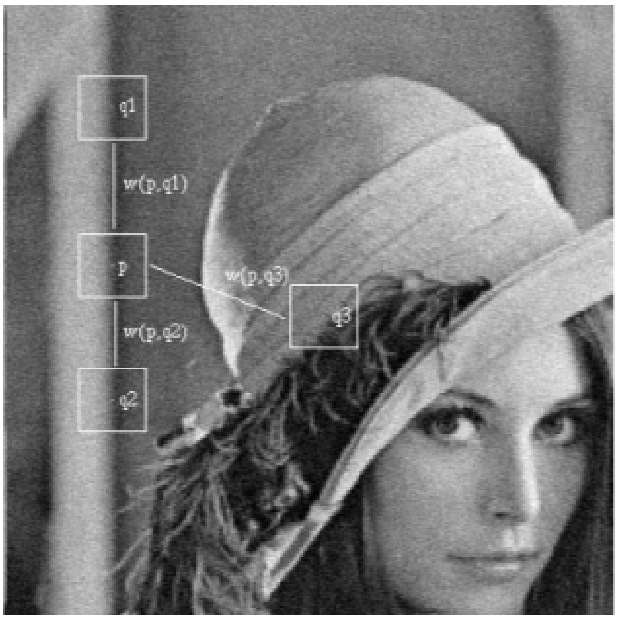


Figure 3: Non-Local Means Filtering - Similar pixel neighborhoods give a large weight, w(p,q1) and w(p,q2), while much different neighborhoods give a small weight w(p,q3). [4]

Now we point out the steps of our implementation, the full matlab code for it was too long to include here, so it can be found in Appendix A.2

- If we want a patch of size n x n, its radius or half width will be $\frac{n-1}{2}$. The -1 is for the center of the patch, the pixel for which we want a n xn patch around it. So, for a 5 x 5 patch, the `halfPatchSize` is 2.

```
halfPatchSize = 2;  % Patch Size (halved)
windowHalfSearchSize = 10;  % Window Size (halved)
gamma = 0.01;          % Bandwidth parameter
```

- Same concept as above for the window size, which is a frame made up of top, bottom, left and right edges; being certain distance away from the pixel in concern. So, for a 21 x 21 patch, the `windowHalfSearchSize` is 10.

- Keep in mind that, the window is NOT a patch in the sense that a patch include ALL the pixels from its left border to the right and top to the bottom. While a window is ONLY those borders.

- Each pixel of that border will have their own patches of `patchSize`, the border pixel being the center of the patch. These patches will be compared to the patch whose center is the center of the window and whose denoised value we want to compute.

```
% Get the patch with the pixel (i, j) as its center and halfPatchSize as its radius
px = img_n(max(1, j - halfPatchSize):min(w, j + halfPatchSize), max(1, i - halfPatchSize):min(h, i + halfPatchSize));
```

- Initially, we set bandwidth parameter γ to 0.01.

- We loop through all pixels of the image: first we get patch P_x for each pixel (x, y) by Matlab's matrix cut/crop out mechanism.

- We initialize the nominator and denominator sums to 0.0 for the denoised pixel equation

$$\hat{I}(x) = \frac{\sum_{y \in W(x)} \exp\{-\gamma ||P(x) - P(y)||_2^2\} I(y)}{\sum_{y \in W(x)} \exp\{-\gamma ||P(x) - P(y)||_2^2\}}$$

- Then we go through each of the four window edges around the pixel (x, y) and get the patches P_y for the pixels on the window edges, calculate the dissimilarity between P_x and P_y through SSD. Example for Left Border:

```
% For each pixel on vertical edges of the window specified by its half size
for k = max(1, j - windowHalfSearchSize):min(w, j + windowHalfSearchSize)

        % For left edge of the window
        if 1 <= i - windowHalfSearchSize
                % Get a patch around that particular pixel on the left edge of the window
                py = img_n(max(1, k - halfPatchSize):min(w, k + halfPatchSize), max(1, i - windowHalfSearchSize - halfPatchSize):min(h, i - windowHalfSearchSize + halfPatchSize));
                % Find out minimum SSD
                if size(px) == size(py)
                        s = (px - py).^2;
                        s = s(:);
                        d = sqrt(sum(s));
                        d1 = exp(-1 * gamma * d);
                        sum_dn = sum_dn + d1;
                        sum_up = sum_up + d1 * img_n(j, i);
                end
        end
end
```

- We put that SSD values and other multiplication factors (bandwidth parameters and pixel values) in the given equation and after going through all four edges, get the final nominator and denominator sums.

- Dividing those will give us the denoised value for pixel (x, y).

Image Name	Least Error - SSD
saturn-noise1g.png	102.8368
saturn-noise1sp.png	95.7399
saturn-noise2g.png	534.652
saturn-noise2sp.png	272.9644

Table 3: Optimal error values at $\gamma = 0.2$ using Non-Local Means Filters

```
% Get the new value for the pixel (i, j)
if sum_dn ~= 0
        img_f(j, i) = sum_up/sum_dn;
        disp(img_f(j,i))
end
```

We experimented with different γ values in the range of [0.01 2], we found that $\gamma = 0.2$ gives the least errors. We report the optimal error [See Table 3], and show the output denoised images with those optimal parameters, for better comparison with other values.

As we can see, non-local means filtering works better on salt and pepper noise, then it does for gaussian noise. Thorough analysis given in the next section.

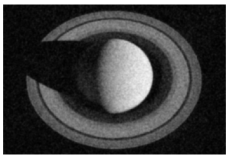
1.4 Results and Conclusions

From Figures 4, 5 and 6; we can infer the following:

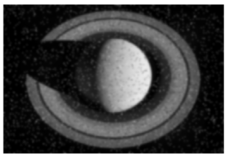
- MEDIAN FILTERS work best for salt and pepper noise, even on highly denoised image, it had an SSD between denoised and original image, of only 18.57.

- That is because salt and pepper noise only affects a pixel, the rest of the neighboring pixels do not change their values, so it is pretty easy to estimate the pixel's value, when adjacent pixels have their values intact, unlike gaussian noise, which affects all the neighbors, even if it the noise is proportional to the distance, the main takeaway is that it changed the pixel values, therefore, the whole image looks blurry as the result.

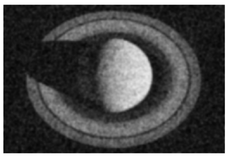
- NON-LOCAL MEANS FILTERS outperforms Gaussian Filters in all four cases, in some cases, from a very little margin.



