# Mini-project 3

CMPSCI 670, Fall 2019, UMass Amherst

Instructor: Subhransu Maji

TAs: Aruni RoyChowdhury, Archan Ray

3

## Guidelines

**Submission.** Submit a *single pdf document* via moodle that includes your solutions, figures and code. The latex source file for the homework is provided which you can modify to produce your report. You are welcome to use other typesetting software as long as the final output is a pdf. For readability you may attach the code printouts at the end of the solutions within the same pdf. Note that we will not run your code. Similarly figures should be included in a manner which makes it easy to compare various approaches. Poorly written or formatted reports will make it harder for the TA to evaluate it and may lead to a partial deduction of credit.

**Late policy.** You could have 24 hours late submission with a 50% mark down. Late submission beyond 24 hours will not be given *any* credits.

**Plagiarism.** We might reuse problem set questions from previous years, covered by papers and webpages, we expect the students not to copy, refer to, or look at the solutions in preparing their answers. We expect students to want to learn and not google for answers.

**Collaboration.** The homework must be done individually, except where otherwise noted in the assignments. 'Individually' means each student must hand in their own answers, and each student must write their own code in the programming part of the assignment. It is acceptable, however, for students to collaborate in figuring out answers and helping each other solve the problems. We will be assuming that you will be taking the responsibility to make sure you personally understand the solution to any work arising from such a collaboration.

**Using other programming languages.** We made the starter code available in Python and Matlab. You are free to use other languages such as Octave or Julia with the caveat that we won't be able to answer or debug non Matlab/Python questions.

**Python requirements.** We will be using Python 2.7. The Python starter code requires `scipy`, `numpy` (at least v1.12), and `scikit-image`. If you are not familiar with installing those libraries through some package manager (like `pip`), the easiest way of using them is installing Anaconda.

# 1 Image denoising [35 points]

In this part of the homework you will evaluate two simple denoising algorithms, namely Gaussian filtering and median filtering. In addition you will also implement a more advanced algorithm based on non-local means. The starter code for this part is in `evalDenoising`. The test images are provided in the `data/denoising` folder. There are five images:

- `saturn.png` – noise-free image for reference
- `ssaturn-noise1g.png`, `saturn-noise2g.png` – two images with i.i.d. Gaussian noise
- `ssaturn-noise1sp.png`, `saturn-noise2sp.png` – two images with "Salt and Pepper" noise

The two images correspond to different amounts of noise. The `evalDenoising` script loads three images, visualizes them, and computes the error (squared-distance) between them as shown below. Your goal is to denoise the images which should result in a lower error value.


Figure 1: Input images for denoising.

- *[5 points]* Using Matlab's `imfilter` function, implement a Gaussian filter. You may find the function `fspecial` useful to create a Gaussian kernel. In Python, you can use the function `convolve` from `scipy.ndimage.filters`. Additionally, we added a function `gaussian` to `utils.py` that does the same as `fspecial('gaussian')`. Experiment with different values of the standard deviation parameter $\sigma$. Report the optimal error and $\sigma$ for each of these images.

  **My Answer:**

| image | optimal sigma | optimal error |
|---|---|---|
| ssaturn-noise1g.png | 1 | 148.38 |
| saturn-noise2g.png | 2 | 547.57 |
| ssaturn-noise1sp.png | 2 | 197.82 |
| saturn-noise2sp.png | 2 | 341.96 |

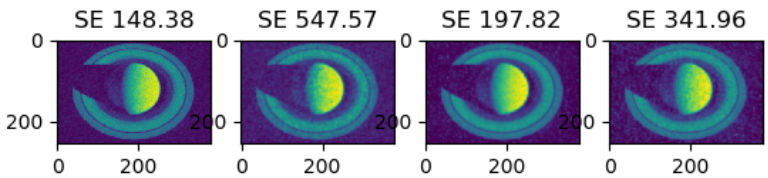



Figure 2: Results of denoising by using Gaussian filter

- *[5 points]* Using Matlab's `medfilt2` function, implement a median filter. In Python, you can use `medfilt2` from `scipy.signal`. Report the optimal error and neighborhood size for each of the images.

**My Answer:**

| image | neighbor size | optimal error |
|---|---|---|
| ssaturn-noise1g.png | 5 | 91.99 |
| saturn-noise2g.png | 7 | 233.42 |
| ssaturn-noise1sp.png | 3 | 8.78 |
| saturn-noise2sp.png | 3 | 18.57 |

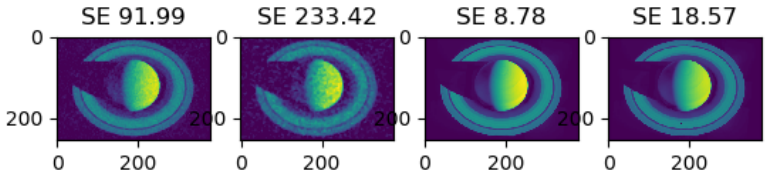



Figure 3: Results of denoising by using Median filter

- *[20 points]* Implement a function to compute non-local means filtering. Recall that 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.

