
Emma Civello's Computer Vision Portfolio

Smith College CSC370 ~ Professor N. Howe

Exercise 7: Two Methods of Texture Synthesis

(Non-parametric Sampling & Image Quilting)

Description and Image Results

I chose to explore texture synthesis in this exercise because I am interested in video game development, and texture synthesis is frequently used to generate backgrounds in games. I looked at two methods: non-parametric sampling and image quilting.

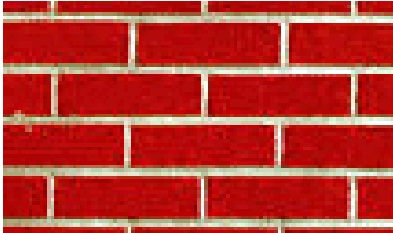
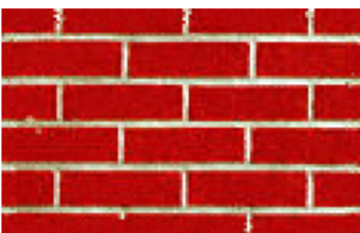

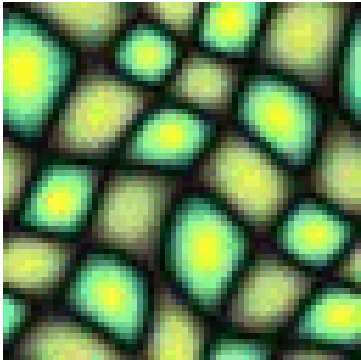
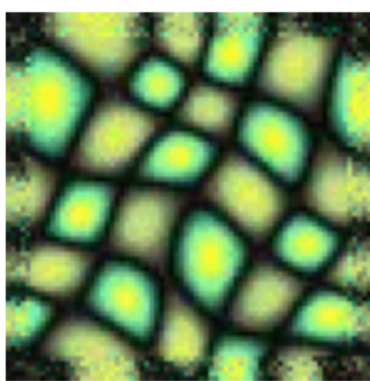
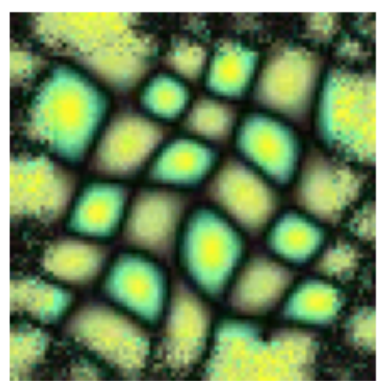
For non-parametric sampling, I was inspired by research (<https://people.eecs.berkeley.edu/~efros/research/EfrosLeung.html>) and pseudocode (<https://people.eecs.berkeley.edu/~efros/research/NPS/alg.html>) from Berkeley. I followed the pseudocode as best as I could, but some parts were left vague (for example, I did not quite understand how they used Gaussian filters), and so I made assumptions and omissions. I believe this is why my results were not as good as the ones the researchers present in the first link above. Here is what my version of their algorithm does:

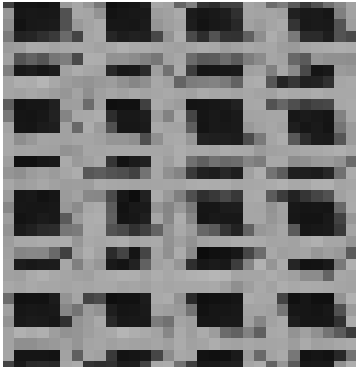
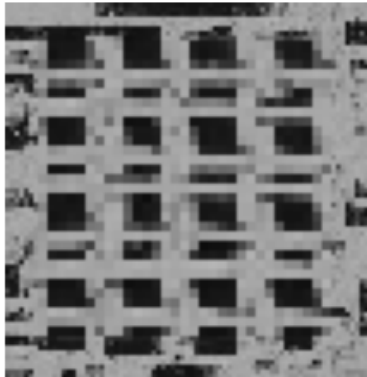
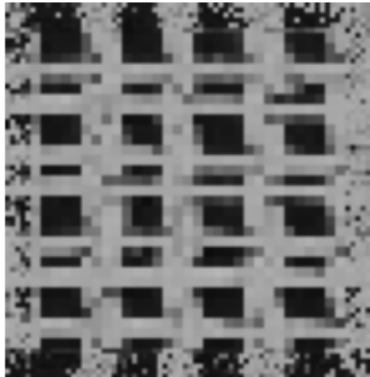