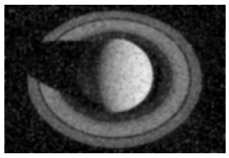
(a) noise1g.png denoised with gaussian filter; original noise error: 692.66, denoised error: 149.91



(b) noise1sp.png denoised with gaussian filter; original noise error: 1982.10, denoised error: 165.32

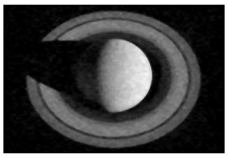


(c) noise2g.png denoised with gaussian filter; original noise error: 3118.10, denoised error: 556.21

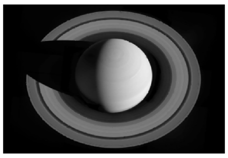


(d) noise2sp.png denoised with gaussian filter; original noise error: 3829.28, denoised error: 349.12

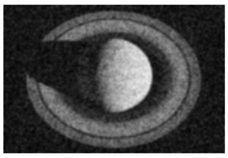
Figure 4: All four noisy image denoised with gaussian filters, each having their own optimal standard deviation σ



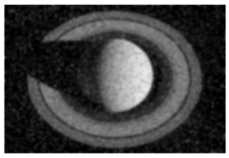
(a) noise1g.png denoised with median filter; original noise error: 692.66, denoised error: 90.11



(b) noise1sp.png denoised with median filter; original noise error: 1982.10, denoised error: 8.78



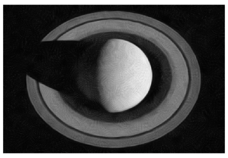
(c) noise2g.png denoised with median filter; original noise error: 3118.10, denoised error: 225.28



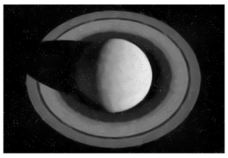
(d) noise2sp.png denoised with gaussian filter; original noise error: 3829.28, denoised error: 18.57

Figure 5: All four noisy image denoised with median filters, each having their own optimal kernal size mxm

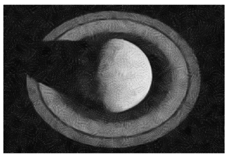
- The reason behind this can be that Non-Local Means considers weighted average of pixels around a noisy pixel(patch) with respect to other patches around its window. That gives a better global view than just considering a local neighboring pixels patch (gaussian filter).
- The reason why it is called non-local is because it does not consider the spatial relationship between pixels, instead, the similarity between different patches and the patch around the target pixel is computed to determine the weight mentioned above (inversely proportional to the distance between its patch and the reference patch of the target pixel).



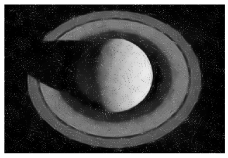
(a) noise1g.png denoised with Non-Local Means filter; original noise error: 692.66, denoised error: 102.8368



(b) noise1sp.png denoised with Non-Local Means filter; original noise error: 1982.10, denoised error: 95.7399



(c) noise2g.png denoised with Non-Local Means filter; original noise error: 3118.10, denoised error: 534.652



(d) noise2sp.png denoised with Non-Local Means filter; original noise error: 3829.28, denoised error: 272.9644

Figure 6: All four noisy image denoised with Non-Local Means filters, each having their bandwidth parameter $\gamma = 0.2$

- MEDIAN FILTERS gives better results for gaussian noised images too.
- Median filter (rectangular kernel) is optimal for reducing random noise in spatial domain (image space). However Median filter is the worst filter for frequency domain, with little ability to separate one band of frequencies from another. Gaussian filter has better performance in frequency domain.
- Gaussian filter uses convolution and is very slow. If we implement Mean filter using recursive formula it will run like very fast.
- **MEAN FILTER IS FAST AND PROBABLY THE BEST SOLUTION IF WE WANT TO REMOVE NOISE FROM IMAGE. IT IS BAD SOLUTION IF YOU WANT TO SEPARATE FREQUENCIES PRESENT IN THE IMAGE.**

2 Problem 2: Texture Synthesis

TEXTURE SYNTHESIS is the process of algorithmically constructing a large digital image from a small digital sample image by taking advantage of its structural content.

The patterns to be synthesized can be **REGULAR** - repeating in fixed spaces in a fixed/ replicated way; the patterns can be **STOCHASTIC** - they look like noise; or they can be **IRREGULAR** - they have no specific recurring structure at all.

2.1 Objective

Our goal for this mini-project is to[5]:

- The output should have the SIZE given by the user.
- The output should be as SIMILAR as possible to the sample.
- The output should NOT HAVE VISIBLE ARTIFACTS such as seams, blocks and misfitting edges.
- The output should not REPEAT, i. e. the same structures in the output image should not appear multiple places.
- The process of texture synthesis should be efficient in COMPUTATION TIME AND IN MEMORY USE.

Now, we look at two methods to achieve our goal, in the next sections.

2.2 (a) Random Tiling

The procedure was simple for random tiling; as there is no logic except to pick up the random tiles from within the input image bounds and to put them on the correct position.

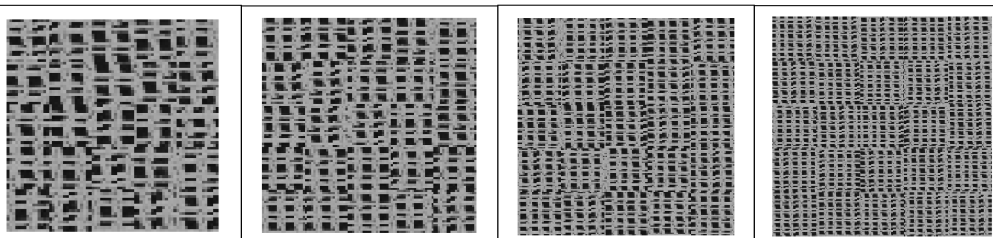
The Matlab code for this is in Appendix B.1 [for the starting point of the program, which calls `synthRandomPatch.m`] and B.2 [for the function `synthRandomPatch` implementation].

Again, we took care of not choosing a random co-ordinate pair in a way that might lead to overrunning the input image bounds and thus, not having the exact same size as the pre-decided `tileSize`.

If that was the case, we simply chose some other random co-ordinate pair which was within the bounds.

2.2.1 Results

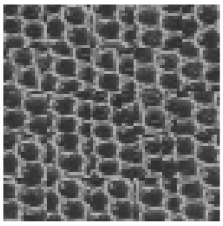
Figures 7, 8 and 9 shows the result of random sampling for four tile sizes - 15, 20, 30, 40.



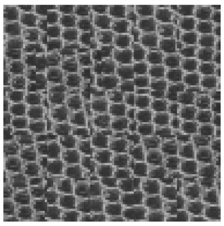
(a) D20.png synthesized with tile size 15	(b) D20.png synthesized with tile size 20	(c) D20.png synthesized with tile size 30	(d) D20.png synthesized with tile size 40

Figure 7: D20.png synthesized with tile sizes - 15, 20, 30, 40. Note that the original sizes of images are $75\text{x}75$, $100\text{x}100$, $150\text{x}150$, $200\text{x}200$; but we are showing them on a same scale for better representation and understanding of how they look side by side

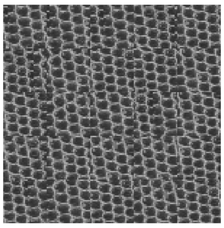
As we can see, only in an extremely rare case, this method will work, when the chosen tiles are always seamless. Even in a fixed pattern, choosing a tile randomly in this way is near to impossible.