  Let $I(x)$ denote the pixel value at $x$, $\hat{I}(x)$ denote the pixel value after denoising, $P(x)$ denote the patch of radius p centered around $x$, and $W(x)$ denote the pixels within a window of radius w around $x$. One choice of weight function is a Gaussian with a paramter ($\gamma$) resulting in the following update equation for each pixel:

$$\hat{I}(x) = \frac{\sum_{y \in W(x)} \exp\{-\gamma ||P(x) - P(y)||_2^2\} I(y)}{\sum_{y \in W(x)} \exp\{-\gamma ||P(x) - P(y)||_2^2\}}. \tag{1}$$

  Here $||x||_2^2$ denotes the squared length of the vector $x$. Write a function to implement this operation. Your function should take as input the noisy image, a patch radius (p), window radius (w), and a bandwidth parameter $\gamma$ and return denoised image.

  This algorithm is slower in comparison to the local-filtering based approaches since it involves pairwise distance computations to all pixels within each neighborhood. My implementation takes about 15 seconds for a patch size of 5x5 and neighborhood size of 21x21. I suggest debugging your algorithm and find the range of parameters that work well on a smaller crop of the image before trying it on the full image. Experiment with different values of the parameters and report the optimal ones and the corresponding errors for each of the images. Include your implementation in the submission.

**My Answer:**

I tried patch_size in the range of (3, 5, 7), window_size in the range of (5, 9, 11), and the value of gamma in the range of (1, 2, 3, 4). The optimal combination of parameters for each image shows

below:

| image | patch size | window size | gamma | error |
|---|---|---|---|---|
| ssaturn-noise1g.png | 7 | 11 | 3 | 100.01 |
| saturn-noise2g.png | 3 | 11 | 2 | 477.62 |
| ssaturn-noise1sp.png | 3 | 9 | 1 | 139.26 |
| saturn-noise2sp.png | 3 | 9 | 1 | 250.59 |

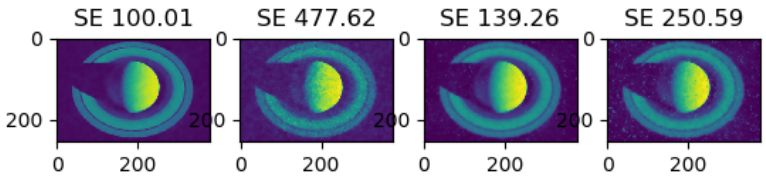



Figure 4: Results of denoising by using Non-local means filter

The steps of implementation:

```
Create a function named "DenoiseNLM", the input parameters are image and
noise images.
   patch_size = (3, 5, 7)
   window_size = (5, 9, 11)
   gamma = (1, 2, 3, 4)
   img_length, img_width = img.shape

   for different noise images:
       assume the initial error is infinite.
       for p in patch_size:
          for w in window_size:
             for r in gamma:
                Create a new array named "denoise_img" which has the same
                size with noise image. And the value of each element is 0.
                Pad the denoise_img with 0, the pad_size=(w-1)/2+(p-1)/2.
                half_patch_size = int((p - 1) / 2)
                half_search_window_size = int((w - 1) / 2)

                For pixel(i,j) in Patch(x),
                i in range(pad_size, img_length + pad_size):
                j in range(pad_size, img_width + pad_size):
                window_x = pad_img[i-half_patch_size: i+half_patch_size+1,
                                   j-half_patch_size: j+half_patch_size+1]
                For pixel(i1,j1) in Patch(y),
                i1 in range(i - half_search_window_size,
                i + half_search_window_size+1),
                j1 in range(j - half_search_window_size,
                j + half_search_window_size+1),
                window_y = pad_img[i1-half_patch_size: i1+half_patch_size + 1,
                                   j1-half_patch_size:j1+half_patch_size + 1]
```

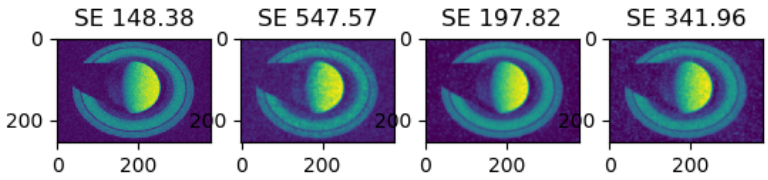
```
Calculate the Euclidean distance(d) between window_x and window_y,
which is quite like weights of I(y).
Calculate the sum of weights * I(y), this is numerator
Calculate the sum of weights this is denominator
denoise_img[i - pad_size, j - pad_size] = numerator / denominator

error = sum of ((denoise_img - img) ** 2)
Find the smallest error and the corresponding patch_size,
window_size, and gamma.
```

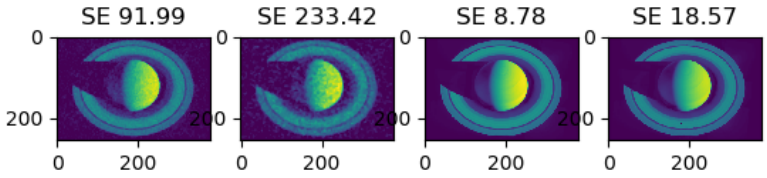
- *[5 points]* Qualitatively compare the outputs of these results. You should include the outputs of the algorithms side-by-side for an easy comparison. Which algorithm works the best for which image?

**My Answer:**

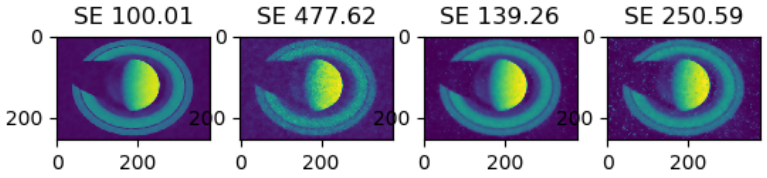
Figure5 shows the outputs of the three filters.




