# COMPSCI 670: Computer Vision Fall 22

Homework 2

Prachi Jain

## 1 Short answers

### 1.1

Assuming a pinhole camera with focal length = 1, a sphere of radius = 1, standard geometry where the origin is the optical center and the image is formed parallel to the x-y plane, and z is along the optical axis.

Projection of a Sphere under a perspective camera model is as follows:

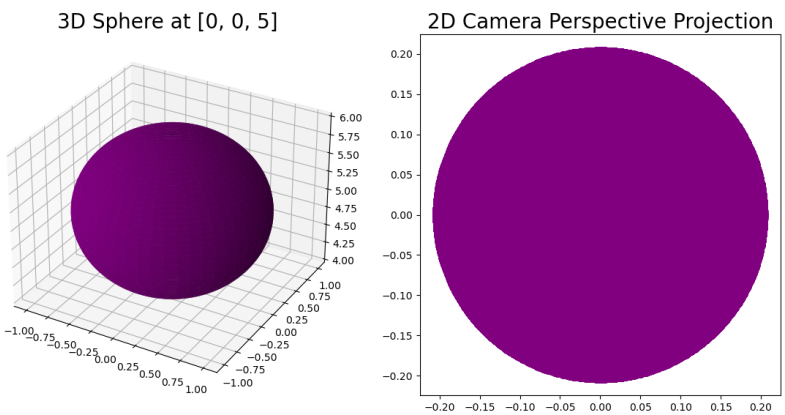



Figure 1: Projection for coordinates (0, 0, 5)

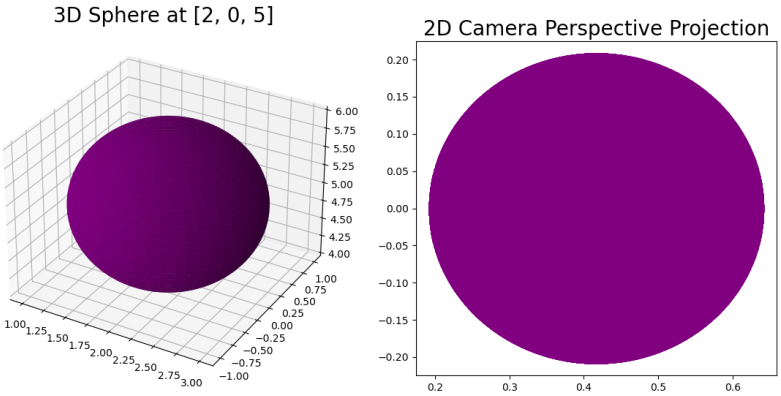



Figure 2: Projection for coordinates (2, 0, 5)

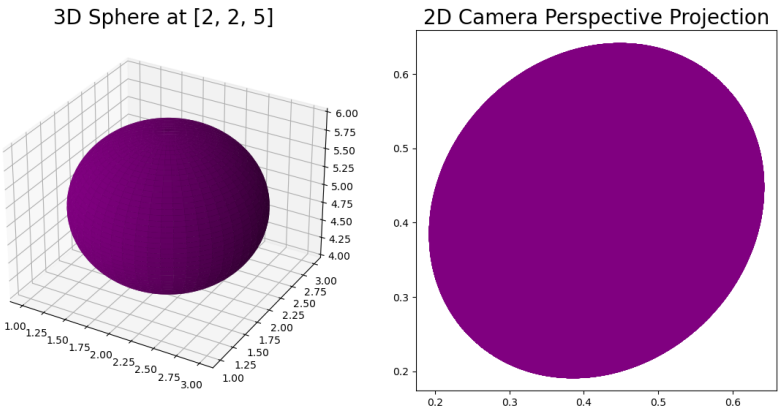



Figure 3: Projection for coordinates (2, 2, 5)

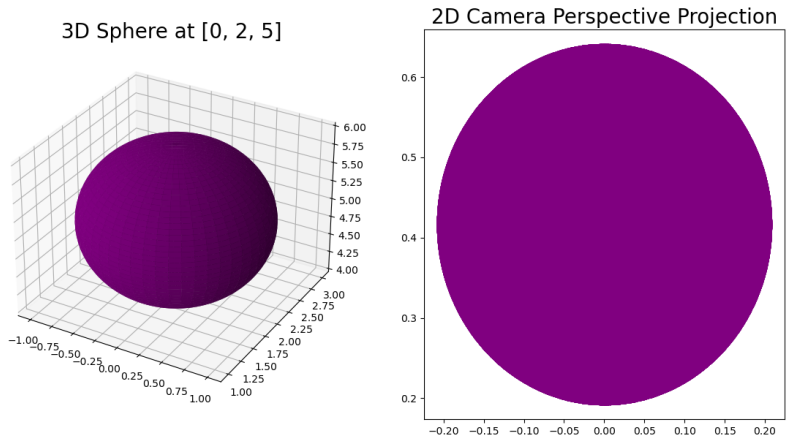



Figure 4: Projection for coordinates (0, 2, 5)

Implementation of *question1_1.py*

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt

#For 3D plotting
from mpl_toolkits import mplot3d

def Sphere3D(x, y, z, radius = 1):
  #Sampling 500 points- a large number of points on the surface of the sphere
  _pi = np.pi
  _phi = np.linspace(0, 2 * _pi, 500)
  _theta = np.linspace(0, _pi, 500)
  _countPhi = np.ones(np.size(_phi))

  a = x + radius * np.cos(_phi)[:,None] * np.sin(_theta)[None,:]
  b = y + radius * np.sin(_phi)[:,None] * np.sin(_theta)[None,:]
  c = z + radius * _countPhi[:,None] * np.cos(_theta)[None,:]

  return a, b, c

def PinholeCamera2D(x, y, z, focal_length = 1):
  #Using Perespective Equations
  pinholeX = (focal_length * x)/z
  pinholeY = (focal_length * y)/z

  return pinholeX, pinholeY

locations = [[0, 0, 5], [2, 0, 5], [2, 2, 5],[0, 2, 5]]

for coords in locations:
  X, Y, Z = Sphere3D(coords[0], coords[1], coords[2])
  pinholeX, pinholeY = PinholeCamera2D(X, Y, Z)
```

```

a , b = pinholeX.shape
numSamplePoints = a * b
cameraX = list(pinholeX.reshape(numSamplePoints,))
cameraY = list(pinholeY.reshape(numSamplePoints,))

fig = plt.figure(figsize =(14, 9))
ax1 = fig.add_subplot(121, projection='3d')
ax1.plot_surface(X, Y, Z, color='purple')
ax1.set_title('3D Sphere at {}'.format(coords), fontsize = 20)

ax2 = fig.add_subplot(122)
ax2.scatter(cameraX, cameraY,  color='purple')
ax2.set_title('2D Camera Perspective Projection', fontsize = 20)
ax2.set_aspect('equal')
plt.show()
```

Shape of the projection for the following views are observed from *figures 1, 2, 3 and 4*:

1. (0, 0, 5) -> Circle
2. (2, 0, 5) -> Ellipse
3. (2, 2, 5) -> Ellipse
4. (0, 2, 5) -> Ellipse

## 1.2

In the beginning, people used a tiny aperture, a.k.a. "pinhole," to capture a 2D representation of the 3D view of the world, capturing a pencil of rays through a barrier. All rays pass through a single point: an aperture (center of projection/ focal point/camera center) to form an image on the image plane. During dimensionality reduction from 3D to 2D, the pinhole camera model preserved only straight lines and incidence leaving away the angles and the lengths. The concept behind the pinhole camera was to remove image blurriness when a film is put in front of an object.

However, there are limitations to the Pinhole Camera model:

1. It only works on specific aperture sizes. If the pinhole is too big, many directions are averaged, blurring the image. If the pinhole is too tiny, diffraction becomes a factor, blurring the image.

2. Generally, pinhole cameras are dark because only a very small set of rays from a particular scene point hits the image plane.

3. Pinhole cameras are slow because only a very small amount of light from a particular scene point hits the image plane per unit of time.

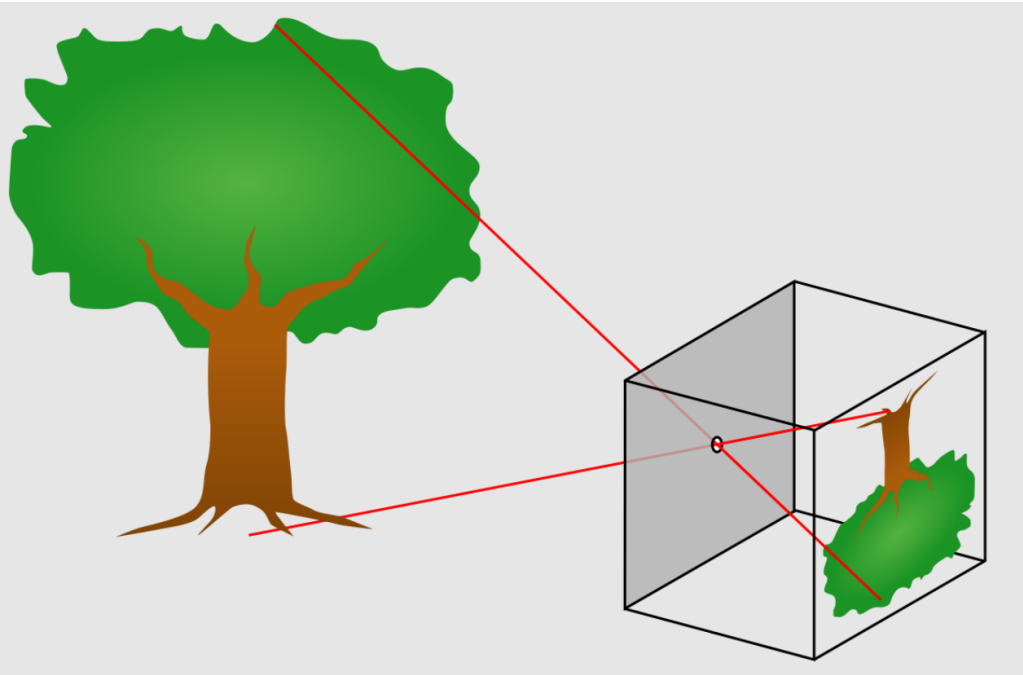


Figure 5: Schematic of a pinhole camera model

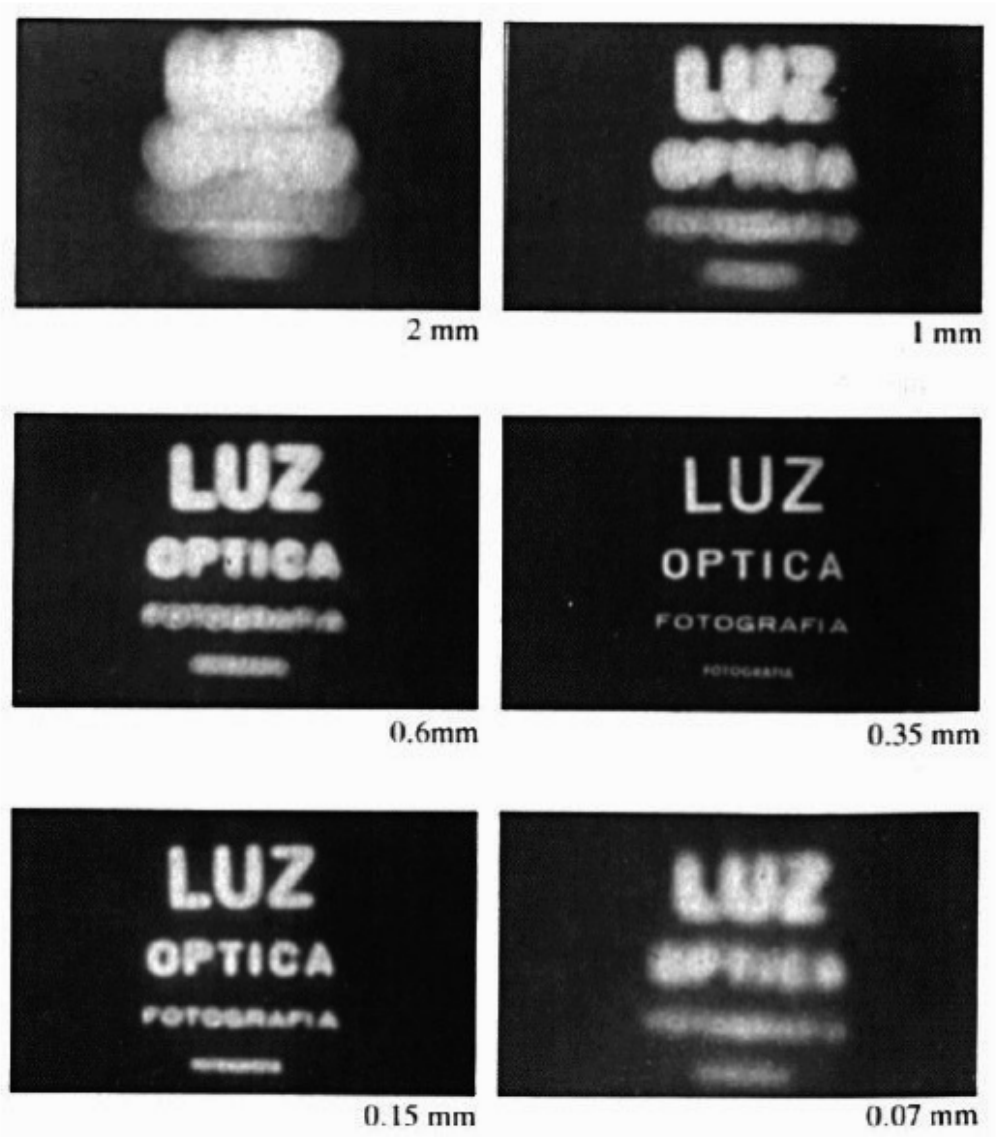





Figure 6: Result of shrinking the pinhole camera aperture.[Image Credit: Credit: E. Hecht. "Optics," Addison-Wesley, 1987]

A Lens is an optical device that focuses (convex) or disperses (concave) light rays through refraction onto a film.

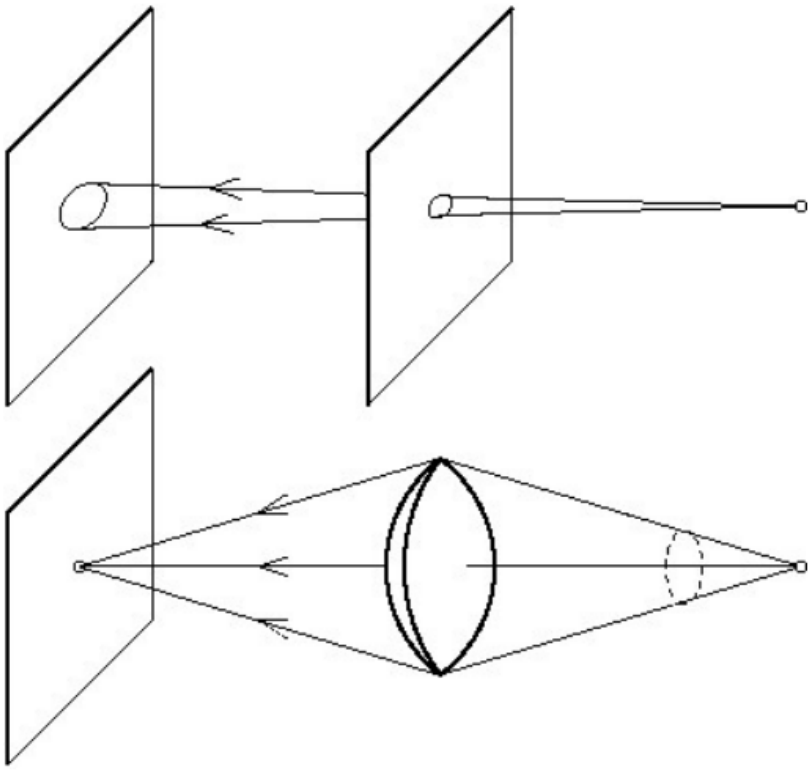


Figure 7: Role of a Lens in Camera

In a Thin lens model:
1. Rays passing through the center do not deviate (the pinhole projection model still holds).
2. All parallel rays converge to one point on a plane which is located at the focal length f.
3. This convergence makes objects "in focus" (greater intensity and sharpness) at a specific distance while other points project onto a "circle of confusion" in the image.

The lens is an abstraction of an ideal pinhole camera.

1. The lens directs and focuses light rays on giving an accurate representation of the 3D counterparts of the object. Without a lens, light from all directions reaches every point on the film or sensor to create a gray blur image.

2. In a film, the sensor or back wall of the human eyeball, the view is usually much smaller than the actual view attempting to capture. Therefore, the bending of light is required to reduce the size of an image. A camera lens also slows down the light speed—the speed of light changes when it passes through translucent materials. So, light is bent and slowed down as it enters and exits a lens (depending on the design of the lens), thus giving it more exposure.

Thus, the role of a lens in a camera is to capture more light while preserving as much as possible to give a high-resolution image as fast as possible. The lens is that device which will focus and amplify the effectiveness of light gathering to a tiny area (retina, a piece of photosensitive film, or a digital photo-diode)

## 1.3

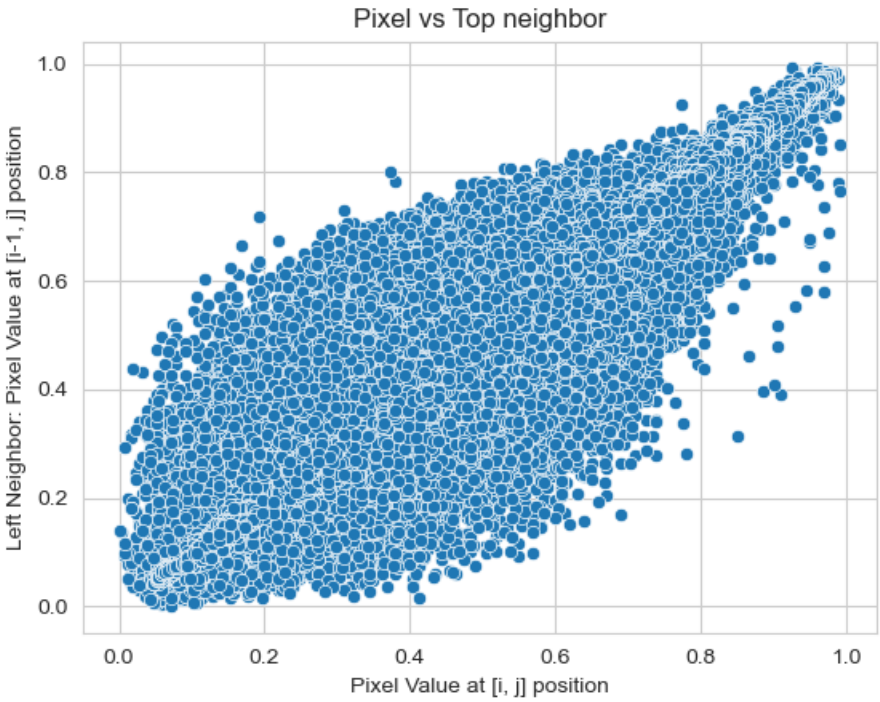



Figure 8: Correlation between Pixel Values and Top Neighbor Values

The higher density along the y = x line proves that a pixel will probably have the same or approximately the same value as its neighbor and is confirmed by pixel value correlation in *figure 8*.

There are few outliers to this assumption, as observed in *figure 8*, because the boundary of 1-pixel thickness has not been considered.

This property can be used in demosaicing an image with the assumption that an unknown pixel's value will be highly probable to have its neighboring pixel's value, which is the core idea behind **Nearest Neighbor Interpolation**.

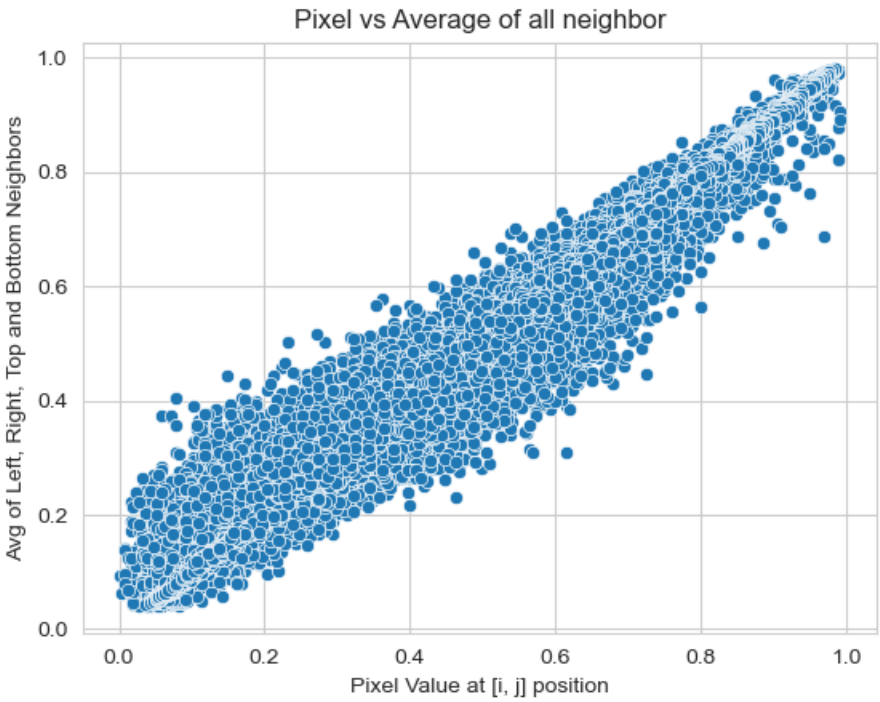



Figure 9: Correlation between Pixel Values and Average of all Neighbor Values

Compared with the correlation plot for the pixel and top neighbor in *figure 8*, the correlation plot for the pixel and average of all neighbors in *figure 9* shows the points more congested towards the y = x line in the plot.

It is because an average of all neighbors will always provide a better idea about the center pixel values instead of just looking at a neighbor in one direction.

Classic cases, like when the left neighbor is black, but all the other three neighbors are white, can make the central pixel almost white too. By only using the black neighbor, the center pixel value can be falsely classified as black too.

Therefore, in demosaicing, it is always better to consider all neighbors whose values are known instead of picking out the closest one. The average is more likely to be near the actual value, which is the idea behind **Linear Interpolation**.

Implementation of *question1_3.py*