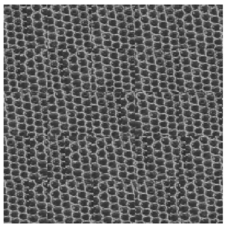
(a) Texture2.bmp synthesized with tile size 15



(b) Texture2.bmp synthesized with tile size 20

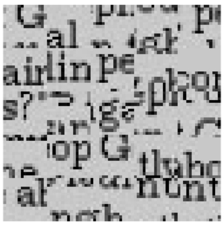


(c) Texture2.bmp synthesized with tile size 30

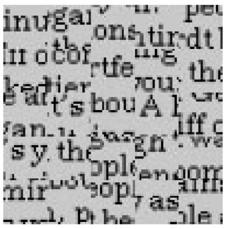


(d) Texture2.bmp synthesized with tile size 40

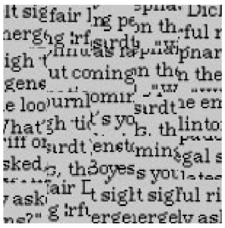
Figure 8: Texture2.bmp synthesized with tile sizes - 15, 20, 30, 40. Note that the original sizes of images are $75x75$, $100x100$, $150x150$, $200x200$; but we are showing them on a same scale for better representation and understanding of how they look side by side



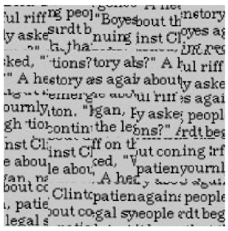
(a) english.jpg synthesized with tile size 15



(b) english.jpg synthesized with tile size 20



(c) english.jpg synthesized with tile size 30



(d) english.jpg synthesized with tile size 40

Figure 9: english.jpg synthesized with tile sizes - 15, 20, 30, 40. Note that the original sizes of images are $75x75$, $100x100$, $150x150$, $200x200$; but we are showing them on a same scale for better representation and understanding of how they look side by side

Therefore, we certainly need a better method which promises a smooth transition from one to the next tile. But random tiling provides us with the basic knowledge to approach the texture synthesis problem.

2.3 (b) Non-parametric Sampling

In Non-parametric Sampling, the texture synthesis process grows a new image outward from AN INITIAL SEED, one pixel at a time.

A Markov random field model is assumed, and the conditional distribution of a pixel given all its neighbors synthesized so far is estimated by querying the sample image and FINDING ALL SIMILAR NEIGHBORHOODS.

The method aims at PRESERVING AS MUCH LOCAL STRUCTURE as possible and produces good results for a wide variety of synthetic and real-world textures.

We want to insert pixel intensities BASED ON EXISTING NEARBY PIXEL values.

Distribution of a value of a pixel is conditioned on its neighbors alone.

The Matlab code for this is in Appendix B.1 [for the starting point of the program, which calls `synthEfrosLeung.m`] and B.3 [for the function `synthEfrosLeung` implementation].

2.3.1 Outputs

We ran our algorithm 4-5 times for each window size and for each image, so we can average the time and decrease the effect of "worse" choice of seed, even if it is just by a very little margin.

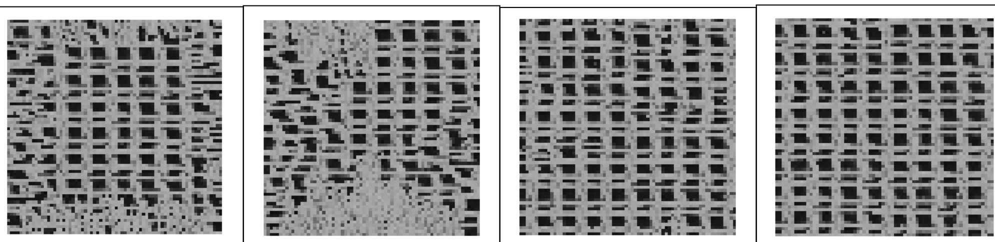
We display the best result [more realistic and reasonable result] out of all the trials.

Figures 10, 11 and 12 shows the result of random sampling for four tile sizes - 5, 7, 11, 15.

2.3.2 Effect of window size on runtime and on synthesis quality

We note down our observation about correlation between window sizes and runtime in Table 4:

We now note down few of our inferences about correlation between window sizes, synthesis quality and runtime:



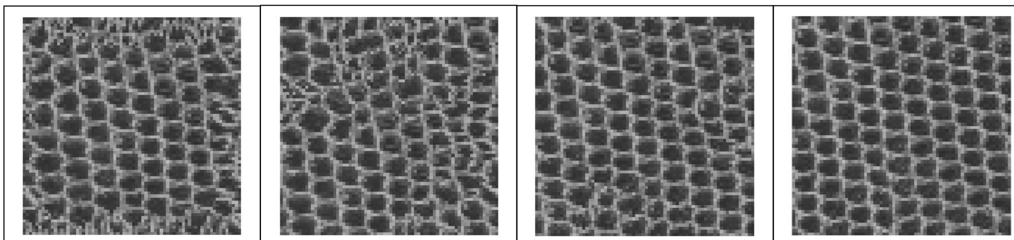
(a) D20.png synthesized with window size 5

(b) D20.png synthesized with window size 7

(c) D20.png synthesized with window size 11

(d) D20.png synthesized with window size 15

Figure 10: D20.png synthesized with window sizes - 5, 7, 11, 15. Note that the sizes of all output images are 70x70



(a) Texture2.bmp synthesized with window size 5

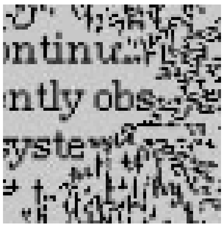
(b) Texture2.bmp synthesized with window size 7

(c) Texture2.bmp synthesized with window size 11

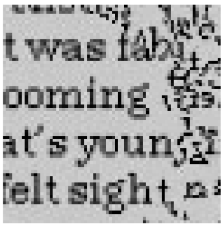
(d) Texture2.bmp synthesized with window size 15

Figure 11: Texture2.bmp synthesized with window sizes - 5, 7, 11, 15. Note that the sizes of all output images are 70x70

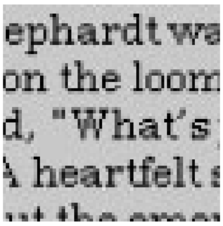
- We can observe from the Table 4 that as the size of the window increases, the runtime increases as the patch sizes will be bigger and thus, the distance metric computations also takes longer.
- But, since the bigger window size means more neighborhood coverage, the windows sizes of 11 and 15 gives a lot more realistic results then the windows of 5 and 7.
- That is because when we look at a small patch, there might be patterns which match with that tiny patch but globally, they might have no correlation at all.
- That makes a particular pixel with tiny patch get "wrong" value, which



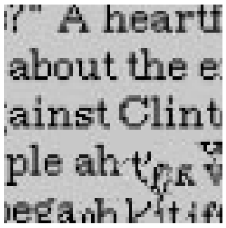
(a) english.jpg synthesized with window size 5



(b) english.jpg synthesized with window size 7



(c) english.jpg synthesized with window size 11



(d) english.jpg synthesized with window size 15

Figure 12: english.jpg synthesized with window sizes - 5, 7, 11, 15. Note that the sizes of all output images are 70x70

just snowballs upon further synthesis.