(a) Outputs of Gaussian filter




(b) Outputs of Median filter




(c) Outputs of Non-local means filter

Figure 5: Outputs of the three filters

Comparing the value of SE, for all of the 4 images (saturn-noise1g.png, saturn-noise2g.png, saturn-noise1sp.png, and saturn-noise2sp.png), the Median filter works best.

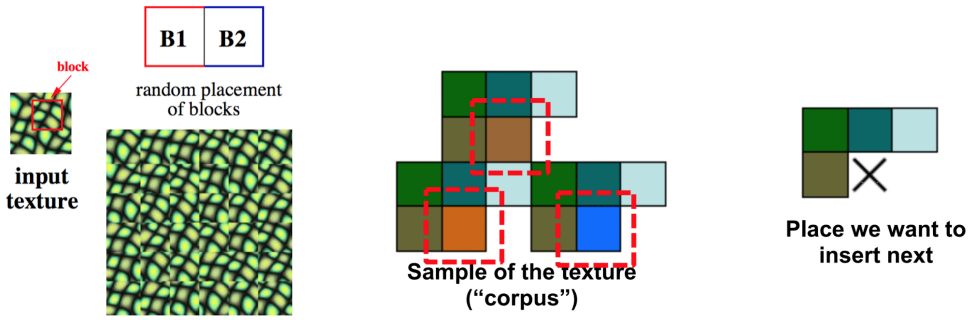
I intuitively feel that the results of median filter are not always the best. For example, for the output images in the second column, i.e. the outputs for saturn-noise2g.png by using three filters. The noise of this image is Gaussian noise. Comparing the output images, it looks like Gaussian filter and Non-local filter

work better than Median filter. However, the SE of Median filter is smallest.

I guess it's because median filter cannot replace all the pixel by median value. As for some pixels they are median value. But Gaussian filter and Non-local filter replace almost all the pixels by new value. Therefore, the SE of Gaussian filter and Non-local filter are bigger than median filter.

## 2 Texture synthesis [30 points]

In this part you will implement various methods synthesizing textures from a source image.




(a) Random tiling (b) Non-parametric sampling

Figure 6: (a) Creating an output image by randomly picking patches from a source image. (b) Generating an output image pixel (*right*) based on its neighbors in the source image (*left*) (from lecture slides). In this example, the output pixel is likely to get a blue color.

- *[5 points]* A simple approach for generating a texture is to randomly tile the target image using patches from a source image, as seen in Figure 6(a). The basic procedure is as follows:

  1. Initialize an empty output image.
  2. Pick a tile randomly from the source image and copy into the output image at the top-left position.
  3. Proceed to the next unfilled position in the output image in raster order (left-to-right, top-to-bottom).

  Your output images should consist of $5 \times 5$ tiles; try various tile sizes – 15, 20, 30, 40. Use grayscale versions of the three images provided in the `data/texture` folder.

  **My Answer:**

  The steps of my implementation:

```
Load three images
For each image:
  cast to the type of np.uint8
  get the width, length, and depth of each image
  define the number of tiles in output is 5 (numTiles = 5)
  try various tile sizes -- 15, 20, 30, 40 (tileSize)
  the output size is numTiles * tileSize
  create an empty image, the size is [outSize, outSize, depth]
      for i in range(numTiles):
```

```
for j in range(numTiles):
   choose the range of tile from source image randomly
   rand_x = np.random.randint(low=0, high=length-tileSize+1)
   rand_y = np.random.randint(low=0, high=width-tileSize+1)
   tile = im[rand_y:rand_y+tileSize,
             rand_x:rand_x+tileSize, :]
  copy tile to the corresponded position in the output image
  im_out[i*tileSize:i*tileSize + tileSize,
         j*tileSize:j*tileSize+tileSize, :] = tile
  write image output
```