```
from skimage import io, color
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

img = io.imread("../data/cat.png")
img = color.rgb2gray(img[:,:,:])

height, width = img.shape
pixel = []
topNeigh = []
allNeigh = []
```

```
#For each pixel in the image except those on the borders (removing layer of 1 pixel around the borders)
for i in range(1, height - 1):
    for j in range(1, width - 1):
        pixel.append(img[i,j])
        topNeigh.append(img[i - 1,j])
        allNeigh.append(np.mean([img[i - 1, j], img[i, j - 1], img[i + 1, j], img[i, j + 1]]))

sns.set_style(style='whitegrid')
p1 = sns.scatterplot(pixel, topNeigh)
p1.set(xlabel ="Pixel Value at [i, j] position", ylabel = "Left Neighbor: Pixel Value at [i-1, j] position", title ='Pixel vs Top neighbor')
plt.show()
p2= sns.scatterplot(pixel, allNeigh)
p2.set(xlabel ="Pixel Value at [i, j] position", ylabel = "Avg of Left, Right, Top and Bottom Neighbors", title='Pixel vs Average of all neighbor')
plt.show()
```

## 1.4

Color Constancy is the human visual system's ability to ensure that objects' perceived color remains relatively constant under varying illumination conditions. Color is consistent under different lighting conditions as long as enough wavelengths(light) are available to see.

An example of subjective constancy is color constancy, a process that allows the brain to recognize a familiar object as being a consistent color regardless of the amount or wavelengths of light reflecting from it at a given moment.

The phenomenon of color constancy occurs when the source of illumination is not directly known. This is why color constancy takes a greater effect on days with sun and clear sky than on overcast days.

Figure 10: Color Constancy Example -The colors of a hot air balloon are recognized as being the same in the sun and shadow.

As written in David A. Forsyth, Jean Ponce - Computer vision_ A modern approach-Prentice Hall (2011) - Von Kries assumed that color constancy was, in essence, the result of independent lightness calculations in each channel, meaning that one can rectify an image by scaling each channel independently. This practice is known as Von Kries' law. Color constancy can be approached as an inference problem; ideally, one needs to work in shadows and surface orientation to solve it.

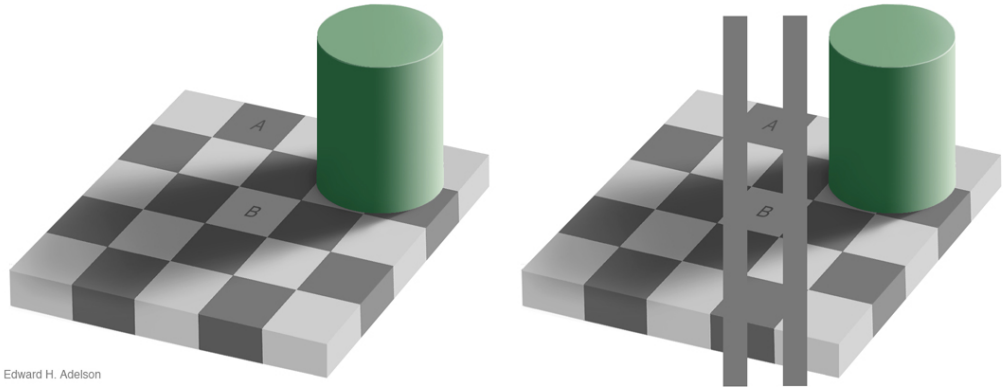



Figure 11: Adelson's checker-shadow illusion: With lines showing Color Constancy the squares marked A and B are the same shade of gray

*Adelson's checker-shadow illusion is an optical illusion published by Edward H. Adelson, professor of vision science at MIT, in 1995.Constancy makes square A appear darker than square B when in fact, they are both exactly the same shade of gray.*

The image depicts a checkerboard with light and dark squares, partly shadowed by another object. The optical illusion is that the area labeled A appears to be a darker color than the area labeled B. However, within the context of the two-dimensional image, they are of identical brightness, i.e., they would be printed with identical mixtures of ink or displayed on a screen with pixels of the identical color.

This illusory effect is called simultaneous lightness contrast (SLC), and it has a high-level explanation and a low-level one.

1. The low-level constancy mechanism is referred to as Hering-type lightness contrast after German physiologist Ewald Hering (1834-1918). Hering emphasized the explanatory power of localized retinal processes affecting the neurons, and retinal ganglion cells, which send signals from the eye to the brain. The retinal ganglion cells receive electrical impulses from the photoreceptive rods and cones contained in the outer retina and send corresponding signals to the brain, producing a measure of the intensity of light hitting the eye (luminance). For each retinal ganglion cell, there is a corresponding receptive field, which may be thought of as the small region of rods and cones which can affect the given cell's firing rate. Two kinds of retinal ganglion cells exist: an ON-centre cell is excited by a central bright spot in its field but is inhibited by a bright surround; an OFF-centre cell has the opposite arrangement. Spatial filtering is the process of turning the retinal stimulus into a neural image. The inhibitory mechanism operates across the 2-D space of the receptive field to effectively remove the image data wherever there is uniform luminance.

2. The high-level explanation of lightness constancy is associated with Hermann von Helmholtz

(1821-1894), a German physician, physicist, and philosopher of science. As a physicist, Helmholtz believed that the eye was an imperfect optical instrument and that the brain must combine raw sensory input and previous experience to arrive at a "best guess" as to what is out in the world. According to Helmholtz (1867), perception results from 'unconscious inferences.' Lightness constancy, in particular, is achieved by the visual system inferring and then discounting the illuminant. While there is a shadow depicted as falling across the checkerboard, there isn't really a shadow there—for Adelson's Checker-Shadow illusion is just a picture of a shadow. And this is what causes the illusion. Adelson's Checker-Shadow Illusion is, therefore, the result of our inductive inferences failing to line up with the world. Our visual system arrives at a false conclusion using premises based on sensory evidence, perceived lighting conditions, and previous experiences.

**Reference:**

1. `https://www.illusionsindex.org/ir/checkershadow`
2. `https://en.wikipedia.org/wiki/Color_constancy`

## 1.5

Tetrachromatic color vision allows an individual to discriminate colors along a perceptual dimension that is unavailable to the normal person. Most human color perception is 'trichromatic,' i.e. their retinas contain three types of light-sensitive cone photopigment, absorbing maximally in the long-wavelength (L), middle-wavelength (M) and short-wavelength (S) parts of the spectrum.

Human tetrachromacy appears in two variants, L or M cone, differing in spectral sensitivity, as present in the human retina. Due to the gender-linked inheritance of the L and M photopigments, this condition occurs only in women. Adjacent genes on the X-chromosome encode the photopigments in the L and M cones. Men inherit a single maternal X-chromosome, whereas women inherit one from each parent. However, to compensate for the difference in X-chromosome dosage between men (XY) and women (XX), most genes on one X-chromosome are inactivated in any given cell of a woman's body. This inactivation is random and initiated very early in embryonic development.

Consider now a woman whose X-chromosomes carry different versions (alleles) of the M opsin gene. Suppose these alleles encode photopigments that differ in their wavelengths of maximal sensitivity. In that case, her retina will contain four types of cone photopigment: S, M, L' (or shifted M), and L. X-chromosome inactivation segregates the alternative gene products in different cones and so provides a retinal cone mosaic with four cone types. The size of retinal cone patches will depend on the time of onset of X inactivation and the migration of cones in the developing retina.

Thus, tetrachromatic women are the daughters of anomalous trichromats for whom one photopigment (M or L) is spectrally shifted.

Following are the variation in tetrachromatic vision, where an extra cone in the human eye can develop the capability to have a differential vision.

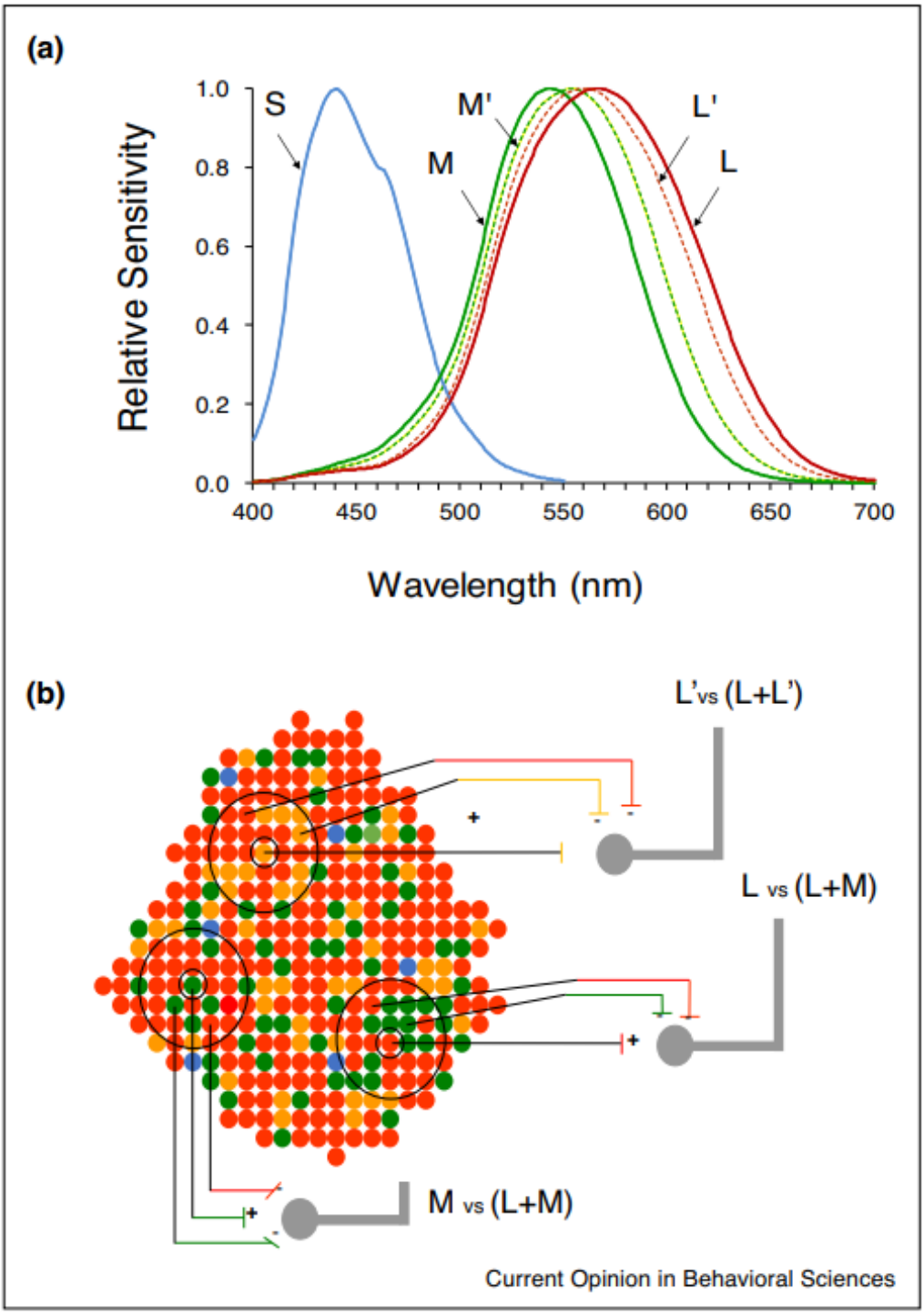



Figure 12: Retinal structures for tetrachromacy
(a) Relative sensitivities for the three normal cone photopigments S, M, L and two hybrid photopigments L' and M'.
(b) Schematic representation of a foveal cone mosaic showing S, M, L, and L' cones in blue, green, red, and orange, respectively. Cone inputs into three possible circular centers surrounding receptive fields of midget ganglion cells are superimposed on the cone mosaic.

**References:** `https://vision.psychol.cam.ac.uk/jdmollon/papers/JordanMollon2019Tetrachromacy.pdf`

**To show mathematically that there must exist two spectra that are indistinguishable from the normal eye but appear different to a tetrachromat:**

Assume a color-matching experiment.
Let $C(\lambda)$ be the spectrum of the light beam which falls on the surface and $S(\lambda)$, $L(\lambda)$, $M(\lambda)$ and $P(\lambda)$ be the 4 photodetectors with spectral sensitivity curves.

Behaviorally, using spectral lights, normal/humans will match both the spectra (color) with just three variable colors. A tetrachromat woman will require four independent stimuli to match colors in two spectra.

**Colour 1:** Suppose one color(spectra) combines S, L, M, and the fourth cone, say P, whose value is 0 (P is absent). This is a weak tetrachromacy case as even with four types of cones, vision might remain trichromatic in the sense that only three variables are needed in a color-matching experiment.

After quantize, radiant energy of the color is given by:

$$S = \int_{\omega} C(\lambda) \cdot S(\lambda)\, d\lambda$$
$$L = \int_{\omega} C(\lambda) \cdot L(\lambda)\, d\lambda$$
$$M = \int_{\omega} C(\lambda) \cdot M(\lambda)\, d\lambda$$
$$P = \int_{\omega} C(\lambda) \cdot P(\lambda)\, d\lambda = 0$$

**Colour 2:** Let's say the second color (spectra) involves using all four cones S, L, M, and P. Here the S, L, and M values are the same as the previous one, and we also have a value for the P cone. (P is present). So it's a strong tetrachromacy case.

After quantize, radiant energy of the color is given by:

$$S = \int_{\omega} C(\lambda) \cdot S(\lambda)\, d\lambda$$
$$L = \int_{\omega} C(\lambda) \cdot L(\lambda)\, d\lambda$$
$$M = \int_{\omega} C(\lambda) \cdot M(\lambda)\, d\lambda$$
$$P = \int_{\omega} C(\lambda) \cdot P(\lambda)\, d\lambda$$

In this experiment, both spectra will look identical to the normal human (trichromatic) eye; hence, they will match both spectra equally, but both spectra will appear different to a tetrachromatic eye. So a tetrachromatic person will add the value of the P cone to the first spectra or subtract the value of the P cone from the second spectra to make both spectra the same in a color-matching experiment.

In conclusion, with another cone a tetrachromatic person will perceive more rich and varied color ranges than a normal human due to more combinations of colors arised from a four set colour value than a 3 set colour value due to the presence of an extra cone.

# 2 Color image demosaicing

## 2.1 Implementation

Implementation of *demosaicImage.py*