- IN CONCLUSION , larger window sizes will give better quality of texture synthesis since they have better "global coverage", but it also takes more runtime for the same reason [more computation for distance metric because more pixes in the patch]

Image Name	Window Size	Average Running Time for 5 Runs (in seconds)
D20.png Input Size: 49 x 53 Output Size: 70 x 70	5	18.762
	7	28.449
	11	49.166
	15	73.546
Texture2.bmp Input Size: 58 x 56 Output Size: 70 x 70	5	19.569
	7	34.032
	11	67.694
	15	81.865
english.jpg Input Size: 151 x 156 Output Size: 70 x 70	5	137.006
	7	207.111
	11	489.050
	15	786.656

Table 4: Relation between window size, input image size and the runtime

3 Extensions for Extra Credit

3.1 Extension 1: Block Matching 3D (BM3D)

BLOCK-MATCHING AND 3D FILTERING (BM3D) is a 3-D block-matching algorithm used primarily for noise reduction in images.[6]

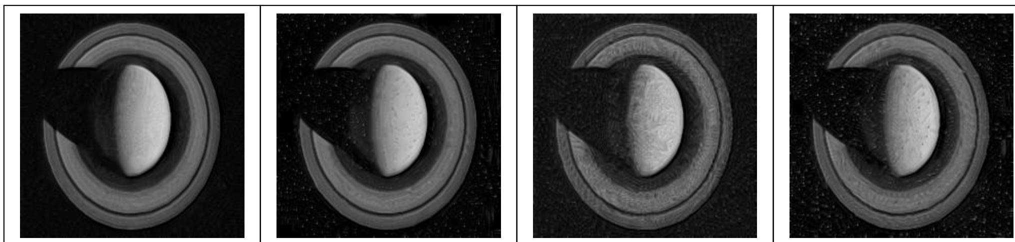
- This strategy is based on an enhanced SPARSE REPRESENTATION in transform-domain. The enhancement of the sparsity is achieved by grouping similar 2D image fragments (e.g. blocks) into 3D data arrays which we call "groups".
- Collaborative filtering is a special procedure developed to deal with these 3D groups. It is realized by using the three successive steps: 3D TRANSFORMATION of 3D group, SHRINKAGE of transform spectrum, and INVERSE 3D transformation.
- The result is a 3D estimate that consists of the JOINTLY FILTERED grouped image blocks.
- By attenuating the noise, the collaborative filtering reveals even the finest details shared by grouped blocks and at the same time it preserves the essential UNIQUE FEATURES of each individual block.
- The filtered blocks are then returned to their original positions.
- Because these BLOCKS ARE OVERLAPPING, for each pixel we obtain many different estimates which need to be combined.
- AGGREGATION is a particular averaging procedure which is exploited to take advantage of this redundancy.
- A significant improvement is obtained by a specially developed collaborative WIENER FILTERING.
- The results presented here demonstrate that the developed methods achieve state-of-the-art denoising performance in terms of both peak signal-to-noise ratio and subjective visual quality.

The PSNR results are shown in 5 and the output images are shown in Figures 13.

We now compare BM3D with NLM from Section 1.3:

Image Name	Optimal Standard Deviation σ	Peak Signal-to-Noise Ratio PSNR	SSD
saturn-noise1g.png	0.5	21.418	203.346
saturn-noise1sp.png	0.3	16.852	370.128
saturn-noise2g.png	0.1	14.8844	482.735
saturn-noise2sp.png	0.1	13.9922	568.804

Table 5: Optimal values of standard deviation for which we get the most similar denoised image, in comparison to the original image; using Block-Matching 3D Filtering



(a) noise1g.png denoised with BM3D filter; original noise error: 692.66, denoised error: 21.418

(b) noise1sp.png denoised with BM3D filter; original noise error: 1982.10, denoised error: 16.852

(c) noise2g.png denoised with BM3D filter; original noise error: 3118.10, denoised error: 14.8844

(d) noise2sp.png denoised with BM3D filter; original noise error: 3829.28, denoised error: 13.9922

Figure 13: All four noisy image denoised with BM3D filters, each having their own optimal standard deviation σ

- Our NLM representation still outperforms BM3D implementation of Kostadin Dabov [6], see Tables 3 and 5
- BM3D takes the concept of non-local means (NLM) in the sense that it also attempts denoising based on finding similar patches within a given window.
- BM3D use a reduced set of adjacent pixels, but in comparison to NLM, it also requires additional, computationally intensive processing steps and continue to suffer from visible artifacts on sharp edges and in smooth regions.
- Hence, the sub-optimal BM3D results as opposed to quite satisfactory results from NLM.

- BM3D approach generates a slightly more noisy image, but with better preserved finer details. And without introducing grid artifacts that are on the other hand visible.

BM3D adopts a NLM search selection strategy, but it also suffers from the noise-to-noise matching problem. That's why we get better results with 3 out 4 noisy images, with BM3D performing only better at large gaussian noise.
Although, there exists better BM3D implementation then the one we used here; which outperforms NLM by a large margin. E.g., BM4D, VBM3D, VBM4D,RF3D, BM3D-CFA, BM3D-SAPCA

3.2 Extension 2: Image Quilting

IMAGE QUILTING proposes a way to minimize these by picking tiles that align well along the boundary (like solving a jigsaw puzzle). The basic idea is to pick a tile that has small SSD along the overlapping boundary with the tiles generated so far.

Here we show the results of Image Quilting techniques described by Efros and Freeman.

We select the first seed random tile as shown below:

```
%% Get random tile co-ordinate pairs for the seed patch
x = floor(1 + rand(1,1) * (w - tileSize -1));
y = floor(1 + rand(1,1) * (h - tileSize -1));

% Get the seed patch
patch = im(x:x+tileSize-1, y:y+tileSize-1);
% Copy the random tile seed patch from source image to the top-left corner
im_patch(1:tileSize, 1:tileSize) = patch;
```

We select the overlapping factor as shown below, if it in floating point, we round it off to a smallest integer greater than the float:

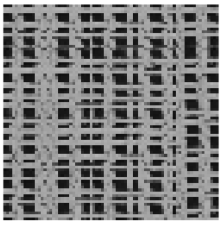
```
%% size of the region of overlap between tiles - typically 1/6
overlap = ceil(1/6 * tileSize);
```

We select the overlapping region for vertical and horizontal parts as shown below:

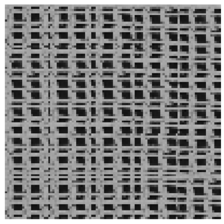
```
% Upper Horizontal overlapping section
A = temp_patch(1:tileSize, 1:overlap);
% Left Vertical overlapping section
B = temp_patch(1:overlap, 1:tileSize);
```

The MATLAB CODE for this is in Appendix B.1 [for the starting point of the program, which calls `synthEfrosLeung.m`] and C.1 [for the function `synthImageQuilting` implementation].

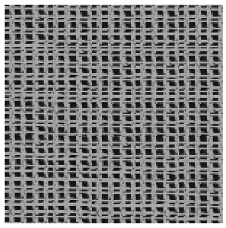
The results are shown in Figures 14, 15 and 16.