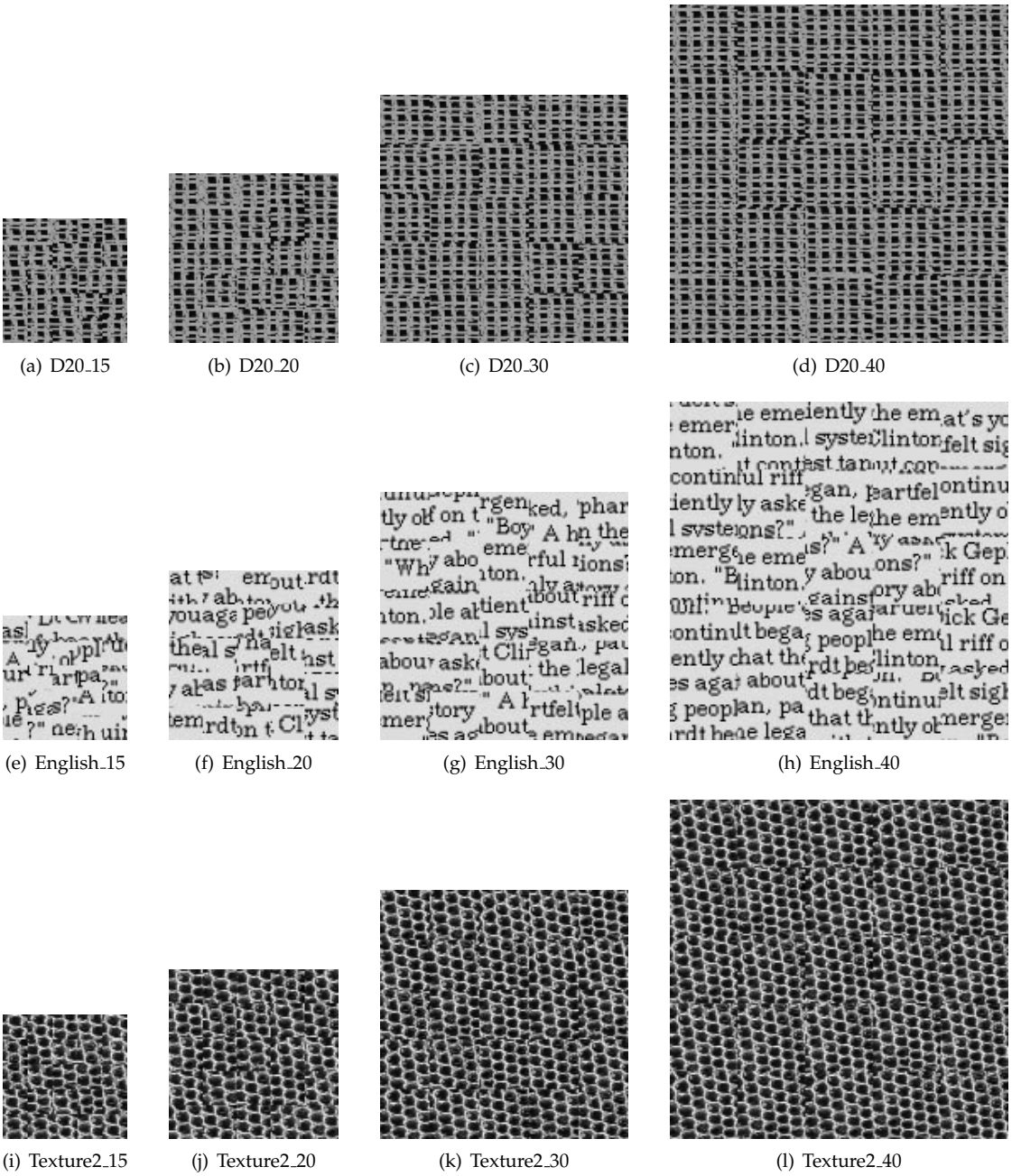
Figure 7 shows the outputs of implementing the simple approach on three images by trying various tile sizes – 15, 20, 30, 40.

- *[25 points]* The above method results in an image with artifacts around the edge of the tiles. This is because the tiles do not align well. The approach of Efros and Leung avoids this by generating the ouput, one pixel at a time, by matching the local neighborhood of the pixel in the generated image and source image, as shown in Figure 6(b). The basic approach is summarized as follows:

  1. Initialize an empty output image.
  2. Pick a $3 \times 3$ seed patch randomly from the source image and copy into the output image at the center position. The output image is generated by growing the borders of the filled pixels in the output image.
  3. Create a list of pixel positions in the output image that are unfilled, but contain filled pixels in their neighbourhood (`pixelList`).
  4. Randomly permute the list and then sort by the number of filled neighbourhood pixels.
  5. For each pixel in `pixelList`, get a `ValidMask` for its neighbourhood. This would be a $windowSize \times windowSize$ kernel, with 0s for unfilled positions and 1s at the neighbourhood positions of that pixel that contain filled pixels. Note, all this is done using the *output* image.
  6. Compute the sum of squared differences (SSD) between every patch in the *input* image w.r.t. the current unfilled pixel in the *output* image. Take care to ensure the `ValidMask` is applied when computing the SSD — we want to 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.
  7. Calculate `minSSD`, which is the smallest distance among all input image pixel neighbourhoods to the current output pixel's neighbourhood.
  8. Select a list of `BestMatches`, which are input image pixels within `minSSD*(1+ErrThreshold)`, where `ErrThreshold` is set to 0.1.
  9. Randomly pick an input image pixel from `BestMatches` and paste its pixel value into the current unfilled output pixel location.
  10. Keep repeating the steps for creating `pixelList` with unfilled output image locations, until all unfilled positions are filled.

  Not all these steps need to implemented exactly. For instance, you could come up with other strategies on how to grow the texture (e.g., grow the border systematically by going clockwise from the top-left corner.).

  The starter code for this part is in `evalTextureSynth`. The source textures are in the `data/texture` folder (Note that there are three). Your goal is to implement the texture synthesis algorithm so that for each small image provided, starting from a $3 \times 3$ seed patch, you are able to generate a $70 \times 70$ image. For this homework you should convert the input images to grayscale for faster processing. Show the effect of various window sizes – 5, 7, 11, 15. Discuss the effect of window size on runtime



(a) D20_15 (b) D20_20 (c) D20_30 (d) D20_40

(e) English_15 (f) English_20 (g) English_30 (h) English_40

(i) Texture2_15 (j) Texture2_20 (k) Texture2_30 (l) Texture2_40

Figure 7: Outputs of simple approach by using various tile sizes – 15, 20, 30, 40

and synthesis quality.