```
# This code is part of:
#
#   CMPSCI 670: Computer Vision, Fall 2022
#   University of Massachusetts, Amherst
#   Instructor: Subhransu Maji

import numpy as np
import matplotlib.pyplot as plt
import math

def rescale(A):
    return (A - np.min(A))/(np.max(A) - np.min(A))

def descale(A):
    return A*(np.max(A) - np.min(A)) + np.min(A)


def demosaicImage(image, method):
    ''' Demosaics image.

    Args:
        img: np.array of size NxM.
        method: demosaicing method (baseline or nn).

    Returns:
        Color image of size NxMx3 computed using method.
    '''

    if method.lower() == "baseline":
        return demosaicBaseline(image.copy())
    elif method.lower() == 'nn':
        return demosaicNN(image.copy()) # Implement this
    elif method.lower() == 'linear':
        return demosaicLinear(image.copy()) # Implement this
    elif method.lower() == 'adagrad':
        return demosaicAdagrad(image.copy()) # Implement this
    elif method.lower() == 'divide':
        return demosaicDivide(image.copy()) # Implement this
    elif method.lower() == 'logdivide':
        return demosaicLogDivide(image.copy()) # Implement this
    else:
        raise ValueError("method {} unkown.".format(method))


def demosaicBaseline(img):
    '''Baseline demosaicing.

    Replaces missing values with the mean of each color channel.

    Args:
        img: np.array of size NxM.

```

```
    Returns:
        Color image of sieze NxMx3 demosaiced using the baseline
        algorithm.
    '''
    mos_img = np.tile(img[:, :, np.newaxis], [1, 1, 3])
    image_height, image_width = img.shape

    red_values = img[1:image_height:2, 1:image_width:2]
    mean_value = red_values.mean()
    mos_img[:, :, 0] = mean_value
    mos_img[1:image_height:2, 1:image_width:2, 0] = img[1:image_height:2, 1:image_width:2]

    blue_values = img[0:image_height:2, 0:image_width:2]
    mean_value = blue_values.mean()
    mos_img[:, :, 2] = mean_value
    mos_img[0:image_height:2, 0:image_width:2, 2] = img[0:image_height:2, 0:image_width:2]

    mask = np.ones((image_height, image_width))
    mask[0:image_height:2, 0:image_width:2] = -1
    mask[1:image_height:2, 1:image_width:2] = -1
    green_values = mos_img[mask > 0]
    mean_value = green_values.mean()

    green_channel = img
    green_channel[mask < 0] = mean_value
    mos_img[:, :, 1] = green_channel

    return mos_img


def demosaicNN(img):
    '''
    Nearest neighbor demosaicing.

    Args:
        img: np.array of size NxM.

    '''
    mos_img = np.tile(img[:, :, np.newaxis], [1, 1, 3])
    image_height, image_width = img.shape

    #Blue channel (odd rows and columns)
    blue_values = np.ones([image_height, image_width])#*(-1)
    blue_values[0:image_height:2, 0:image_width:2] = img[0:image_height:2, 0:image_width:2]

    #For every alternate row, starting from 1st row
    for i in range(0, image_height, 2):
    #For every alternate column, starting from 1st column
        for j in range(0, image_width, 2):
        #Put the same value as that of known pixel's, to the pixel at right
            if( i + 1 < image_height):
                blue_values[i+1, j] = blue_values[i, j]
        #Put the same value as that of known pixel's, to the pixel at bottom
            if( j + 1 < image_width):
                blue_values[i, j+1] = blue_values[i, j]
```

```
        #Put the same value as that of known pixel's, to the pixel at bottom right
            if( i + 1 < image_height) and ( j + 1 < image_width):
                blue_values[i+1, j+1] = blue_values[i, j]

    mos_img[:, :, 2] = blue_values

    #Red channel (even rows and columns)
    red_values = np.ones([image_height, image_width])#*(-1)
    red_values[1:image_height:2, 1:image_width:2] = img[1:image_height:2, 1:image_width:2]

    #For every alternate row, starting from 2nd row
    for i in range(1, image_height, 2):
    #For every alternate column, starting from 2nd column
        for j in range(1, image_width, 2):
        #Put the same value as that of known pixel's, to the pixel at right
            if( i + 1 < image_height):
                red_values[i+1, j] = red_values[i, j]
        #Put the same value as that of known pixel's, to the pixel at bottom
            if( j + 1 < image_width):
                red_values[i, j+1] = red_values[i, j]
        #Put the same value as that of known pixel's, to the pixel at bottom right
            if( i + 1 < image_height) and ( j + 1 < image_width):
                red_values[i+1, j+1] = red_values[i, j]

    # For the first row, copy the values from 2nd row, they are the "nearest"
    red_values[1,:] = red_values[2,:]
    # For the first column, copy the values from 2nd column, they are the "nearest"
    red_values[:,1] = red_values[:,2]

    #Make the interpolated red value matrix, the red channel of the final output image
    mos_img[:, :, 0] = red_values

    #Green Channel
    green_values = np.ones([image_height, image_width])#*(-1)

    #Odd rows have green pixel's known values at even columns
    green_values[0:image_height:2, 1:image_width:2] = img[0:image_height:2, 1:image_width:2]
    #Even rows have green pixel's known values at odd columns
    green_values[1:image_height:2, 0:image_width:2] = img[1:image_height:2, 0:image_width:2]

    #Odd rows
    for i in range(0, image_height, 2):
        #Even columns
        for j in range(1, image_width, 2):
        #Put the same value as that of known pixel's, to the pixel at left
            if(i-1 > 0):
               green_values[i-1, j] = green_values[i, j]

    #Even rows
    for i in range(1, image_height, 2):
        #Odd columns
```

```
        for j in range(0, image_width, 2):
        #Put the same value as that of known pixel's, to the pixel at right
            if(i+1< image_height):
               green_values[i+1, j] = green_values[i, j]

    mos_img[:, :, 1] = green_values

    return  mos_img


def demosaicLinear(img):
    '''
    Linear Interpolation
    Args:
        img: np.array of size NxM.
    '''
    mos_img = np.tile(img[:, :, np.newaxis], [1, 1, 3])
    image_height, image_width = img.shape

    #Blue
    blue_values = np.zeros((image_height, image_width))
    blue_values[0:image_height:2, 0:image_width:2] = 1
    blue = mos_img[:, :, 2] * blue_values

    #Red
    red_values = np.zeros((image_height, image_width))
    red_values[1:image_height:2, 1:image_width:2] = 1
    red = mos_img[:, :, 0] * red_values

    #Green
    green = mos_img[:,:,1] - (blue + red)

    #Interpolation
    interpolateB  = np.argwhere(blue!=0)

    blue = np.hstack((blue, np.zeros((image_height, 2))))
    blue = np.vstack((blue, np.zeros((2, image_width + 2))))

    array1= np.zeros_like(blue)
    array2= np.zeros_like(blue)
    array3= np.zeros_like(blue)

    for x in interpolateB:
        xh, xw = x[0], x[1]
        array1[xh+1, xw+1] = np.divide((blue[xh,xw]+blue[xh,xw+2]+blue[xh+2,xw]+
    blue[xh+2,xw+2]), 4)
        array2[xh+1, xw] = np.divide((blue[xh,xw]+blue[xh+2,xw]), 2)
        array3[xh, xw+1] = np.divide((blue[xh,xw]+blue[xh,xw+2]), 2)

    blue = array1 + array2 + array3 + blue

    blue[:,image_width - 1] = blue[:, image_width - 2]
    blue[image_height-1,:]= blue[image_height-2,:]
    mos_img[:, :, 2] = blue[0:image_height, 0:image_width]

    interpolateR = np.argwhere(red!=0)

    red = np.hstack((red, np.zeros((image_height, 2))))
```

```
    red = np.vstack((red, np.zeros((2,image_width + 2))))

    array1= np.zeros_like(red)
    array2= np.zeros_like(red)
    array3= np.zeros_like(red)

    for x in interpolateR :
        xh, xw = x[0], x[1]
        array1[xh+1, xw+1] = np.divide((red[xh,xw]+red[xh,xw+2]+red[xh+2,xw]+red[xh+2,xw+2]), 4)
        array2[xh+1, xw] = np.divide((red[xh,xw]+red[xh+2,xw]), 2)
        array3[xh, xw+1] = np.divide((red[xh,xw]+red[xh,xw+2]), 2)

    red = array1 + array2 + array3 + red
    red[0,:] = red[1,:]
    red[:,0] = red[:,1]
    mos_img[:, :, 0] = red[0:image_height, 0:image_width]

    interpolateG = np.argwhere(green==0)
    green = np.hstack((green, np.zeros((image_height,1))))
    green = np.vstack((green, np.zeros((1,image_width+1))))

    for x in interpolateG:
        xh, xw = x[0], x[1]
        if((xh!=image_height and xw!=image_width)):
            green[xh,xw]= np.divide((green[xh,xw-1] + green[xh-1,xw] + green[xh+1,xw]+ green[xh,xw+1]),4)

        if (xh == 0 and xw!=image_width) or (xh!=image_height and xw==0) or (xh==image_height-1 and xw!=image_width)or (xh!=image_height and xw==image_width-1):
            green[xh,xw]=np.divide((green[xh,xw-1]+green[xh-1,xw]+green[xh+1,xw]+green[xh,xw+1]), 3)

        if (xh in [0,image_height - 1]) and (xw in [0, image_width - 1]):
            green[xh,xw]=np.divide((green[xh,xw-1] + green[xh-1,xw] + green[xh+1,xw] + green[xh,xw+1]), 2)

    mos_img[:, :, 1] = green[0:image_height, 0:image_width]

    return mos_img



def demosaicAdagrad(img):
    '''adaptive gradient interpolation demosaicing.