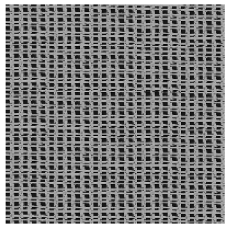
(a) D20.png synthesized with tile size 15



(b) D20.png synthesized with tile size 20

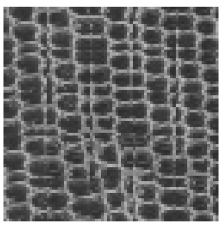


(c) D20.png synthesized with tile size 30

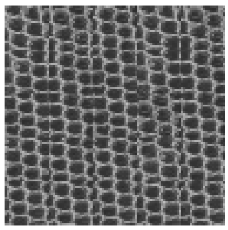


(d) D20.png synthesized with tile size 40

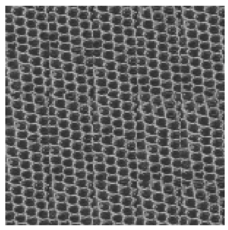
Figure 14: D20.png quilted with tile sizes - 15, 20, 30, 40. Note that the original sizes of images are 75x75, 100x100, 150x150, 200x200; but we are showing them on a same scale for better representation and understanding of how they look side by side



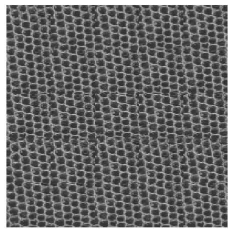
(a) Texture2.bmp synthesized with tile size 15



(b) Texture2.bmp synthesized with tile size 20



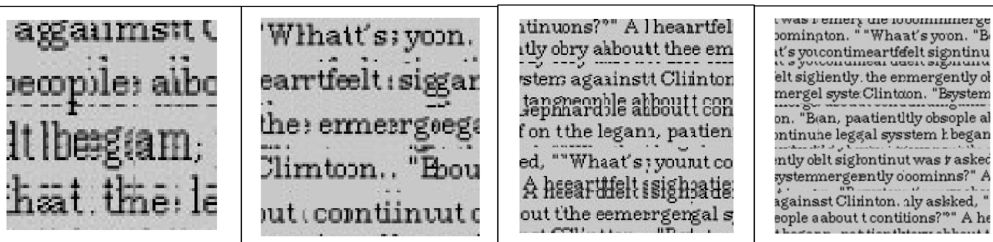
(c) Texture2.bmp synthesized with tile size 30



(d) Texture2.bmp synthesized with tile size 40

Figure 15: Texture2.bmp quilted with tile sizes - 15, 20, 30, 40. Note that the original sizes of images are 75x75, 100x100, 150x150, 200x200; but we are showing them on a same scale for better representation and understanding of how they look side by side

As we can see, the text image `english.jpg` still needs to be adjusted to proper window size, in order to address the doubling/replicating effect.



(a) english.jpg synthesized with tile size 15

(b) english.jpg synthesized with tile size 20

(c) english.jpg synthesized with tile size 30

(d) english.jpg synthesized with tile size 40

Figure 16: english.jpg quilted with tile sizes - 15, 20, 30, 40. Note that the original sizes of images are 75x75, 100x100, 150x150, 200x200; but we are showing them on a same scale for better representation and understanding of how they look side by side

Notwithstanding its plainness, this technique works astoundingly good when tried for texture synthesis, fabricating results that are equal or better than the Efros & Leung family of algorithms but with upgraded strength (less chance of “growing garbage”) and at a snippet of the computational price.

References

[1] Subhransu Maji, *Linear Filtering* https://www.dropbox.com/s/tfsmuiunffyu7ml/lec09\%2B10_linear_filtering.pdf?dl=0

[2] *Noise Reduction* https://en.wikipedia.org/wiki/Noise_reduction#In_images

[3] Richard Szeliski, *Computer Vision: Algorithms and Applications* ISBN: 978-1-848-82934-3

[4] Antoni Buades, Bartomeu Coll, Jean-Michel Morel, *A non-local algorithm for image denoising* https://www.iro.umontreal.ca/~mignotte/IFT6150/Articles/Buades-NonLocal.pdf

[5] *Texture synthesis* https://en.wikipedia.org/wiki/Texture_synthesis

[6] *Block-matching and 3D filtering (BM3D) algorithm and its extensions* http://www.cs.tut.fi/~foi/GCF-BM3D/index.html#ref_problems

A

Matlab code for Problem 1: Image Denoising

A.1 Implementation of `evalDenosing.m`

```matlab
% Entry code for evaluating demosaicing algorithms
% The code loops over all images and methods, computes the error and
% displays them in a table.
%
%
% This code is part of:
%
%   CMPSCI 670: Computer Vision
%   University of Massachusetts, Amherst
%   Instructor: Subhransu Maji
%
close all;
clc;
% Load images
im = im2double(imread('../data/denoising/saturn.png'));
noise1 = im2double(imread('../data/denoising/saturn-noise1g.png'));
noise2 = im2double(imread('../data/denoising/saturn-noise1sp.png'));
%noise1 = im2double(imread('../data/denoising/saturn-noise2g.png'));
%noise2 = im2double(imread('../data/denoising/saturn-noise2sp.png'));

% Compute errors
error1 = sum(sum((im - noise1).^2));
error2 = sum(sum((im - noise2).^2));
fprintf('Input, Errors: %.2f %.2f\n', error1, error2)

% Display the images
figure(1);
```

```matlab
subplot(1,3,1); imshow(im); title('Input');
subplot(1,3,2); imshow(noise1); title(sprintf('SE %.2f', error1));
subplot(1,3,3); imshow(noise2); title(sprintf('SE %.2f', error2));

%% Denoising algorithm (Gaussian filtering)

count = 1;
% Experiment with different values of standard deviation
for i = 1.85:0.01:1.95
    % Give noisy image and std dev as parameters to gaussian filter to get a denoised image
    filtered3 = imgaussfilt(noise1, i);
    filtered4 = imgaussfilt(noise2, i);

    % Compute errors - SSD
    error3 = sum(sum((im - filtered3).^2));
    error4 = sum(sum((im - filtered4).^2));

    % Print results
    fprintf('Gaussian Filter Simga %.2f- Input, Errors: %.2f %.2f\n', i, error3, error4)

    %Display results
    figure(2);
    subplot(5, 8, count); imshow(filtered3); title(sprintf('SE %.2f', error3));
    figure(3);
    subplot(5, 8, count); imshow(filtered4); title(sprintf('SE %.2f', error4));
    count = count + 1;


%% Denoising algorithm (Median filtering)

% To find the std dev where there is least SSD
min5 = Inf; index_x5 = 0; index_y5 = 0;
min6 = Inf; index_x6 = 0; index_y6 = 0;