**My Answer:**

There is a fourth layer of alpha transparency in 'D20.png'. In order to observe the texture of the output image more clearly, I only took the [0:3] layer for all output images. That is [width, length, depth] for each image.

As Table 1 shows, there are two interesting regulars can be found from the running time:

1. The running time of each texture synthesis image will increase as the window size increasing. Because the number of pixels to be calculated increases as the window size of the kernel increasing. 2. No matter what the value of window size is, the runtime of image ”English” always ¿ ”Texture2” ¿ ”D20”. And the runtime of ”English” images is much more than ”Texture2” and ”D20”. It may be because the distribution of the texture features of the letters in ”English” is irregular compared to the other two images. It seems that the time of texture synthesis is also related to the complexity or randomness of the texture.

Table 1: Runtime (seconds) by using various window size

| Image | 5 | 7 | 11 | 15 |
|---|---|---|---|---|
| D20.png | 79.2 | 146.0 | 202.4 | 249.4 |
| Texture2.bmp | 415.4 | 524.0 | 601.7 | 657.1 |
| english.jpg | 1633.9 | 2649.2 | 3739.7 | 4920.1 |

Found from the image quality:

Intuitively, the quality of the texture synthetic image improved as the window sizes increase. As Figure 8 shows, the synthetic images close to the left used the smaller window size. The images close to the right used the larger window size. When the window size is too small, the method cannot capture enough features, i.e. cannot find the most similar patch during the synthesis process. Therefore, the texture synthesized in the left image is very random. When the window size is bigger, the more similar patch can be found. Therefore, synthetic texture is more regular. The window size corresponds to the degree of randomness in the resulting textures.

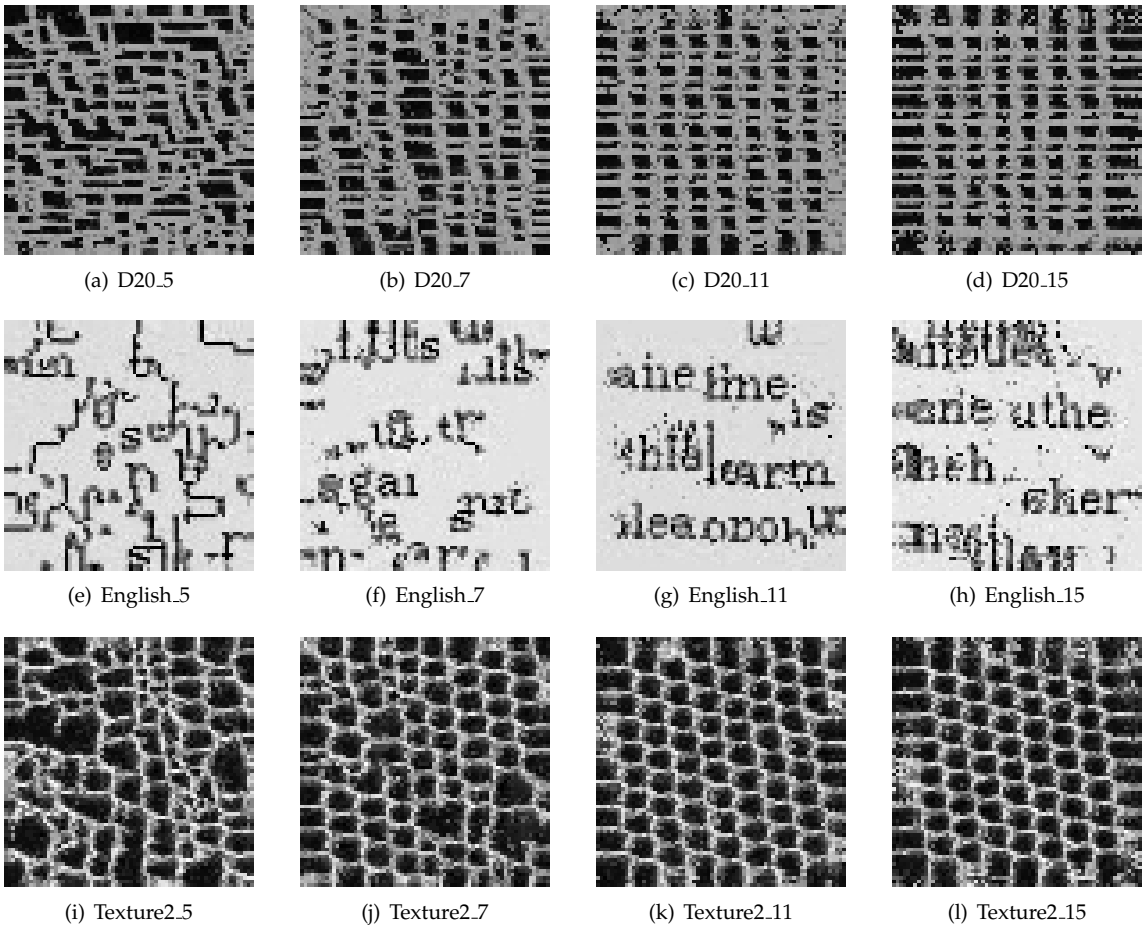
# 3 Report writing and presentation [10 points]

Please follow the guidelines for writing a good report. Graders will penalize reports that are poorly written and fail to present the results in a reasonable manner.