    Args:
        img: np.array of size NxM.
    '''
    mos_img = np.tile(img[:, :, np.newaxis], [1, 1, 3])
    image_height, image_width = img.shape

    blue_values = np.zeros((image_height, image_width))
    blue_values[0:image_height:2, 0:image_width:2] = 1
    blue = mos_img[:, :, 2] * blue_values

    red_values = np.zeros((image_height, image_width))
```

```
    red_values[1:image_height:2, 1:image_width:2] = 1
    red = mos_img[:, :, 0] * red_values

    green = mos_img[:,:,1] - (blue + red)

    #Interpolation
    interpolateB  = np.argwhere(blue!=0)
    blue = np.hstack((blue, np.zeros((image_height, 2))))
    blue = np.vstack((blue, np.zeros((2, image_width + 2))))

    array1= np.zeros_like(blue)
    array2= np.zeros_like(blue)
    array3= np.zeros_like(blue)

    for x in interpolateB:
        xh, xw = x[0], x[1]
        array1[xh+1, xw+1] = np.divide((blue[xh,xw]+blue[xh,xw+2]+blue[xh+2,xw]+blue[xh+2,xw+2]), 4)
        array2[xh+1, xw] = np.divide((blue[xh,xw]+blue[xh+2,xw]), 2)
        array3[xh, xw+1] = np.divide((blue[xh,xw]+blue[xh,xw+2]), 2)

    blue = array1 + array2 + array3 + blue

    blue[:,image_width-1]= blue[:,image_width-2]
    blue[image_height-1,:]= blue[image_height-2,:]
    mos_img[:, :, 2] = blue[0:image_height, 0:image_width]

    interpolateR = np.argwhere(red!=0)
    red = np.hstack((red, np.zeros((image_height, 2))))
    red = np.vstack((red, np.zeros((2,image_width + 2))))

    array1= np.zeros_like(red)
    array2= np.zeros_like(red)
    array3= np.zeros_like(red)

    for x in interpolateR :
        xh, xw = x[0], x[1]
        array1[xh+1, xw+1] = np.divide((red[xh,xw]+red[xh,xw+2]+red[xh+2,xw]+red[xh+2,xw+2]), 4)
        array2[xh+1, xw] = np.divide((red[xh,xw]+red[xh+2,xw]), 2)
        array3[xh, xw+1] = np.divide((red[xh,xw]+red[xh,xw+2]), 2)

    red = array1 + array2 + array3 + red
    red[0,:] = red[1,:]
    red[:,0] = red[:,1]
    mos_img[:, :, 0] = red[0:image_height, 0:image_width]

    interpolateG = np.argwhere(green==0)
    green = np.hstack((green, np.zeros((image_height,1))))
    green = np.vstack((green, np.zeros((1,image_width+1))))

    for x in interpolateG:
        xh, xw = x[0], x[1]
        if((xh!=image_height and xw!=image_width)):
            if abs(green[xh,xw-1]- green[xh,xw+1]) < abs(green[xh-1,xw]-green[xh+1,xw]):
                green[xh,xw]= np.divide((green[xh,xw-1] + green[xh,xw+1]),2)
            else:
```

```
                green[xh,xw]=np.divide((green[xh-1,xw] + green[xh+1,xw]), 2)

    mos_img[:, :, 1] = green[0:image_height, 0:image_width]

    return mos_img


def demosaicDivide(img):
    '''
    Tranformed Color Space Divide demosaicing

    Args:
        img: np.array of size NxM.
    '''
    #Dimensions of Filter values
    image_height, image_width = img.shape

    #Green Channel Interpolation by calling Adaptive Gradient
    mos_img = demosaicAdagrad(img)
    greenChannel = mos_img[:,:,1]

    #Replacing all zeros by 1e-6 to tackle divide by zero error
    greenChannel[np.where(greenChannel==0)] = 1e-6

    #Blue Filter Values
    blue_values = np.zeros([image_height, image_width])
    blue_values[0:image_height:2, 0:image_width:2] = img[0:image_height:2, 0:image_width:2]

    #Divide Blue Filter values with Interpolated Green Channel
    blue_values = blue_values/greenChannel

    #Red Filter Values
    red_values = np.zeros([image_height, image_width])
    red_values[1:image_height:2, 1:image_width:2] = img[1:image_height:2, 1:image_width:2]

    #Divide Red Filter values with Interpolated Green Channel
    red_values = red_values/greenChannel

    #Making new Green Filter
    green_values = np.zeros([image_height, image_width])
    green_values[0:image_height:2, 1:image_width:2] = img[0:image_height:2, 1:image_width:2]
    green_values[1:image_height:2, 0:image_width:2] = img[1:image_height:2, 0:image_width:2]

    #RGB Channels
    rgb = red_values + green_values + blue_values

    #Finding Interpolated RGB Channels by calling Adaptive Gradient
    mos_img = demosaicAdagrad(rgb)

    #Multiply Blue and Red Filter values with Interpolated Green Channel
    mos_img[:, :, 2] = np.multiply(mos_img[:, :, 2] , greenChannel)
    mos_img[:, :, 0] = np.multiply(mos_img[:, :, 0] , greenChannel)

    #Handling all outlier values greater than 1 in mos_img
```

```
    mos_img[np.where(mos_img>1)] = 1
    return mos_img

def demosaicLoglDivide(img):
    '''
    Tranformed Color Space Divide and Log demosaicing

    Args:
        img: np.array of size NxM.
    '''
    mos_img = np.tile(img[:, :, np.newaxis], [1, 1, 3])

    #Dimensions of the filter values
    image_height, image_width = img.shape

    #Green Channel Interpolation by calling Adaptive Gradient
    mos_img = demosaicAdagrad(img)
    greenChannel = mos_img[:,:,1]

    #Replacing all zeros by 1e-6 to tackle divide by zero error
    greenChannel[np.where(greenChannel == 0)] = 1e-6

    #Blue Filter Values
    blue_values = np.zeros([image_height, image_width])
    blue_values[0:image_height:2, 0:image_width:2] = img[0:image_height:2, 0:image_width:2]

    #Divide Blue Filter values with Interpolated Green Channel
    #Add 1 inside log to tackle log(0) edge case
    blue_values = np.log(1+ blue_values/greenChannel)

    #Get Red Filter Values
    red_values = np.zeros([image_height, image_width])
    red_values[1:image_height:2, 1:image_width:2] = img[1:image_height:2, 1:image_width:2]

    #Divide Red Filter values with Interpolated Green Chzannel
    #Add 1 inside log to tackle log(0) edge case
    red_values = np.log(1+ red_values/greenChannel)

    #Making new Green Filter
    green_values = np.zeros([image_height, image_width])
    green_values[0:image_height:2, 1:image_width:2] = img[0:image_height:2, 1:image_width:2]
    green_values[1:image_height:2, 0:image_width:2] = img[1:image_height:2, 0:image_width:2]

    #RGB Channels
    rgb = red_values + green_values + blue_values

    #Finding Interpolated RGB Channels by calling Adaptive Gradient
    mos_img = demosaicAdagrad(rgb)

    #Multiply Blue and Red Filter values with Interpolated Green Channel
    #Take exponential which is the inverse of log
    #Subtract 1 because 1 was added before taking log of the ratio
    mos_img[:, :, 2] = np.multiply(np.exp(mos_img[:, :, 2]-1) , greenChannel)
    mos_img[:, :, 0] = np.multiply(np.exp(mos_img[:, :, 0]-1) , greenChannel)
```

```
    #Handling all outlier values greater than 1 in mos_img
    mos_img[np.where(mos_img > 1)] = 1
    return mos_img
```

## 2.2 Output

Nearest-neighbour interpolation.

Figure 13: Nearest Neighbor

Figure 14: Nearest Neighbor

Figure 15: Nearest Neighbor

Figure 16: Nearest Neighbor

Figure 17: Nearest Neighbor

Figure 18: Nearest Neighbor

Figure 19: Nearest Neighbor

Figure 20: Nearest Neighbor

Figure 21: Nearest Neighbor

Figure 22: Nearest Neighbor

Linear interpolation

Figure 23: Linear

Figure 24: Linear

Figure 25: Linear

Figure 26: Linear

Figure 27: Linear

Figure 28: Linear

Figure 29: Linear

Figure 30: Linear

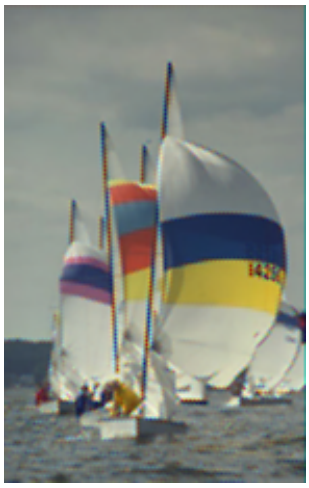


Figure 31: Linear

Figure 32: Linear

Adaptive gradient interpolation

Figure 33: Adaptive gradient interpolation

Figure 34: Adaptive gradient interpolation

Figure 35: Adaptive gradient interpolation

Figure 36: Adaptive gradient interpolation

Figure 37: Adaptive gradient interpolation

Figure 38: Adaptive gradient interpolation

Figure 39: Adaptive gradient interpolation

Figure 40: Adaptive gradient interpolation

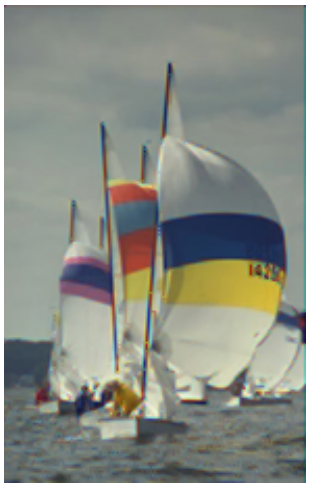
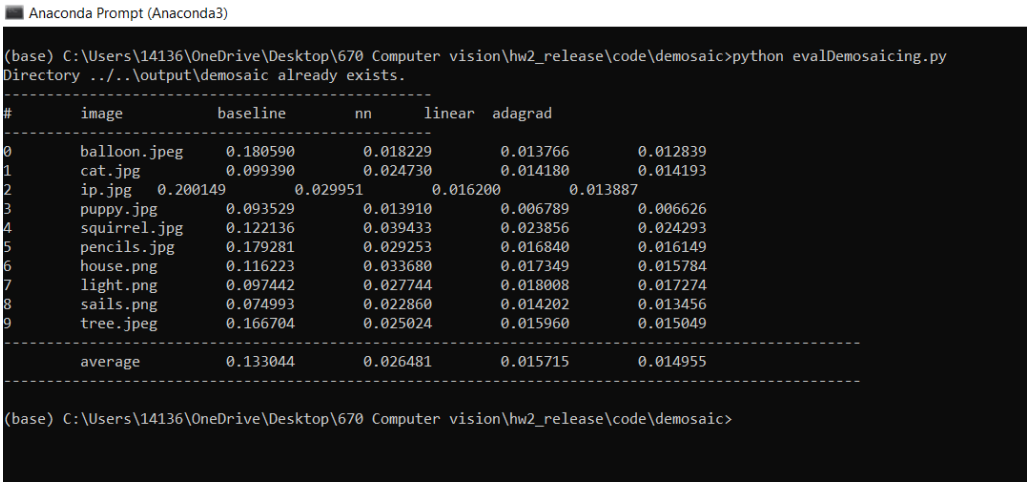


Figure 41: Adaptive gradient interpolation

Figure 42: Adaptive gradient interpolation



```
Anaconda Prompt (Anaconda3)

(base) C:\Users\14136\OneDrive\Desktop\670 Computer vision\hw2_release\code\demosaic>python evalDemosaicing.py
Directory ../..\output\demosaic already exists.
--------------------------------------------------
#        image           baseline        nn      linear  adagrad
--------------------------------------------------
0        balloon.jpeg     0.180590        0.018229        0.013766        0.012839
1        cat.jpg          0.099390        0.024730        0.014180        0.014193
2        ip.jpg   0.200149        0.029951        0.016200        0.013887
3        puppy.jpg        0.093529        0.013910        0.006789        0.006626
4        squirrel.jpg     0.122136        0.039433        0.023856        0.024293
5        pencils.jpg      0.179281        0.029253        0.016840        0.016149
6        house.png        0.116223        0.033680        0.017349        0.015784
7        light.png        0.097442        0.027744        0.018008        0.017274
8        sails.png        0.074993        0.022860        0.014202        0.013456
9        tree.jpeg        0.166704        0.025024        0.015960        0.015049
--------------------------------------------------------------------------------------------------
         average          0.133044        0.026481        0.015715        0.014955
--------------------------------------------------------------------------------------------------