```

```matlab
% Experiment with different filter kernel dimensions
for i = 1:10
    for j = 1:10

        % Convolute noisy image with i x j dimension median filter kernel
        filtered5 = medfilt2(noise1, [i, j]);
        filtered6 = medfilt2(noise2, [i, j]);

        % Compute errors - SSD
        error5 = sum(sum((im - filtered5).^2));
        error6 = sum(sum((im - filtered6).^2));
        fprintf('Median Filter Neighborhood %d %d-Input, Errors: %.2f %.2f\n', i, j, error5, error6)

        % Save the filter dimensions of the kernel with the least error
        if error5 < min5
            min5 = error5;
            index_x5 = i;
            index_y5 = j;
        end
        if error6 < min6
            min6 = error6;
            index_x6 = i;
            index_y6 = j;
        end
    end
end

% Print the results
fprintf('Median Filter Neighborhood %d %d- image1, Error: %.2f\n', index_x5, index_y5, min5)
fprintf('Median Filter Neighborhood %d %d- image2, Error: %.2f\n', index_x6, index_y6, min6)

%% Denoising alogirthm (Non-local means)

img_n = noise1;      % Noisy Image
halfPatchSize = 2;  % Patch Size (halved)
```

```
windowHalfSearchSize = 10;  % Window Size (halved)
gamma = 0.01;         % Bandwidth parameter
[w, h] = size(img_n);  % Size of the noisy image

img_f = img_n;        % Denoised Image

% Go through whole image
for i = 1:h
    for j = 1:w

        % Get the patch with the pixel (i, j) as its center and halfPatchSize as its radius
        px = img_n(max(1, j - halfPatchSize):min(w, j + halfPatchSize), max(1, i - halfPatchSize):min(h, i + halfPatchSize));

        % nominator of the denoised pixel equation
        sum_up = 0.0;
        % denominator of the denoised pixel equation
        sum_dn = 0.0;

        % For each pixel on vertical edges of the window specified by its half size
        for k = max(1, j - windowHalfSearchSize):min(w, j + windowHalfSearchSize)

            % For left edge of the window
            if 1 <= i - windowHalfSearchSize
                % Get a patch around that particular pixel on the left edge of the window
                py = img_n(max(1, k - halfPatchSize):min(w, k + halfPatchSize), max(1, i - windowHalfSearchSize - halfPatchSize):min(h, i - windowHalfSearchSize + halfPatchSize));
                % Find out minimum SSD
                if size(px) == size(py)
                    s = (px - py).^2;
                    s = s(:);
```

```
                d = sqrt(sum(s));
                d1 = exp(-1 * gamma * d);
                sum_dn = sum_dn + d1;
                sum_up = sum_up + d1 * img_n(j, i);
            end
        end

        % For right edge of the window
        if h >= i + windowHalfSearchSize
            % Get a patch around that particular pixel on the right edge of the window
            py = img_n(max(1, k - halfPatchSize):min(w, k + halfPatchSize), max(1, i + windowHalfSearchSize - halfPatchSize):min(h, i + windowHalfSearchSize + halfPatchSize));
            % Find out minimum SSD
            if size(px) == size(py)
                s = (px - py).^2;
                s = s(:);
                d = sqrt(sum(s));
                d1 = exp(-1 * gamma * d);
                sum_dn = sum_dn + d1;
                sum_up = sum_up + d1 * img_n(j, i);
            end
        end
    end

    % For each pixel on horizontal edges of the window specified by its half size
    for k = max(1, i - windowHalfSearchSize):min(h, i + windowHalfSearchSize)
        % For top edge of the window
        if 1 <= j - windowHalfSearchSize
            % Get a patch around that particular pixel on the upper edge of the window
```

```
            py = img_n(max(1, j - windowHalfSearchSize - halfPatchSize):min(w, j - windowHalfSearchSize + halfPatchSize), max(1, k - halfPatchSize):min(h, k + halfPatchSize));
            % Find out minimum SSD
            if size(px) == size(py)
                s = (px - py).^2;
                s = s(:);
                d = sqrt(sum(s));
                d1 = exp(-1 * gamma * d);
                sum_dn = sum_dn + d1;
                sum_up = sum_up + d1 * img_n(j, i);
            end
        end

        % For bottom edge of the window
        if w >= j + windowHalfSearchSize
            % Get a patch around that particular pixel on the lower edge of the window
            py = img_n(max(1, j + windowHalfSearchSize - halfPatchSize):min(w, j + windowHalfSearchSize + halfPatchSize), max(1, k - halfPatchSize):min(h, k + halfPatchSize));
            % Find out minimum SSD
            if size(px) == size(py)
                s = (px - py).^2;
                s = s(:);
                d = sqrt(sum(s));
                d1 = exp(-1 * gamma * d);
                sum_dn = sum_dn + d1;
                sum_up = sum_up + d1 * img_n(j, i);
            end
```

```
            end
        end

        % Get the new value for the pixel (i, j)
        if sum_dn ~= 0
            img_f(j, i) = sum_up/sum_dn;
            disp(img_f(j,i))
        end

    end
end

% Caluclate Error - SSD
error7 = sum(sum((im - img_f).^2));
% Print the error
fprintf('Non-Linear Means Errors: %.6f\n', error7)
% Show the results
figure;
subplot(121); imshow(im);
subplot(122); imshow(img_f);
```

A.2 Implementation of Non-Local Means Filters

NOTE: this is included in the `evalDenosing.m` shown in A.1 too, the separate copy is just for clarification.

```
%% Denoising alogirthm (Non-local means)

img_n = noise1;        % Noisy Image
halfPatchSize = 2;     % Patch Size (halved)
windowHalfSearchSize = 10;  % Window Size (halved)
gamma = 0.01;          % Bandwidth parameter
[w, h] = size(img_n);  % Size of the noisy image

img_f = img_n;         % Denoised Image

% Go through whole image
for i = 1:h
    for j = 1:w

        % Get the patch with the pixel (i, j) as its center and halfPatchSize as its radius
```

```
        px = img_n(max(1, j - halfPatchSize):min(w, j + halfPatchSize), max(1, i - halfPatchSize):min(h, i + halfPatchSize));

        % nominator of the denoised pixel equation
        sum_up = 0.0;
        % denominator of the denoised pixel equation
        sum_dn = 0.0;

        % For each pixel on vertical edges of the window specified by its half size
        for k = max(1, j - windowHalfSearchSize):min(w, j + windowHalfSearchSize)

            % For left edge of the window
            if 1 <= i - windowHalfSearchSize
                % Get a patch around that particular pixel on the left edge of the window
                py = img_n(max(1, k - halfPatchSize):min(w, k + halfPatchSize), max(1, i - windowHalfSearchSize - halfPatchSize):min(h, i - windowHalfSearchSize + halfPatchSize));
                % Find out minimum SSD
                if size(px) == size(py)
                    s = (px - py).^2;
                    s = s(:);
                    d = sqrt(sum(s));
                    d1 = exp(-1 * gamma * d);
                    sum_dn = sum_dn + d1;
                    sum_up = sum_up + d1 * img_n(j, i);
                end
            end

            % For right edge of the window
            if h >= i + windowHalfSearchSize
                % Get a patch around that particular pixel on the right edge of the
```

```
                window
            py = img_n(max(1, k - halfPatchSize):min(w, k + halfPatchSize), max(1, i + windowHalfSearchSize - halfPatchSize):min(h, i + windowHalfSearchSize + halfPatchSize));
            % Find out minimum SSD
            if size(px) == size(py)
                s = (px - py).^2;
                s = s(:);
                d = sqrt(sum(s));
                d1 = exp(-1 * gamma * d);
                sum_dn = sum_dn + d1;
                sum_up = sum_up + d1 * img_n(j, i);
            end
        end
    end

    % For each pixel on horizontal edges of the window specified by its half size
    for k = max(1, i - windowHalfSearchSize):min(h, i + windowHalfSearchSize)
        % For top edge of the window
        if 1 <= j - windowHalfSearchSize
            % Get a patch around that particular pixel on the upper edge of the window
            py = img_n(max(1, j - windowHalfSearchSize - halfPatchSize):min(w, j - windowHalfSearchSize + halfPatchSize), max(1, k - halfPatchSize):min(h, k + halfPatchSize));
            % Find out minimum SSD
            if size(px) == size(py)
                s = (px - py).^2;
                s = s(:);
                d = sqrt(sum(s));
```

```
                    d1 = exp(-1 * gamma * d);
                    sum_dn = sum_dn + d1;
                    sum_up = sum_up + d1 * img_n(j, i);
                end
            end

            % For bottom edge of the window
            if w >= j + windowHalfSearchSize
                % Get a patch around that particular pixel on the lower edge of the window
                py = img_n(max(1, j + windowHalfSearchSize - halfPatchSize):min(w, j + windowHalfSearchSize + halfPatchSize), max(1, k - halfPatchSize):min(h, k + halfPatchSize));
                % Find out minimum SSD
                if size(px) == size(py)
                    s = (px - py).^2;
                    s = s(:);
                    d = sqrt(sum(s));
                    d1 = exp(-1 * gamma * d);
                    sum_dn = sum_dn + d1;
                    sum_up = sum_up + d1 * img_n(j, i);
                end
            end
        end

        % Get the new value for the pixel (i, j)
        if sum_dn ~= 0
            img_f(j, i) = sum_up/sum_dn;
            disp(img_f(j,i))
        end

    end
end
```

```
% Caluclate Error - SSD
error7 = sum(sum((im - img_f).^2));
% Print the error
fprintf('Non-Linear Means Errors: %.6f\n', error7)
% Show the results
figure;
subplot(121); imshow(im);
subplot(122); imshow(img_f);
```