# 4 Extra credit [10 points]

Here are some ideas for extra credit.

1. **Block Matching 3D (BM3D).** The non-local means algorithm takes the weighted mean of the patches within a neighborhood. However there are other sophisticated approaches for estimating the pixel values. One such approach is BM3D (http://www.cs.tut.fi/~foi/GCF-BM3D/) which estimates the pixel value using a sparse basis reconstruction of the blocks. Compare an implementation of this approach to the non-local means algorithm.
2. **Image quilting.** The random tiling approach works reasonably well for some textures and is much faster than the pixel-by-pixel synthesis approach. However, it may have visible artifacts at the border. The “Image Quilting” approach from Efros and Freeman, 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, as seen in Figure 9. An extention is to carve a “seam” along the boundary



(a) D20_5 (b) D20_7 (c) D20_11 (d) D20_15

(e) English_5 (f) English_7 (g) English_11 (h) English_15

(i) Texture2_5 (j) Texture2_7 (k) Texture2_11 (l) Texture2_15

Figure 8: Outputs of the approach of Efros and Leung by using various tile sizes – 5, 7, 11, 15

which minimizes this error further, which can be solved using dynamic programming. The basic procedure (without seam carving) is as follows:

(a) Initialize an empty output image and copy a random tile from source image to the top-left corner

(b) Define the size of the region of overlap between tiles – typically 1/6.

(c) Find 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

(d) Find the tile in source image with lowest SSD (`minSSD`) with respect to the current tile in output image

(e) Make a list of all other tiles in the source image within a tolerance of the best tile's SSD (within `(1 + errTol) * minSSD`).

(f) Pick a tile randomly from the above list and copy into the output image

(g) Proceed to the next un-filled position in the output image (left-to-right, top-to-bottom)

Take a look at the paper for details: http://www.cs.berkeley.edu/~efros/research/quilting.html.