(base) C:\Users\14136\OneDrive\Desktop\670 Computer vision\hw2_release\code\demosaic>
```

Figure 43: Interpolation Output

## 2.3 Description

**Nearest-Neighbor Interpolation**
Here, we interpolate each unknown pixel's value by simply making it of the value of it's closet neighbor whose value we know from the filter pattern.

We interpolate the "missing" pixels from each color channel with the nearest pixel.

We can see the blue channel mosaic has repetitive grid of 2 x 2 grids where the 2 x 2 grid is always in pattern like a mask, where one means the value is already available in the blue channel of Bayer Filer, and we need to interpolate values for the pixels which are at 0 in the mask.
Hence, we can just copy the (1, 1) pixel's value of all the 2 x 2 grids to the pixels at (1, 2) [bottom]; (2, 1) [right] and (2, 2) [bottom right].

Same concept applies for Red Channel as follows: It follows the same basic mask concept as blue channel, but a minor change is that the position of the blue cells demand us to interpolate different unknown cells than we did in red mask's case. We can see the red channel mosaic has repetitive grid of 2 x 2 grids because red cells are starting at (2; 2). Hence, we can just copy the (2, 2) pixel's value of all the 2 x 2 grids to the pixels at (1, 2) [top]; (2, 1) [left] and (1, 1) [top left].

OR we can use the exact same method of interpolating bottom, right and bottom right pixels by assuming that the known pixel value is at (1, 1) of the 2 x 2 grid. That will leave the 1st row and column "un-interpolated". So, we just interpolate them in the end with the nearest values from 2nd row and 2nd column respectively.

The Green channel the algorithm is a bit different, as there are only two missing colors, instead of three. For each block of 2 x 2 pixels, it fills the first "missing" pixel with the same value as the first pixel and the second "missing" pixel with the same value as the second pixel.

We can see the green channel mosaic has repetitive grid of 2 x 2 grids where We interpolate pixel value at (1, 2) with the value of pixel at (1, 1) and the pixel value at (2, 1) with the value of pixel at (2, 2).

**Linear Gradient Interpolation**

Linear interpolation takes average value of all neighbor pixels.
We will have to make 3 kinds of calculation for Red and Blue cells, we focus on Blue cells, the logic behind blue cells is very similar:
1. Diagonal Cross Interpolation: The "missing" pixel has four neighbors cells around it. The average value of those four pixels will be the new value for the "missing" pixel.

2. Horizontal Interpolation The "missing" pixel has a cell at its left and a cell at its right. Similarly to the previous case, the average value of those two pixels will be the new value for the "missing" pixel.

3. Vertical Interpolation: For the last case, the "missing" pixel has two cells, one at its top and one at its bottom. The average value of those two pixels will be the new value for the "missing" pixel.

Green Channel Linear Demosaicing:
CROSS INTERPOLATION: In this case, the "missing" pixel has four neighbors cells around it. The average value of those four pixels will be the new value for the "missing" pixel.

**Adaptive Gradient Interpolation**
First, we will use Adaptive Gradient for Green Channel and use Linear Interpolation for Red and Blue Channels. Later, we will try to improve the Red and Blue Interpolation techniques.

Green Channel Adaptive Gradient Interpolation Demosaicing

This takes average of the pixels based on the structure of neighbors.
**If |top-pixel - botton-pixel| > |left-pixel - right-pixel|**
Then interpolated-pixel = (left-pixel + right-pixel)/2
**Else interpolated-pixel = (top-pixel + bottom-pixel)/2**

From the output of baseline, nearest neighbor, linear and adaptive gradient algorithms for each image, We can see that linear and adaptive algorithms give a very near match to the original images.

The best algorithm is the adaptive algorithm where red and blue channels were just linearly interpolated. That achieved an average error of as low as 0.014955

Linear interpolation also performed good as achieved error of only 0.015715.

Nearest Neighbor didn't perform as good (although much better than baseline) with the error if 0.026481. Because it's just a copy of its one neighbor, which might not be accurate for every case.

# 3 Texture synthesis

## 3.1 Random tile

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.
Consideration was given for not selecting a random co-ordinate pair in a way that might lead to overrunning the input image bounds and hence, not having the exact same size as the pre-decided *tilesize*.

If it was the case, simply some other random co-ordinate pair were chosen which was within the bounds
Following shows the result of random sampling for four tile sizes - 5, 15, 20, 30 and 40.

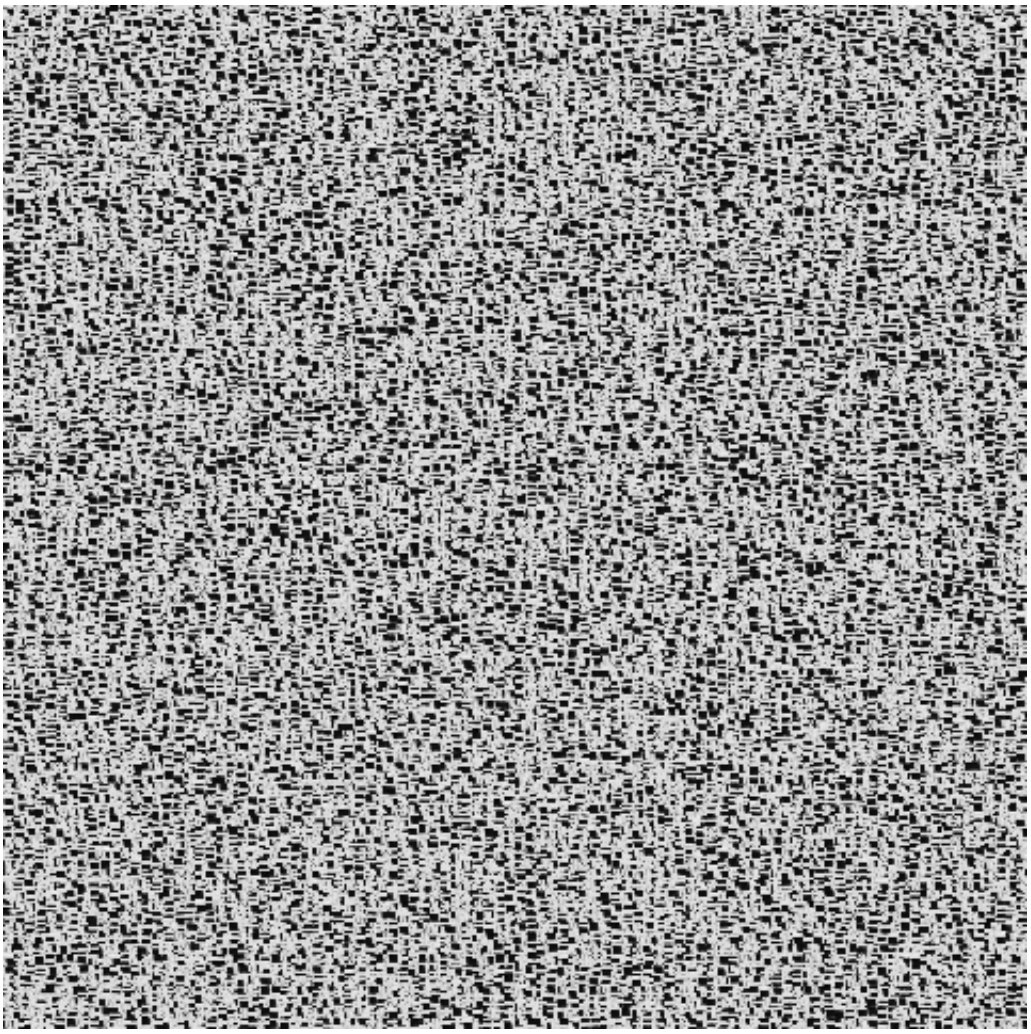


Figure 44: D20.png synthesized with tile size 5

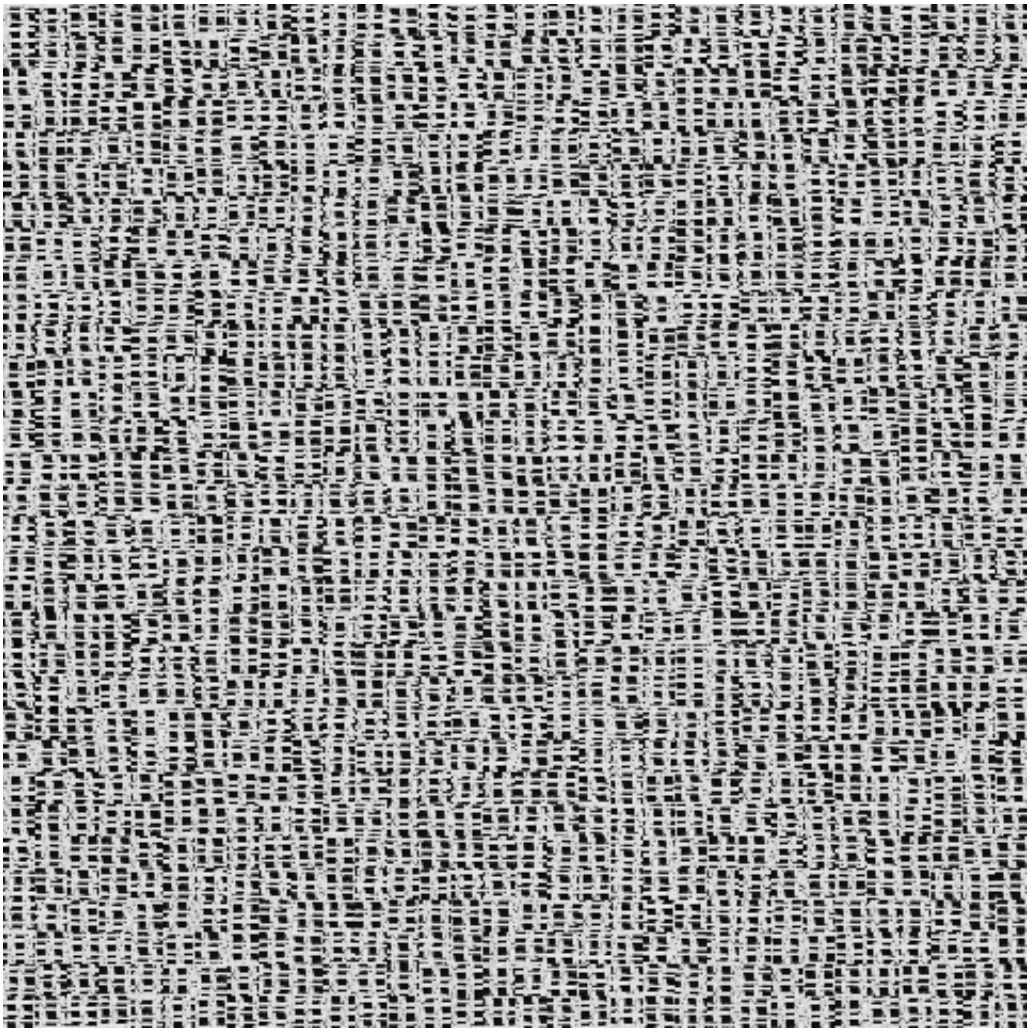


Figure 45: D20.png synthesized with tile size 15

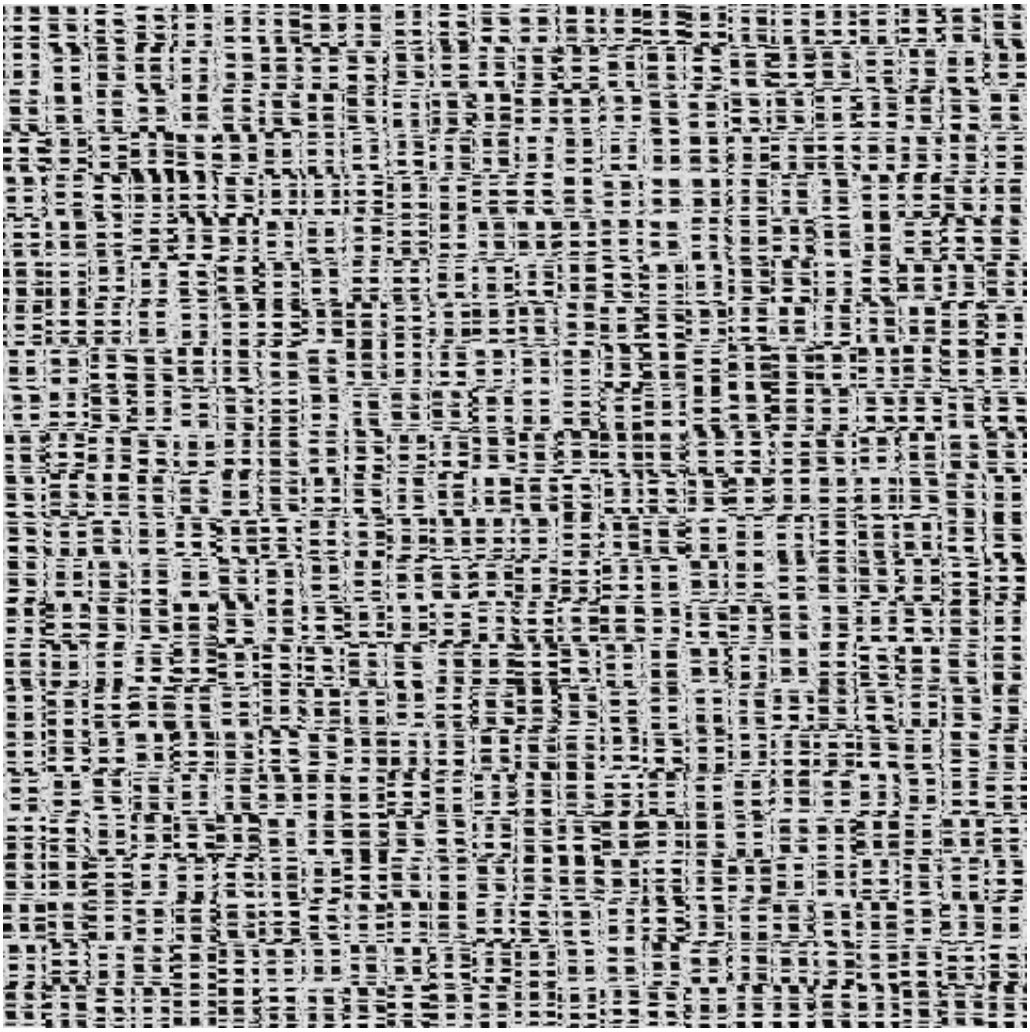


Figure 46: D20.png synthesized with tile size 20

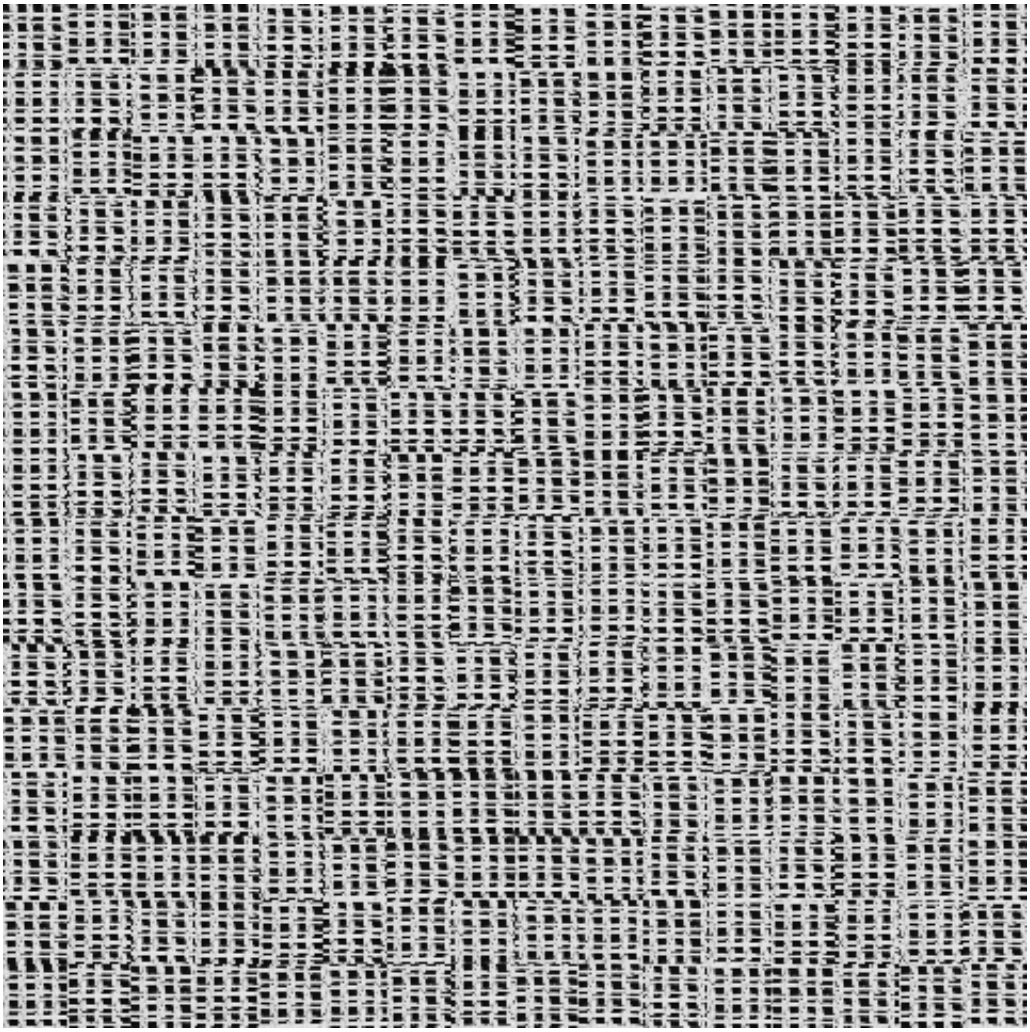


Figure 47: D20.png synthesized with tile size 30

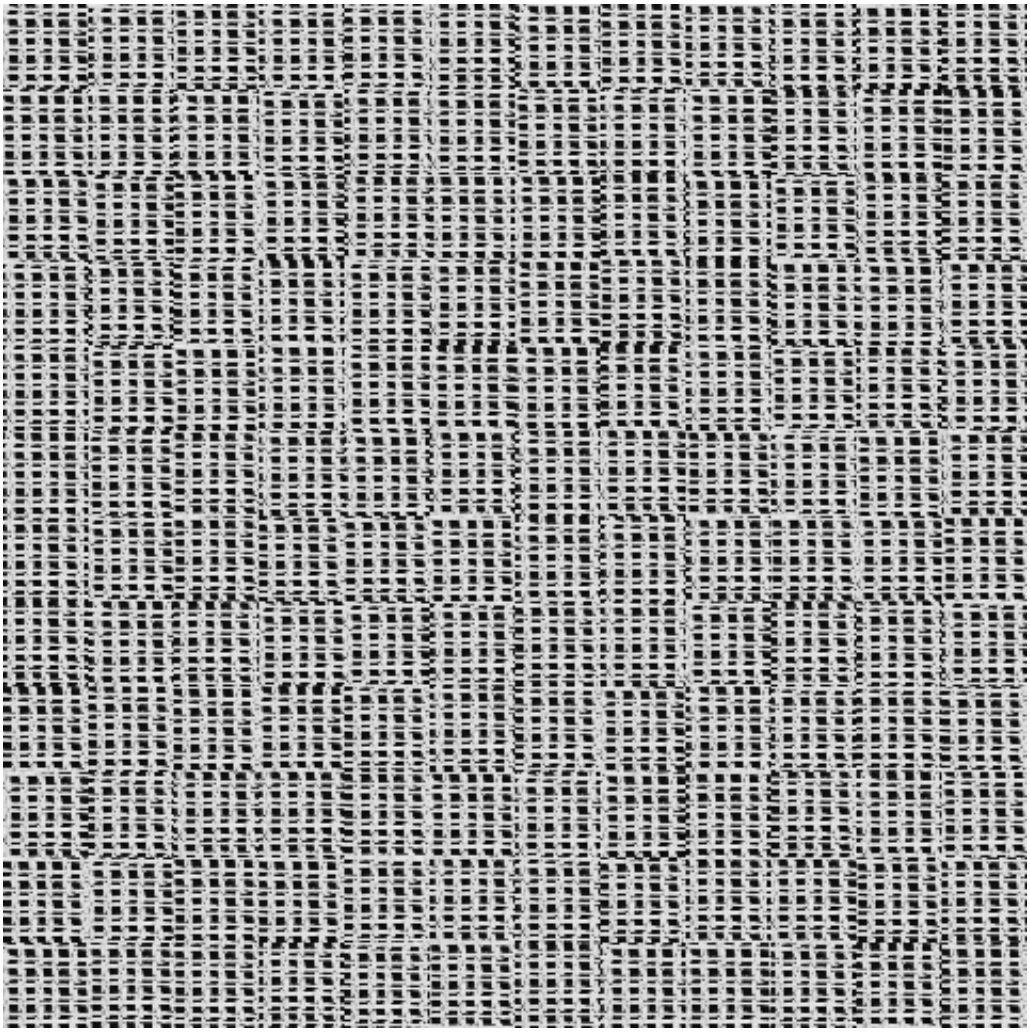


Figure 48: D20.png synthesized with tile size 40

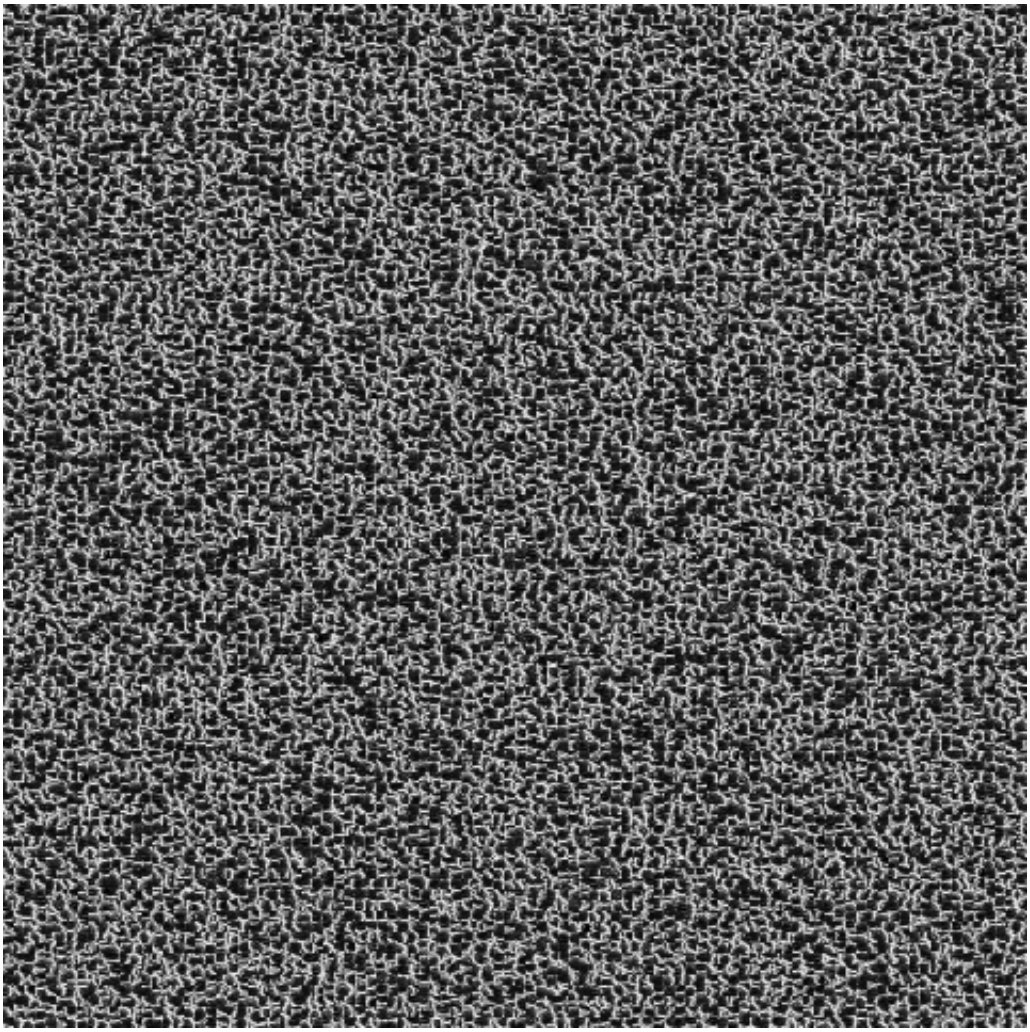


Figure 49: Texture2.bmp synthesized with tile size 5

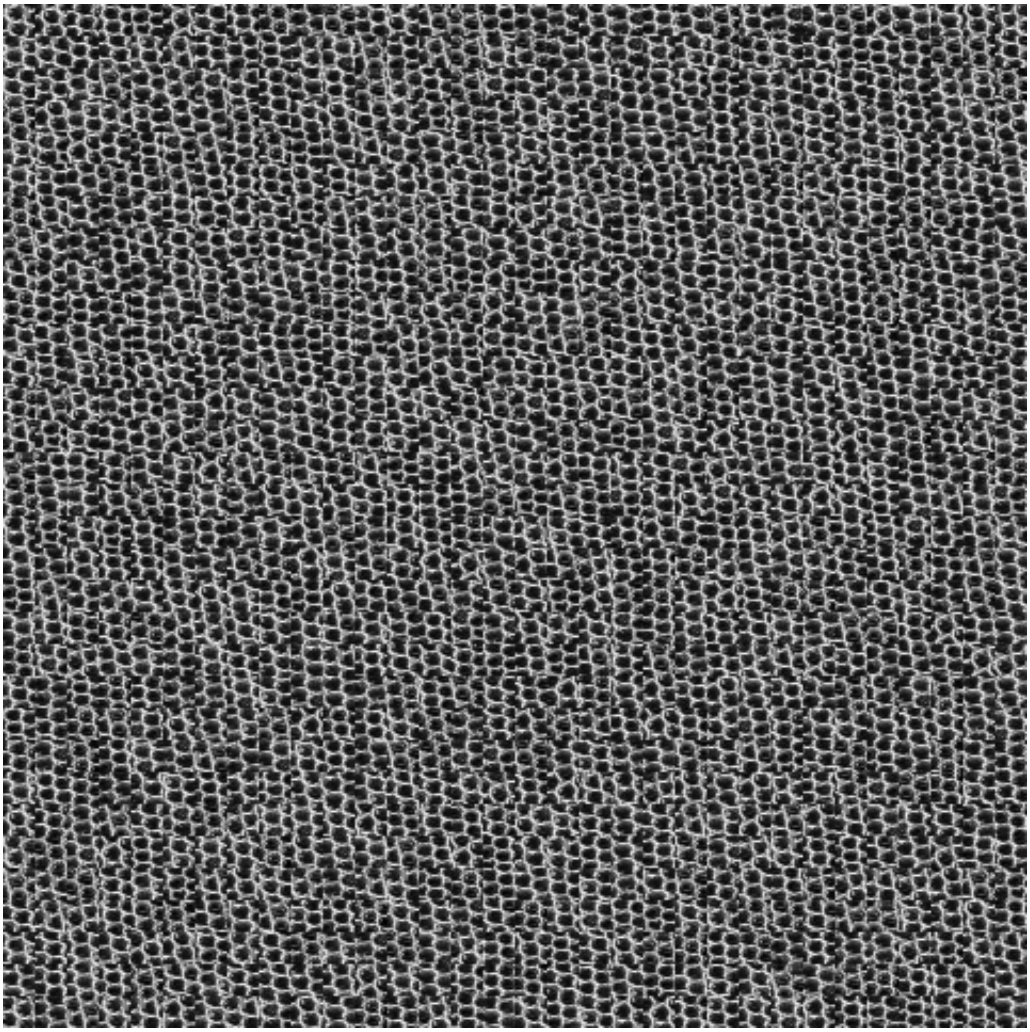


Figure 50: Texture2.bmp synthesized with tile size 15

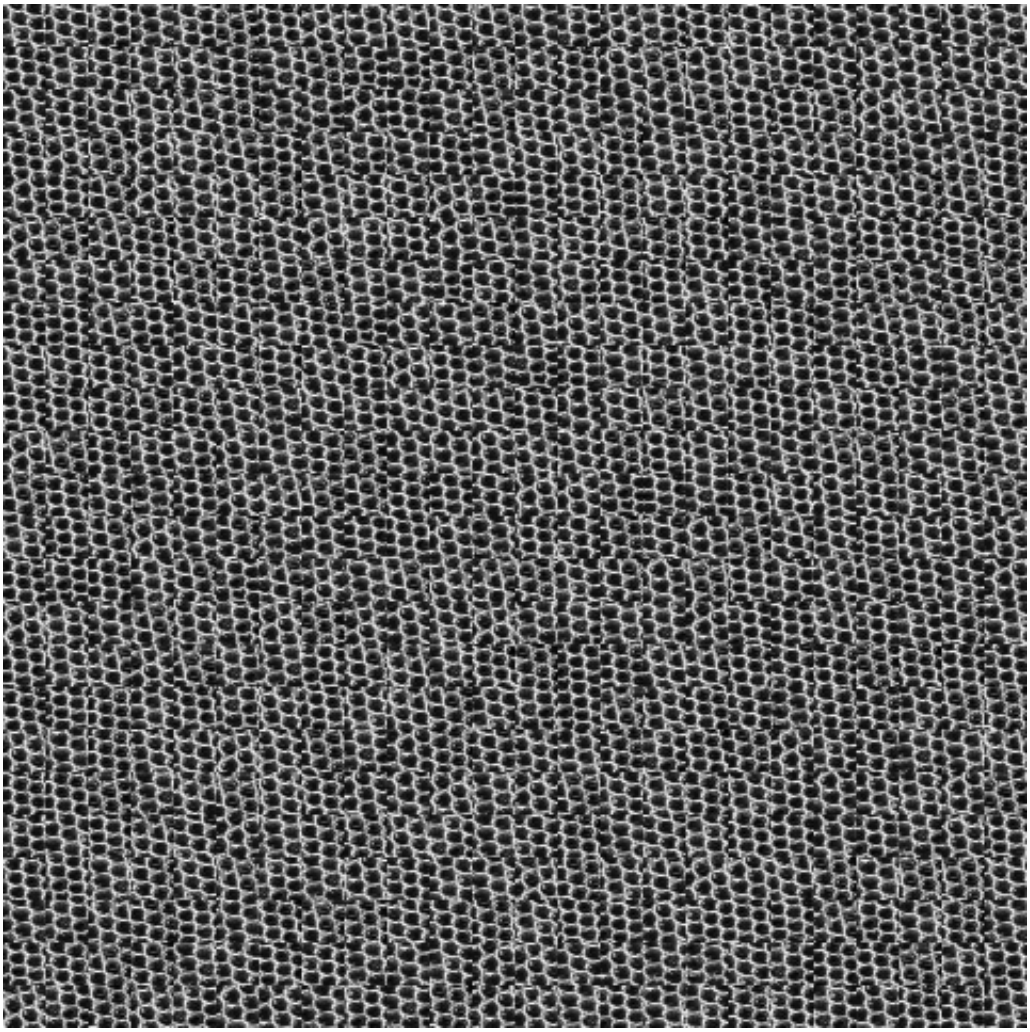


Figure 51: Texture2.bmp synthesized with tile size 20

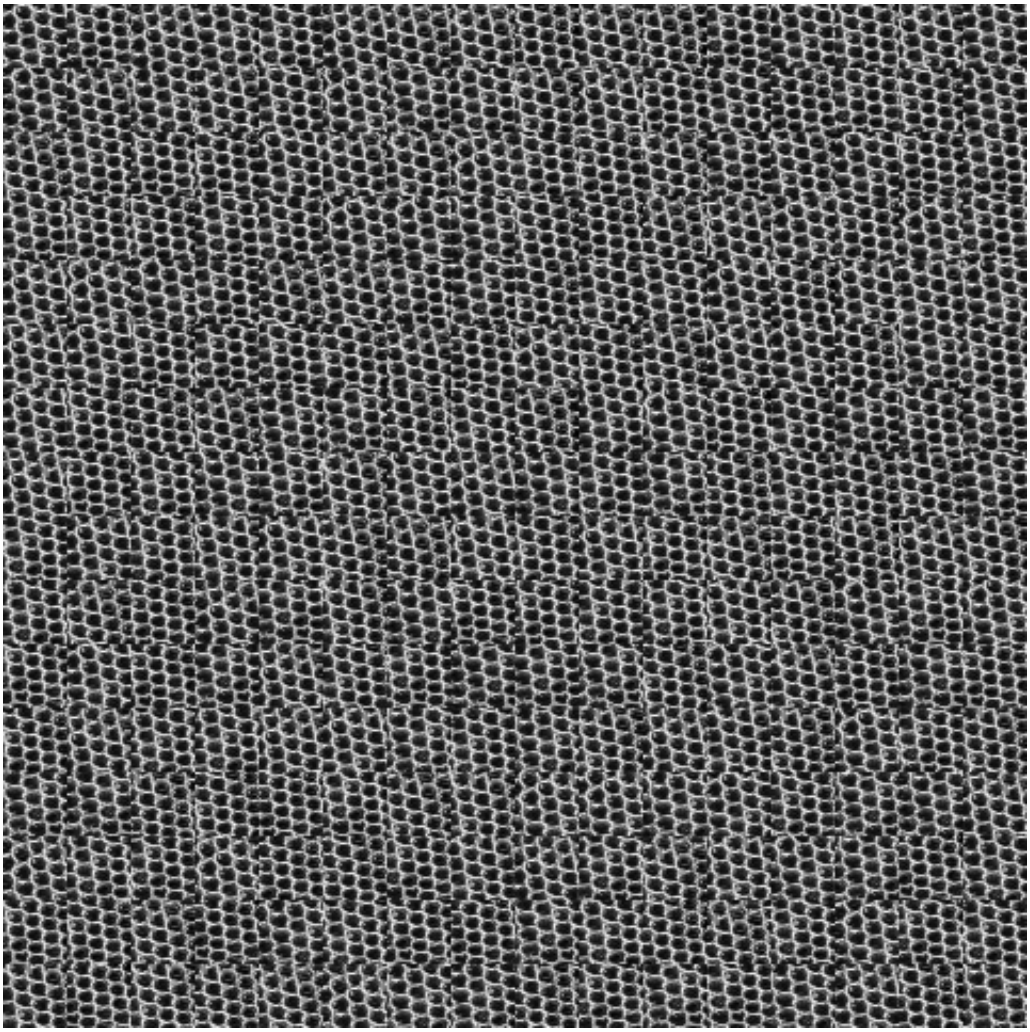


Figure 52: Texture2.bmp synthesized with tile size 30

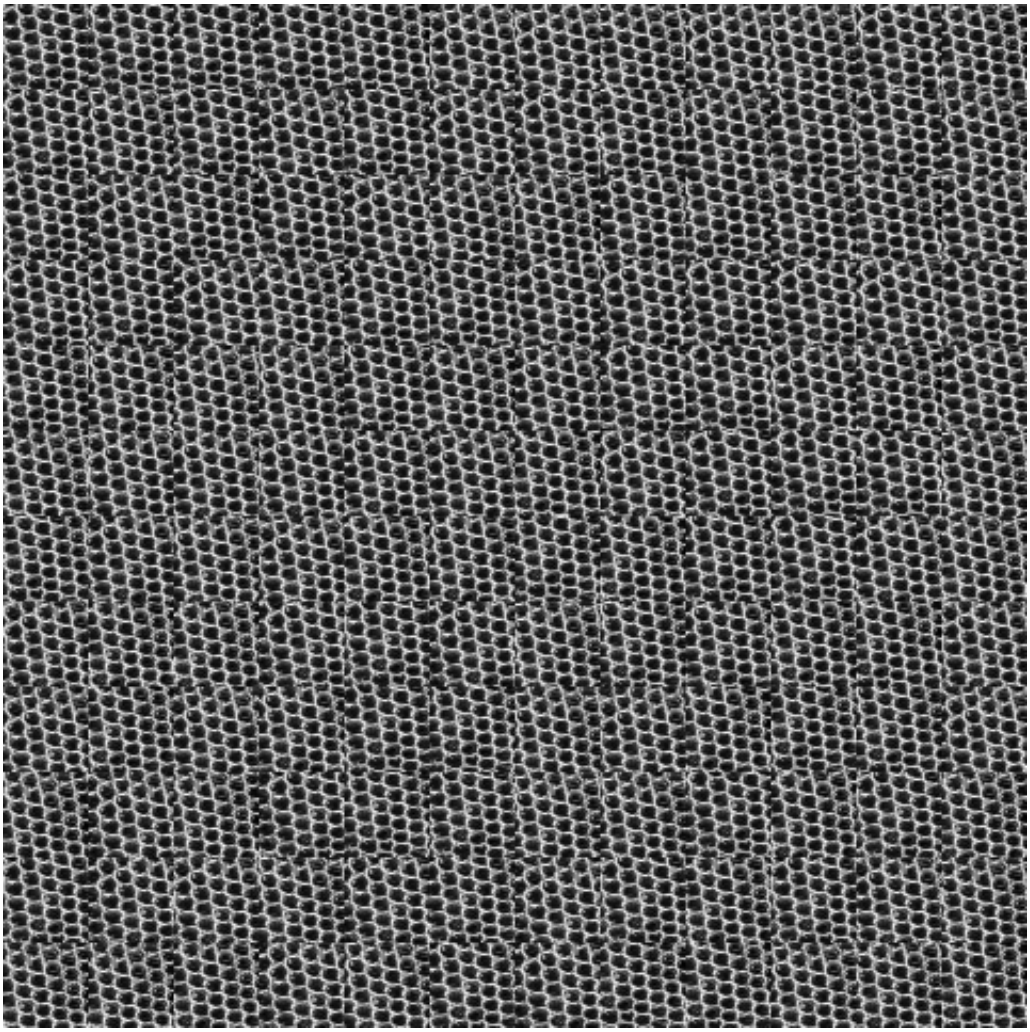


Figure 53: Texture2.bmp synthesized with tile size 40

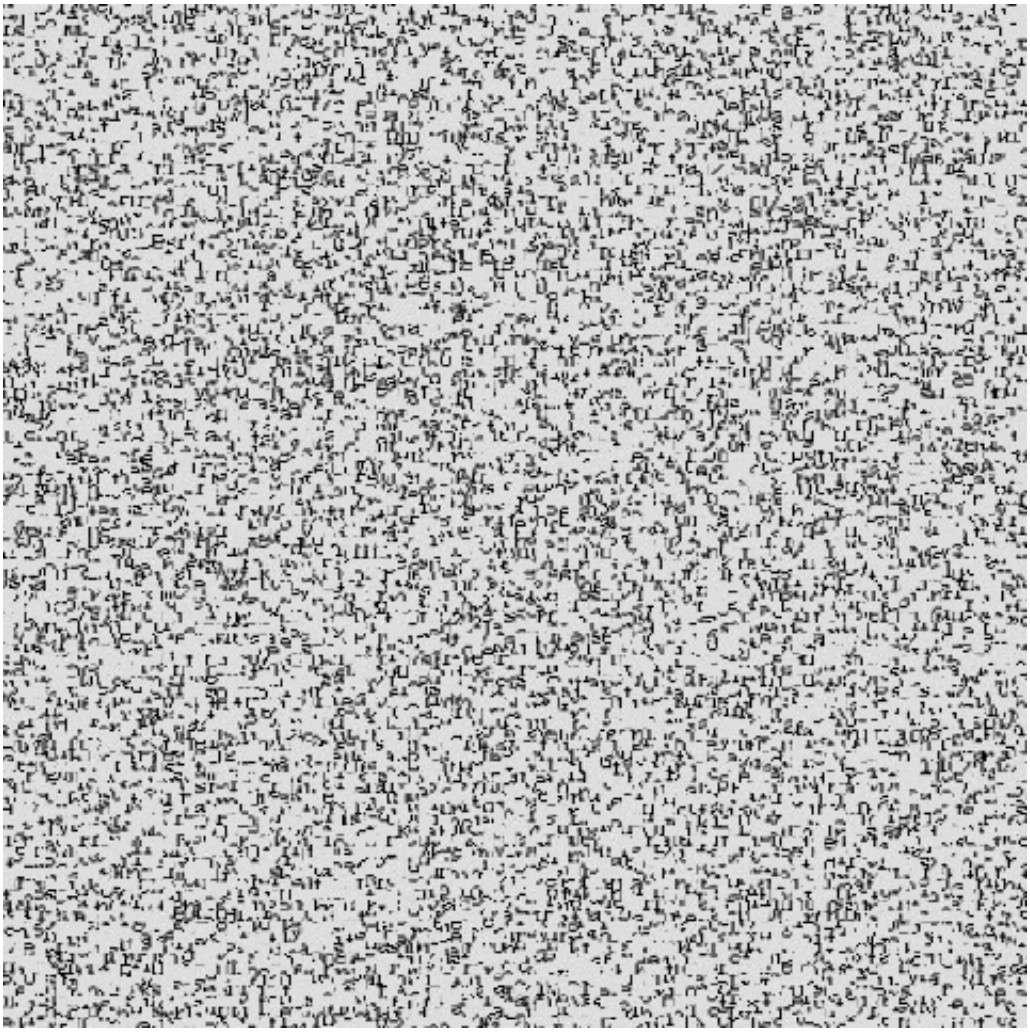


Figure 54: english.jpg synthesized with tile size 5

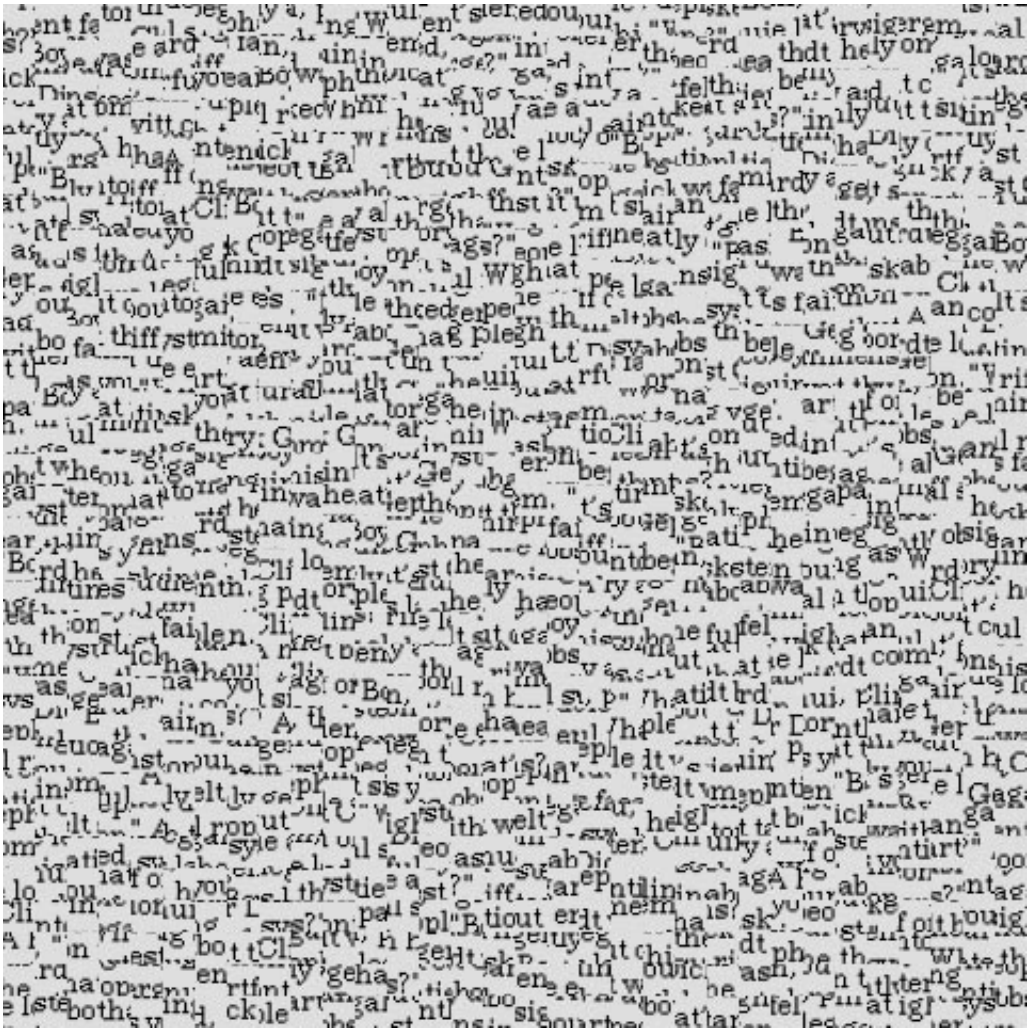


Figure 55: english.jpg synthesized with tile size 15

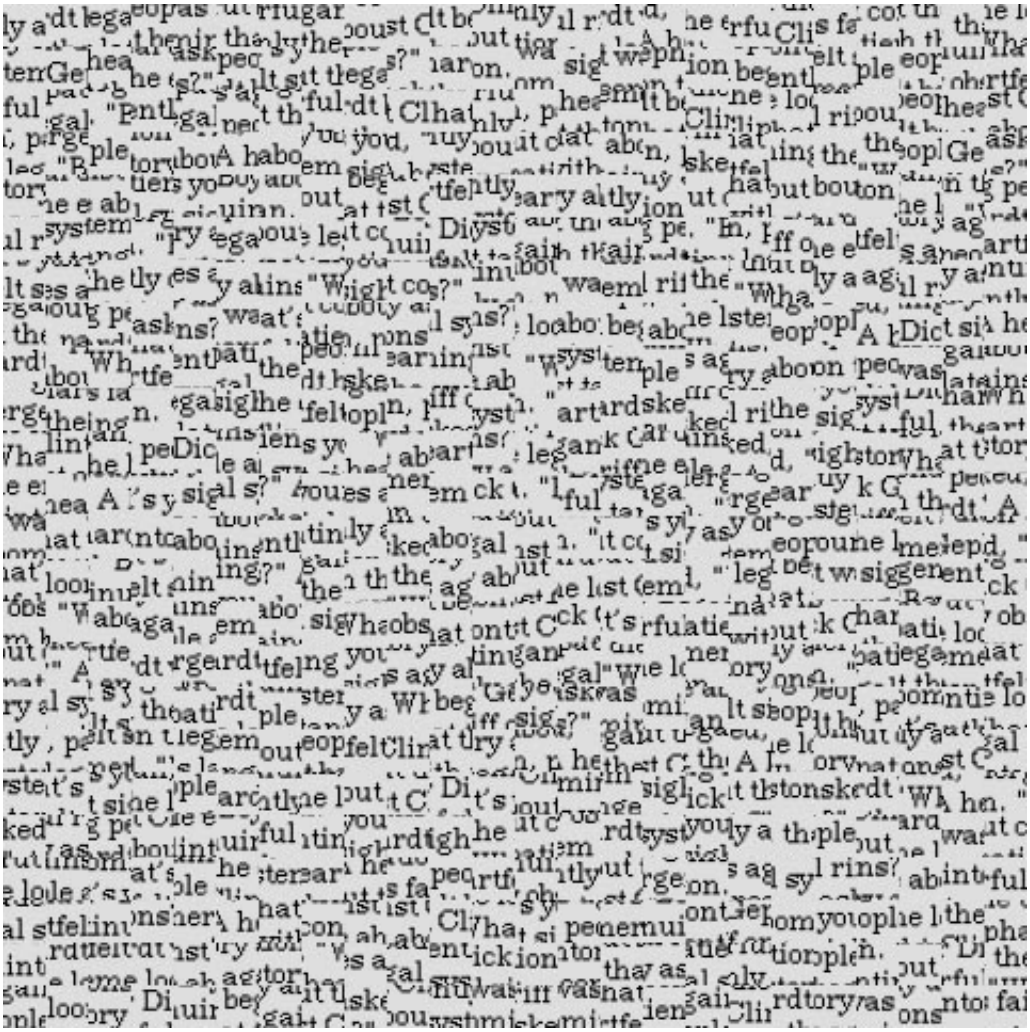


Figure 56: english.jpg synthesized with tile size 20

Figure 57: english.jpg synthesized with tile size 30

Figure 58: english.jpg synthesized with tile size 40