B
Matlab code for Problem 2: Texture Synthesis

B.1 Implementation of `evalTextureSynth.m`

```
close;

% Load images
% im = im2double(imread('../data/texture/D20.png'));
% im = im2double(imread('../data/texture/Texture2.bmp'));
 im = im2double(imread('../data/texture/english.jpg'));

figure(1); imshow(im);


%% Random patches
tileSize = 40;  % specify block sizes - 15 20 30 40
numTiles = 5;
outSize = numTiles * tileSize; % calculate output image size
% save the random-patch output and record run-times
im_patch = synthRandomPatch(im, tileSize, numTiles, outSize);
figure;
imshow(im_patch);

%% Non-parametric Texture Synthesis using Efros & Leung algorithm
winsize = 11;  % specify window size (5, 7, 11, 15)
outSize = 70;  % specify size of the output image to be synthesized (square for simplicity)
% save the synthesized image and record the run-times
im_synth = synthEfrosLeung(im, winsize, outSize);
figure(3);
imshow(im_synth);
```

```
%% Image Quilting
tileSize = 40;  % specify block sizes - 15 20 30 40
numTiles = 5;
outSize = numTiles * tileSize; % calculate output image size
% save the random-patch output and record run-times
im_patch = synthImageQuilting(im, tileSize, numTiles, outSize);
figure;
imshow(im_patch);
```

B.2 Implementation of `synthRandomPatch.m`

```
function im_patch = synthRandomPatch(im, tileSize, numTiles, outSize)

    % Get size of original image whose texture we want to synthesize
    [w, h] = size(im);

    % Initialize output/ result to the size specified
    im_patch = zeros(outSize);

    % Create a synthesized image block-by-block or tile-by-tile
    for i = 1:numTiles
        for j = 1:numTiles

            % Randomly selects an x-axis as a top-left corner of a random tile
            x = floor(1 + rand(1,1) * (w - tileSize -1));
            % Randomly selects an y-axis as a top-left corner of a random tile
            y = floor(1 + rand(1,1) * (h - tileSize -1));

            % Cut out the copy of the tile
            patch = im(x:x+tileSize-1, y:y+tileSize-1);
            % Paste it in the assigned top-left corner
```

```
                of the output image
            % [Top to Bottom; Left to Right]
            im_patch((i-1)*tileSize+1:(i-1)*tileSize+
               tileSize, (j-1)*tileSize+1:(j-1)*
               tileSize+tileSize) = patch;
        end
    end
end
```