# 5 Code

1. **Image Denoising**

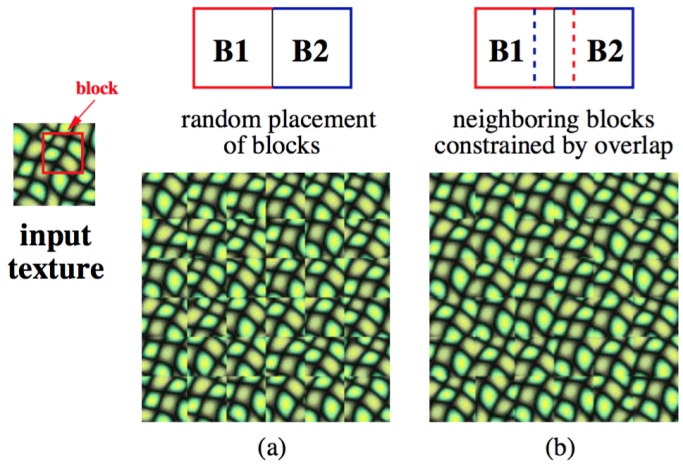



Figure 9: Image quilting. Figure source: https://people.eecs.berkeley.edu/~efros/research/quilting.html.

```
import os
import time
import numpy as np
import matplotlib.pyplot as plt
import sys
from scipy import signal
import scipy.ndimage.filters as scif
import matplotlib.image as mpimg

# Denoising algorithm (Gaussian filtering)
def DenoiseGaussian(img, noise_image):
  sigma = (0.7, 0.8, 0.9, 1, 2, 3, 4, 5)
  plt.figure(2)
  for n in range(4):
    error_best = np.Inf
    noise = noise_image[n]
    for s in sigma:
      output_n = scif.gaussian_filter(noise, s, order=0, output=None,
          mode='constant', cval=0.0)
      SSE_error = ((img - output_n) ** 2).sum()
      if SSE_error < error_best:
        error_best = SSE_error
        sb = s
    print(' For image{}, optimal sigma is {}, optimal SSE_error is {:.2f}
        '.format(n, sb, error_best))
    denoise_best = scif.gaussian_filter(noise, sb, order=0, output=None,
        mode='constant', cval=0.0)
    plt.subplot(141+n) #show 4 images in one picture
    plt.imshow(denoise_best)
    plt.title('SE {:.2f}'.format(error_best))
  plt.show()


# Denoising algorithm (Median filtering)
def DenoiseMedian(im, noise_image):
  neighbor_size = (1, 3, 5, 7, 9)
```

```
    plt.figure(3)
    for n in range(4):
      error_best = np.Inf
      noise = noise_image[n]
      for sz in neighbor_size:
        denoise = signal.medfilt2d(noise, sz)
        error = ((im - denoise)**2).sum()
        if error < error_best:
            error_best = error
            szb = sz
      denoise_best = signal.medfilt2d(noise, szb)
      print('neighbor_size = {}, error = {:.2f}'.format(szb, error_best))
      plt.subplot(141+n)
      plt.imshow(denoise_best)
      plt.title('SE {:.2f}'.format(error_best))
    plt.show()

def DenoiseNLM(img, noise_image):
    # n_length, n_width = noise.shape
    patch_size = (3, 5, 7)
    window_size = (9, 11)
    gama = (1, 2, 3)
    img_length, img_width = img.shape
    fig2 = plt.figure(4)
    for n in range(4):
       error_best = np.Inf
       for p in patch_size:
          for w in window_size:
             for r in gama:
                noise = noise_image[n]
                denoise = np.zeros(noise.shape, dtype=np.float)
                pad_size = int((w - 1) / 2 + (p - 1) / 2)
                pad_img = np.pad(noise, pad_size, 'constant')

                half_patch_size = int((p - 1) / 2)
                half_search_window_size = int((w - 1) / 2)

                # for P(x)
                for i in range(pad_size, img_length + pad_size):
                   for j in range(pad_size, img_width + pad_size):
                      win_x = pad_img[i - half_patch_size: i + half_patch_size +
                          1,j - half_patch_size: j + half_patch_size + 1]
                      s_w_value = 0
                      s_weight = 0

                      # for P(y)
                      for i1 in range(i - half_search_window_size, i +
                          half_search_window_size + 1):
                         for j1 in range(j - half_search_window_size, j +
                             half_search_window_size + 1):
                            win_y = pad_img[i1 - half_patch_size: i1 +
                                half_patch_size + 1,j1 - half_patch_size:j1 +
                                half_patch_size + 1]
                            d = np.linalg.norm((win_x - win_y), 2) ** 2
                            s_w_value += np.exp(-r * d) * pad_img[i1, j1]
                            s_weight += np.exp(-r * d)
                      denoise[i - pad_size, j - pad_size] = s_w_value / s_weight

                error1 = ((denoise - img) ** 2).sum()
```

```
                print('p = {}, w = {}, r = {}, error = {}'.format(p, w, r, error1))
                if error1 < error_best:
                    error_best = error1
                    wb = w
                    rb = r
                    pb = p
                    denoise_best = denoise

        print(pb, wb, rb)
        print(' For image{}, pb is {}, wb is{}, rb is{}, error is {:.2f}
            '.format(n, pb, wb, rb, error_best))

    plt.subplot(141 + n)
    plt.imshow(denoise_best)
    plt.title('SE {:.2f}'.format(error_best))

plt.show()

img = mpimg.imread('../data/denoising/saturn.png')
noise11 = mpimg.imread('../data/denoising/saturn-noise1g.png')
noise12 = mpimg.imread('../data/denoising/saturn-noise2g.png')
noise21 = mpimg.imread('../data/denoising/saturn-noise1sp.png')
noise22 = mpimg.imread('../data/denoising/saturn-noise2sp.png')

error11 = ((img - noise11)**2).sum()
error12 = ((img - noise12)**2).sum()

error21 = ((img - noise21)**2).sum()
error22 = ((img - noise22)**2).sum()

print('Input, Errors: {:.2f} {:.2f} {:.2f} {:.2f}'.format(error11, error12,
    error21, error22))

plt.figure(1)

plt.subplot(231)
plt.imshow(img)
plt.title('Input')

plt.subplot(232)
plt.imshow(noise11)
plt.title('SE {:.2f}'.format(error11))

plt.subplot(233)
plt.imshow(noise12)
plt.title('SE {:.2f}'.format(error12))

plt.subplot(235)
plt.imshow(noise21)
plt.title('SE {:.2f}'.format(error21))

plt.subplot(236)
plt.imshow(noise22)
plt.title('SE {:.2f}'.format(error22))

plt.show()

noise_image = (noise11, noise12, noise21, noise22)
```

```
DenoiseGaussian(img, noise_image)
DenoiseMedian(img, noise_image)
DenoiseNLM(img, noise_image)
```

# 2. Texture Synthesis

## 2.1 Simple Approach

```
import numpy as np
import os
from skimage import io
import matplotlib.pyplot as plt

D20 = io.imread('../data/texture/D20.png')
Texture2 = io.imread('../data/texture/Texture2.bmp')
english = io.imread('../data/texture/english.jpg')
img = (D20, Texture2, english)

def synthRandomPatch(img):
   for n in range(3):
      im = img[n]
      im = im.astype(np.uint8)
      [width, length, depth] = im.shape
      tileSizeList = (15, 20, 30, 40)
      numTiles = 5
      for m in range(4):
         tileSize = tileSizeList[m]
         outSize = numTiles * tileSize # calculate output image size
         im_out = np.zeros([outSize, outSize, depth])
         for i in range(numTiles):
            for j in range(numTiles):
               rand_x = np.random.randint(low=0, high=length-tileSize+1)
               rand_y = np.random.randint(low=0, high=width-tileSize+1)
               tile = im[rand_y:rand_y+tileSize,
                       rand_x:rand_x+tileSize, :]
               im_out[i*tileSize:i*tileSize + tileSize,
                   j*tileSize:j*tileSize+tileSize, :] = tile
               # Write image output
               imageName = ('D20', 'Texture2', 'english')
               out_imgname = '{}_{}.png'.format(imageName[n], tileSizeList[m])
               out_imgpath = os.path.join('../data/output/', out_imgname)
               im_out_nor = np.interp(im_out, (im_out.min(), im_out.max()), (0,
                   +1))
               plt.imsave(out_imgpath, im_out_nor)
      plt.figure()
      plt.imshow(im_out)
      plt.show()
   return im_out

im_patch = synthRandomPatch(img)
```