```
(base) C:\Users\14136\OneDrive\Desktop\670 Computer vision\hw2_release\code>python evalTextureSynth.py
RandomTextureSynthesis5x5_D20.png, Runtime =0.020000
RandomTextureSynthesis15x15_D20.png, Runtime =0.003000
RandomTextureSynthesis20x20_D20.png, Runtime =0.002007
RandomTextureSynthesis30x30_D20.png, Runtime =0.001000
RandomTextureSynthesis40x40_D20.png, Runtime =0.001000
RandomTextureSynthesis5x5_Texture2.png, Runtime =0.019994
RandomTextureSynthesis15x15_Texture2.png, Runtime =0.003000
RandomTextureSynthesis20x20_Texture2.png, Runtime =0.002230
RandomTextureSynthesis30x30_Texture2.png, Runtime =0.001999
RandomTextureSynthesis40x40_Texture2.png, Runtime =0.001000
RandomTextureSynthesis5x5_english.png, Runtime =0.021001
RandomTextureSynthesis15x15_english.png, Runtime =0.022000
RandomTextureSynthesis20x20_english.png, Runtime =0.001996
RandomTextureSynthesis30x30_english.png, Runtime =0.002000
RandomTextureSynthesis40x40_english.png, Runtime =0.001010

(base) C:\Users\14136\OneDrive\Desktop\670 Computer vision\hw2_release\code>
```

Figure 59: Random Synthesis Runtime Analysis with outputsize = 480 and tile size = 5, 15, 20, 30 and 40

As observed from the output images certainly a better method is needed 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.

## 3.2 Efros and Leung Approach

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 logical structure as possible and produces good results for a wide variety of synthetic and real-world textures. Pixel intensities are inserted based on existing nearby pixel values. The distribution of a value of a pixel is conditioned on its neighbors alone.

**Effect of various window sizes – 5, 7, 11, 15.**

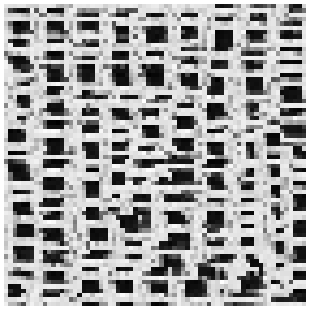


Figure 60: D20.png window size 5

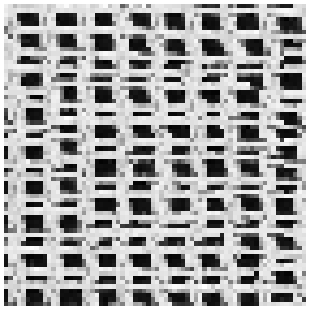


Figure 61: D20.png synthesized with window size 7

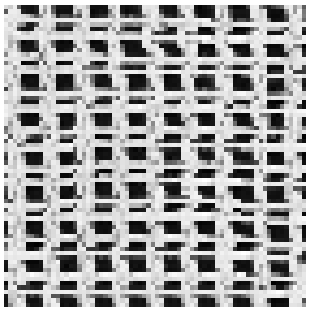


Figure 62: D20.png synthesized with window size 11

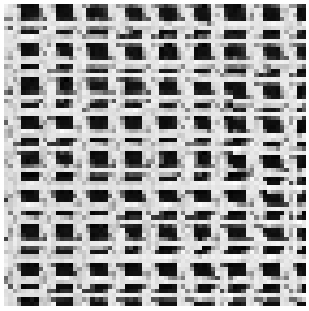


Figure 63: D20.png synthesized with window size 15

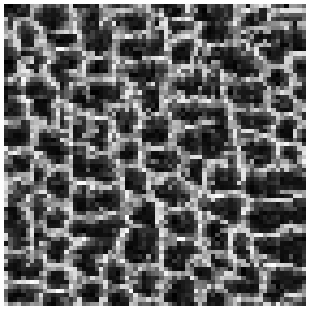


Figure 64: Texture2.bmp window size 5

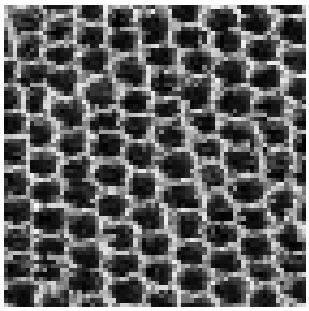


Figure 65: Texture2.bmp synthesized with window size 7

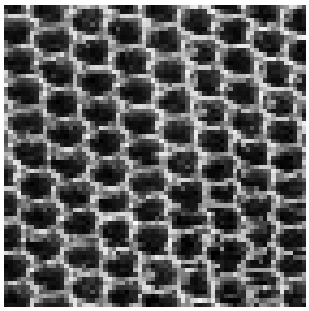


Figure 66: Texture2.bmp synthesized with window size 11

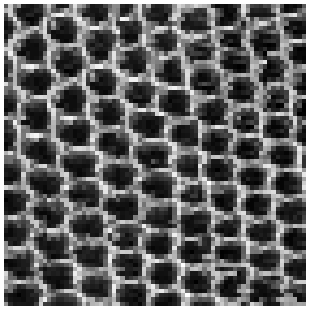


Figure 67: Texture2.bmp synthesized with window size 15

Figure 68: english.jpg window size 5

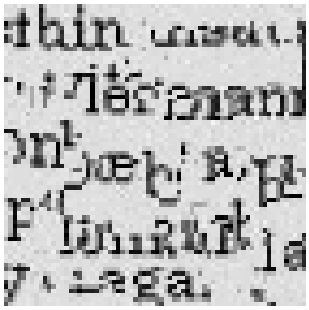


Figure 69: english.jpg synthesized with window size 7

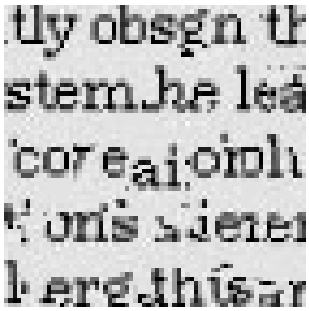


Figure 70: english.jpg synthesized with window size 11

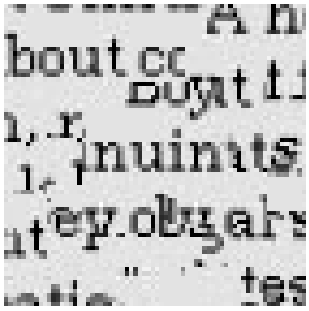


Figure 71: english.jpg synthesized with window size 15