B.3 Implementation of `synthEfrosLeung.m`

```
function im_synth = synthEfrosLeung(im, winsize,
   outSize)
    %% Initializing output/synthesized image with all
       -1 (not with 0s as they might depict an actual
       value)
    im_synth = zeros(outSize)-1;
    % Maximum number of pixels which might need to get
        their values interpolated
    n = outSize * outSize;

    % Get half window size
    halfWinSize = floor(winsize/2);
    % Error Threshold for finding best matches to
       interpolate a pixel's value
    error_thresold = 0.1;

    %% Get input image size
    [w, h, c] = size(im);

    % Randomly choose a pixel value co-ordinate pair;
       "-3" shows we must choose the pixel in a way
       that a 3 x 3 patch with this pixel at top-left
       is possible
    x = floor(1 + rand(1,1) * (w - 3 - 1));
    y = floor(1 + rand(1,1) * (h - 3 - 1));
    % A 3x3 SEED PATCH
    patch = im(x:x+3-1, y:y+3-1);

    % Copy the seed patch to the center of the output/
       synthesized image
```

```
im_synth(outSize/2-1:outSize/2+1, outSize/2-1:outSize/2+1) = patch;

%% Growing the borders of the filled pixels in the output image
for k = 1:outSize*outSize
    %% list of pixel positions in the output image that are unfilled, but contain filled pixels in their neighbourhood
    pixelList = zeros(n, 3);
    count = 1;

    % For all pixels of the output/synthesized image
    for i = 1:outSize
        for j = 1:outSize
            flag = 0;

            % For any pixel whose value has yet to be calculated
            if im_synth(i, j) == -1
                % Add it to the list but...
                pixelList(count, 1) = i;
                pixelList(count, 2) = j;

                % Identify how many of its 8 neighbors have their values interpolated already
                % Add the # neighbors_having_valid_values to the pixelList frequency counter
                if i + 1 <= outSize && im_synth(i+1, j) ~= -1
                    flag = 1;
                    pixelList(count, 3) = pixelList(count, 3) + 1;
                end
                if i - 1 >= 1 && im_synth(i-1, j) ~= -1
                    flag = 1;
                    pixelList(count, 3) = pixelList
```

```
(count, 3) + 1;
end
if j + 1 <= outSize && im_synth(i, j+1) ~= -1
    flag = 1;
    pixelList(count, 3) = pixelList(count, 3) + 1;
end
if j - 1 >= 1 && im_synth(i, j-1) ~= -1
    flag = 1;
    pixelList(count, 3) = pixelList(count, 3) + 1;
end
if i + 1 <= outSize && j - 1 >= 1 && im_synth(i+1, j-1) ~= -1
    flag = 1;
    pixelList(count, 3) = pixelList(count, 3) + 1;
end
if i - 1 >= 1 && j - 1 >= 1 && im_synth(i-1, j-1) ~= -1
    flag = 1;
    pixelList(count, 3) = pixelList(count, 3) + 1;
end
if i + 1 <= outSize && j + 1 <= outSize && im_synth(i+1, j+1) ~= -1
    flag = 1;
    pixelList(count, 3) = pixelList(count, 3) + 1;
end
if i - 1 >= 1 && j + 1 <= outSize && im_synth(i-1, j+1) ~= -1
    flag = 1;
    pixelList(count, 3) = pixelList(count, 3) + 1;
end

% If the output pixel has at least
```

```
one neighbor having valid value, only then that pixel would count
                if flag == 1
                    count = count + 1;
                end
            end
        end
    end

    %% Now we have a list of all the pixels whose values we will interpolate in this pass
    pixelList = pixelList(1:count-1,:);
    % Sort the list by the number of filled neighboorhood pixels
    pixelList = sortrows(pixelList, 3, 'descend');

    %% For each pixel in pixelList
    for p = 1:size(pixelList, 1)
        % Initialize SSD to maximum integer
        ssd_min = Inf;

        %% Initialize a validMask of windowSize x windowSize
        validMask = im_synth(max(1, pixelList(p, 1) - halfWinSize): min(outSize, pixelList(p, 1) + halfWinSize), max(1, pixelList(p, 2) - halfWinSize): min(outSize, pixelList(p, 2) + halfWinSize));
        % 1s at the neighboorhood positions of that pixel that contains filled pixels
        validMask(validMask ~= -1) = 1;
        % 0s for unfilled positions
        validMask(validMask == -1) = 0;

        % Get a patch of windowSize x windowSize around the pixel whose value we need to find
        patch1 = im_synth(max(1, pixelList(p, 1) - halfWinSize): min(outSize, pixelList(p
```

```
, 1) + halfWinSize), max(1, pixelList(p, 2) - halfWinSize): min(outSize, pixelList(p, 2) + halfWinSize)).*validMask;

    %% For every patch in the input image, compare it with the patch we got for the pixel we want to compute
    for x = 1:w-winsize
        for y = 1:h-winsize
            if size(im(x:x+winsize-1,y:y+winsize-1)) == size(validMask)
                % Apply Valid Mask
                patch2 = im(x:x+winsize-1,y:y+winsize-1).*validMask;
                if size(patch1) == size(patch2)
                    % Find the minimum SSD
                    dist = (patch1 - patch2);
                    ssd = sum(dist(:).^2);
                end
                % Find the pixels in the input image have a similar neighbourhood w.r.t. the filled neighbours of the currently unfilled pixel in the output image
                if ssd < ssd_min
                    % Get minimum SSD
                    ssd_min = ssd;
                    patch3 = im(x:x+winsize-1, y:y+winsize-1);
                end
            end
        end
    end

    %% Get the best matches
    bestMatches = zeros(w*h, 1);
    count1 = 1;
    for x = 1:w
```

```
                for y = 1:h
                    pix = im(x, y);
                    if ssd_min*(1+error_thresold) >= pix
                        bestMatches(count1, 1) = pix;
                        count1 = count1 + 1;
                    end
                end
            end

            %% Randomly pick an input image pixel from BestMatches and paste its pixel value into the current unfilled output pixel location
            im_synth(pixelList(p, 1), pixelList(p, 2)) = bestMatches(randperm(size(bestMatches, 1), 1), 1);

        end
    end
end
```

C Matlab code for Extension for Extra Credit

C.1 Implementation of `synthImageQuilting.m`

```
function im_patch = synthImageQuilting(im, tileSize, numTiles, outSize)
    %% Get the dimensions of the input image
    [w, h, c] = size(im);

    %% Initialize the output image with all 0s
    im_patch = zeros(outSize);

    %% Get random tile co-ordinate pairs for the seed patch
    x = floor(1 + rand(1,1) * (w - tileSize -1));
    y = floor(1 + rand(1,1) * (h - tileSize -1));

    % Get the seed patch
    patch = im(x:x+tileSize-1, y:y+tileSize-1);
    % Copy the random tile seed patch from source image to the top-left corner
    im_patch(1:tileSize, 1:tileSize) = patch;

    %% size of the region of overlap between tiles - typically 1/6
    overlap = ceil(1/6 * tileSize);

    %% For each tile of the output/synthesized image
    for i = 1:numTiles
        for j = 1:numTiles

            % Do nothing for the first tile as it is already set by the random patch
            if (i == 1 && j == 1)
                continue;
            end

            ssd_C_min = inf;
            ssd_D_min = inf;
```

```
            %% SSD (sum of squared difference) between the neighbors of the output image tile position and each tile position in the source image, considering the region of overlap
            for m = 1 : w - tileSize
                for n = 1 :h - tileSize
                    temp_patch = im(m:m+tileSize-1, n:n+tileSize-1);

                    % Upper Horizontal overlapping section
                    A = temp_patch(1:tileSize , 1:overlap);
                    % Left Vertical overlapping section
                    B = temp_patch(1:overlap, 1:tileSize);

                    if (i ~= 1 && j ~= 1)
                        C = im_patch((i-1)*tileSize+1:(i-1)*tileSize+tileSize , (j-1)*tileSize-overlap:(j-1)*tileSize-1);
                        diff = A - C;
                        ssd_C = sum(sum(diff(:).^2));
                        D = im_patch((i-1)*tileSize-overlap:(i-1)*tileSize-1, (j-1)*tileSize+1:(j-1)*tileSize+tileSize);
                        diff = B - D;
                        ssd_D = sum(sum(diff(:).^2));
                        if ssd_C + ssd_D < ssd_C_min + ssd_D_min
                            patch = temp_patch;
                            ssd_C_min = ssd_C;
                            ssd_D_min = ssd_D;
                        end
                    elseif(i ~= 1)
                        D = im_patch((i-1)*tileSize-
```

```
overlap:(i-1)*tileSize-1, (j-1)*tileSize+1:(j-1)*tileSize+tileSize);
                        diff = B - D;
                        ssd_D = sum(sum(diff(:).^2));

                        if ssd_D < ssd_D_min
                            patch = temp_patch;
                            ssd_D_min = ssd_D;
                        end
                    elseif(j ~= 1)
                        C = im_patch((i-1)*tileSize+1:(i-1)*tileSize+tileSize, (j-1)*tileSize-overlap:(j-1)*tileSize-1);
                        diff = A - C;
                        ssd_C = sum(sum(diff(:).^2));

                        if ssd_C < ssd_C_min
                            patch = temp_patch;
                            ssd_C_min = ssd_C;
                        end
                    end

                end
            end

            %% Copy the tile in source image with lowest SSD (minSSD) with respect to the current tile in output image, to the output image
            im_patch((i-1)*tileSize+1:(i-1)*tileSize+tileSize, (j-1)*tileSize+1:(j-1)*tileSize+tileSize) = patch;

            %% To the next un-filled position in the output image (left-to-right, top-to-bottom)
        end
    end

```

```
end
```