## 2.2 Efros and Leung

```
import numpy as np
from skimage import io
from datetime import datetime
import os
import matplotlib.pyplot as plt
```

```
def process_pixel(x, y, img_data, new_img_data, mask, win_size):
   ErrThreshold = 0.1
   win_size = int(win_size/2)
   x0 = max(0, x - win_size)
   y0 = max(0, y - win_size)
   x1 = min(new_img_data.shape[0], x + win_size+1)
   y1 = min(new_img_data.shape[1], y + win_size+1)

   new_window = new_img_data[x0 : x1, y0 : y1, 0]
   mask_window = mask[x0 : x1, y0 : y1]
   len_mask = float(sum(sum((mask_window))))
   [xs, ys] = new_window.shape
   img_xsize = img_data.shape[0]
   img_ysize = img_data.shape[1]

   cx = int(np.floor(xs/2))
   cy = int(np.floor(ys/2))

   BestMatches = []
   dists = []

   for i in range(xs, img_xsize - xs):
      for j in range(ys, img_ysize - ys):
         if(np.random.randint(0,2) != 0): continue # To improve runtime
         origin_window = img_data[i : i+xs, j : j+ys, 0]
         # distance
         s = (origin_window - new_window)
         error_matrix = (s**2)*mask_window
         d = sum(sum(error_matrix)) / len_mask
         BestMatches.append(origin_window[cx, cy])
         dists.append(d)
   best_dists_index = (dists) <= (1 + ErrThreshold) * min(dists)
   BestMatches = np.extract(best_dists_index, BestMatches)

   # pick random among candidates
   if len(BestMatches) < 1:
      return 0.0
   else:
      if len(BestMatches) != 1:
         r = np.random.randint(0, len(BestMatches) - 1)
      else:
         r = 0
   return BestMatches[r]

def getPixelList(valid_pixel_map):
   pixel_list = []
   for i in range(0, valid_pixel_map.shape[0]):
      for j in range(0, valid_pixel_map.shape[1]):
         x0 = max(0, i - 1)
         y0 = max(0, j - 1)
         x1 = min(valid_pixel_map.shape[0], i + 2)
         y1 = min(valid_pixel_map.shape[1], j + 2)

         if (valid_pixel_map[i,j] == 0):
            if(sum(sum(valid_pixel_map[x0:x1, y0:y1])) > 0):
               pixel_list.append([i, j])

   return np.array(pixel_list)
```

```
def synthEfrosLeung(im, winSize, outSize):
   start_t = datetime.now()
   for n in range(3):
      print('For image', n)
      img = im[n]
      img = img.astype(np.uint8)
      [width, length, depth] = img.shape
      for m in range(4):
         winsize = winSize[m]
         seed_size = 3
         half_output_size = int(outSize/2)
         half_seed_size = 1
         half_win_size = int(winsize/2)

         img_out = np.zeros([outSize, outSize, depth])
         valid_pixel_map = np.zeros((outSize, outSize)) # 0 means pixel not
             filled yet, 1 means pixel already filled

         seed_x = np.random.randint(low=0, high=img.shape[0] - seed_size)
         seed_y = np.random.randint(low=0, high=img.shape[1] - seed_size)

         # take 3x3 start image (seed) in the original image
         seed_data = img[seed_x: seed_x + seed_size, seed_y: seed_y + seed_size,
             :]

         img_out[half_output_size - half_seed_size: half_output_size +
             half_seed_size+1,
               half_output_size - half_seed_size: half_output_size +
                   half_seed_size+1, :] = seed_data

         valid_pixel_map[half_output_size - half_seed_size: half_output_size +
             half_seed_size+1,
                         half_output_size - half_seed_size: half_output_size +
                             half_seed_size+1] = 1

         # TO DO: non-square images
         while (int(np.sum(valid_pixel_map)) !=
             valid_pixel_map.size):#(getPixelList(valid_pixel_map.astype(int)).all()):
            #print("Processed {} of {} pixels".format(np.sum(valid_pixel_map),
                valid_pixel_map.size))
            #pixel_list = getPixelList(valid_pixel_map.astype(int))
            pixel_list = getPixelList(valid_pixel_map.astype(np.uint8))

            for pixel_position in pixel_list:
               x = pixel_position[0]
               y = pixel_position[1]

               pixel_out = process_pixel(x, y, img, img_out, valid_pixel_map,
                   winsize)
               img_out[x, y] = pixel_out
               valid_pixel_map[x, y] = 1

         # Time and write image output
         end_t = datetime.now()
         imageName = ('D20', 'Texture2', 'english')
         out_imgname = '{}_{}.png'.format(imageName[n], winsize)
         print("Process Time of", '{}_{}.png'.format(imageName[n], winsize),
             int((end_t - start_t).total_seconds() * 1000), " milliseconds")
```

```
            out_imgpath = os.path.join('../data/output/q2/', out_imgname)
            im_out_nor = np.interp(img_out[:, :, 0:3], (img_out.min(),
                img_out.max()), (0, +1))
            plt.imsave(out_imgpath, im_out_nor)

        plt.figure()
        plt.imshow(img_out[:, :, 0:3].astype(np.uint8))
        plt.show()
    return img_out

# Load images
D20 = io.imread('../data/texture/D20.png')
Texture2 = io.imread('../data/texture/Texture2.bmp')
english = io.imread('../data/texture/english.jpg')
img = (D20, Texture2, english)

# Non-parametric Texture Synthesis using Efros & Leung algorithm
winSize = (5, 7, 11, 15)
outSize = 70
im_synth = synthEfrosLeung(img, winSize, outSize)
```