```
(base) C:\Users\14136\OneDrive\Desktop\670 Computer vision\hw2_release\code>python evalTextureSynth.py
EfrosLeungTextureSynthesis5x5_D20.png, Runtime =33.273540
EfrosLeungTextureSynthesis7x7_D20.png, Runtime =26.006836
EfrosLeungTextureSynthesis11x11_D20.png, Runtime =20.214093
EfrosLeungTextureSynthesis15x15_D20.png, Runtime =10.001894
EfrosLeungTextureSynthesis5x5_Texture2.png, Runtime =42.926779
EfrosLeungTextureSynthesis7x7_Texture2.png, Runtime =31.535202
EfrosLeungTextureSynthesis11x11_Texture2.png, Runtime =26.273104
EfrosLeungTextureSynthesis15x15_Texture2.png, Runtime =13.513030
EfrosLeungTextureSynthesis5x5_english.png, Runtime =348.709546
EfrosLeungTextureSynthesis7x7_english.png, Runtime =282.986645
EfrosLeungTextureSynthesis11x11_english.png, Runtime =266.316631
EfrosLeungTextureSynthesis15x15_english.png, Runtime =151.162251

(base) C:\Users\14136\OneDrive\Desktop\670 Computer vision\hw2_release\code>
```

Figure 72: Efros Leung Synthesis Runtime Analysis with output size = 70 x 70 and window size = 5, 7, 11 and 15

Note: all output images are 70x70
Observation about correlation between window sizes on runtime and and synthesis quality:
1. 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.
2. 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.
3. It 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.
4. This makes a particular pixel with tiny patch get "wrong" value, which just snowballs upon further synthesis.
Hence, 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].

Implementation of *evalTextureSynth.py*

```
import numpy as np
from skimage import io
from skimage import color
import matplotlib.pyplot as plt
from scipy import ndimage
import random
import time
import warnings

import warnings
warnings.filterwarnings('ignore')
```

```python

def synthRandomPatch(image, tilesize, numtile, outsize):
    '''
    Texture Synthesis using Random tiling
    '''
    image2 = color.rgb2gray(image[:,:,:])
    synthimage = np.zeros((outsize,outsize))
    height, width = image2.shape

    for i in range(0, outsize, tilesize):
        for j in range(0,outsize,tilesize):
            tileX, tileY = tilesize, tilesize
            if(i + tilesize) >= outsize:
                tileX = outsize - i
            if(j + tilesize) >= outsize:
                tileY = outsize - j

            rowX = random.randint(0, height - tileX)
            colY = random.randint(0, width - tileY)

            image3 = image2[rowX:rowX + tileX, colY:colY + tileY]
            synthimage[i:i + tileX, j:j + tileY] = image3

    return synthimage

def synthEfrosLeung(image, winsize, outsize):
    '''
    Non-parametric Texture Synthesis using Efros & Leung algorithm
    '''
    image1 = color.rgb2gray(image[:,:,:])
    ErrThreshold = 0.1

    height, width = image1.shape
    synthimage = np.zeros((outsize, outsize))

    for i in range(height):
        for j in range(width):
            if image1[i][j] == 0:
                image1[i][j] = 1e-6

    centerX, centerY = int(outsize/2), int(outsize/2)
    rowX = random.randint(0, height - 3)
    columnY = random.randint(0, width -3)
    synthimage[centerX - 1:centerX + 2, centerY - 1:centerY + 2] = image1[rowX:
   rowX + 3, columnY:columnY + 3]

    pixelList = []
    structure = ndimage.generate_binary_structure(2, 2)
    NMask = ndimage.binary_dilation(synthimage, structure = structure)

    counter = 0
    for i in range(outsize):
        for j in range(outsize):
            if NMask[i][j]!=0 and synthimage[i][j]==0:
                pixelList.append((i,j))
                counter += 1

    prevList = None
```

```
    while pixelList:
        prevList = pixelList
        pixel = {}
        for p in pixelList:
            score = np.sum(synthimage[p[0]-1:p[0]+2, p[1]-1:p[1]+2] >0)
            pixel[(p[0], p[1])] = score

        sortedPixList = [k for k, v in sorted(pixel.items(), key= lambda x: x[1], reverse=True)]

        for p in sortedPixList:
            winXS, winXE = p[0] - int(winsize/2), p[0] + int(winsize/2) + 1
            winYS,winYE  = p[1] - int(winsize/2), p[1] + int(winsize/2) + 1
            if p[0] - int(winsize/2) < 0:
                winXS = 0
                winXE = winsize
            if p[0] + int(winsize/2) + 1 > outsize:
                winXS = outsize - winsize
                winXE = outsize
            if p[1] - int(winsize/2) < 0:
                winYS = 0
                winYE = winsize
            if p[1] + int(winsize/2) + 1 > outsize:
                winYS = outsize - winsize
                winYE = outsize

            outputPatch = synthimage[winXS:winXE, winYS:winYE]
            ValidMask = outputPatch > 0
            minSSD = float('inf')
            SSD = {}

            for row in range(0, height - winsize):
                for col in range(0,width - winsize):
                    patch = image1[row:row + winsize, col:col + winsize]
                    difference = (patch * ValidMask) - outputPatch
                    SSD[(row,col)] = np.sum(np.multiply(difference, difference))
                    minSSD = min(minSSD, np.sum(np.multiply(difference, difference)))

            BestMatches = []
            for pixel in list(SSD.keys()):
                if SSD[pixel] >= minSSD and SSD[pixel] <= (minSSD + ErrThreshold):
                    BestMatches.append(pixel)

            bmlegth = len(BestMatches)
            pickRPixel = BestMatches[random.randint(0, bmlegth -1)]
            pickPatch = image1[pickRPixel[0]:pickRPixel[0] + winsize, pickRPixel[1]:pickRPixel[1] + winsize]

            outputPatchHeight, outputPatchWidth = outputPatch.shape

            for row in range(outputPatchHeight):
                for col in range(outputPatchWidth):
                    if outputPatch[row,col]==0:
                        outputPatch[row,col] = pickPatch[row,col]

```

```
            synthimage[winXS:winXE, winYS:winYE] = outputPatch

        updateList = []
        structure1 = ndimage.generate_binary_structure(2, 2)
        NMask1 = ndimage.binary_dilation(synthimage, structure = structure1)

        counter1 = 0
        for i in range(outsize):
            for j in range(outsize):
                if NMask1[i][j]!=0 and synthimage[i][j]==0:
                    updateList.append((i,j))
                    counter1 += 1

        pixelList = updateList

    return synthimage

# ##########################################################################################
# TESTING                                                                                  #
# ##########################################################################################

#Load images
img1 = io.imread('../data/texture/D20.png')
img2 = io.imread('../data/texture/Texture2.bmp')
img3 = io.imread('../data/texture/english.jpg')

#UNCOMMENT TO TEST RANDOM PATCHES
"""
randomPaths = [img1, img2, img3]
name = ['D20.png','Texture2.png','english.png']
counter = 0
for img in randomPaths:
    iname = name[counter]
    counter = counter + 1
    outsize = 480
    #Different Block sizes
    for tilesize in [5, 15, 20, 30, 40]:
        numtile = int(outsize/tilesize)
        #Record Runtimes
        start = time.time()
        randomPatch = synthRandomPatch(img, tilesize, numtile, outsize)
        end = time.time()
        outfile_path = "RandomTextureSynthesis{}x{}_{}".format(tilesize, tilesize,iname)
        print("{}, Runtime ={:.6f}".format(outfile_path, (end - start)))
        #Displaying Random Patch output
        plt.subplot(3, 4, 1)
        plt.imshow(randomPatch, cmap = 'gray')
        plt.title("Synthesized Random Image = {}".format(iname))
        #Save Random Patch output
        plt.imsave(outfile_path, randomPatch, cmap = "gray")

"""
```

```
#Testing Non-parametric Texture Synthesis using Efros & Leung algorithm
EfrosLeungPaths = [img1, img2, img3]
name = ['D20.png','Texture2.png','english.png']
counter1 = 0
for img in EfrosLeungPaths:
    iname = name[counter1]
    counter1 = counter1 + 1
    outsize = 70
    #Different Window sizes
    for winsize in [5, 7, 11, 15]:
        #Record Runtimes
        start = time.time()
        Patch = synthEfrosLeung(img, winsize, outsize)
        end = time.time()
        outfile_path = "EfrosLeungTextureSynthesis{}x{}_{}".format(winsize,
    winsize, iname)
        print("{}, Runtime ={:.6f}".format(outfile_path, (end - start)))
        #Displaying Patch output
        plt.subplot(3, 4, 1)
        plt.imshow(Patch, cmap = 'gray')
        plt.title("Synthesized Efros & Leung Image = {}".format(iname))
        #Save Patch output
        plt.imsave(outfile_path, Patch, cmap = "gray")
```