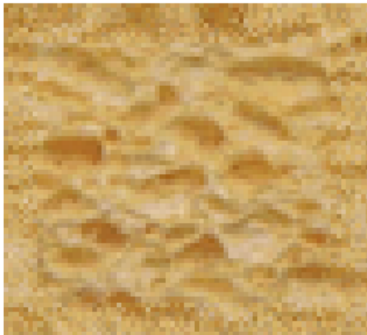

- Start with the original texture image and choose how many columns and rows you want to add as a border around it; create a new image with placeholder values for the pixels in these new rows and columns and the original image in the center.
- Make a list of all the unfilled / place-holder pixels that have one or more filled neighbors (in one of the cardinal directions)
- For each pixel with filled neighbors:

- Find a given number of random patches in the original sample image
- Find the one that gives a minimum when SSD is calculated between it and the neighborhood around the current pixel
- Give the current pixel the found neighborhood's center value

(and repeat until there are no unfilled pixels)

There were some problems with the output: it was noisy, it did not preserve structures (like the holes in the yellow sponge), and once it got off-track, it was prone to getting further off-track – there was no method of course-correction (this is especially evident in the green-and-black pattern). Some results, not to scale, are in the table below.

Initial Image (all are from: https://people.eecs.berkeley.edu/~efros/research/EfrosLeung.html)	Example 1	Example 2
		
	Neighborhood radius: 3 Rows / columns added: 10	Neighborhood radius: 3 Rows / columns added: 20
		
	Neighborhood radius: 3	Neighborhood radius: 3

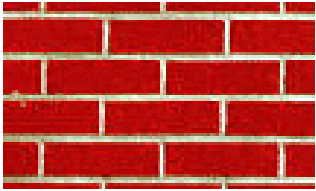

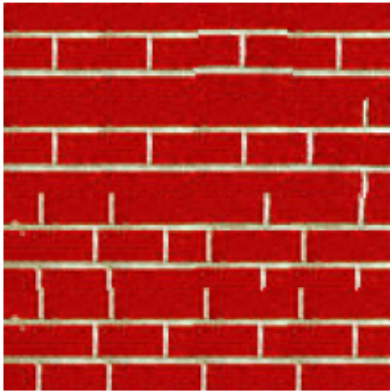
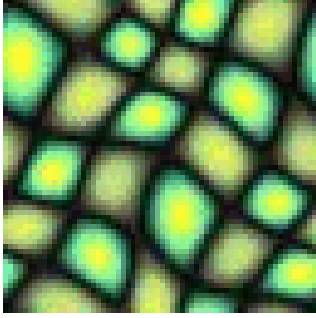
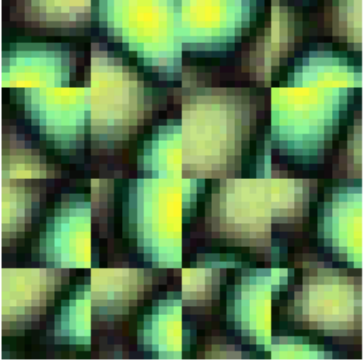
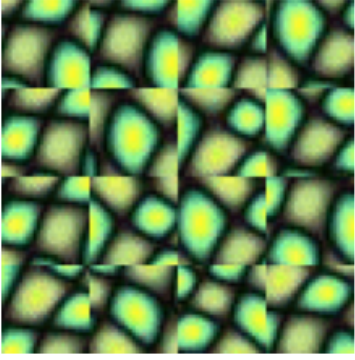
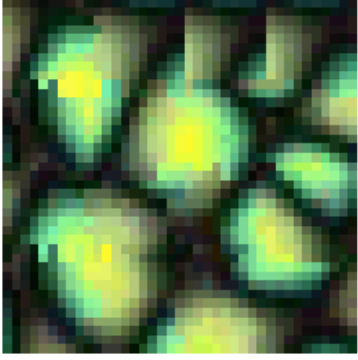
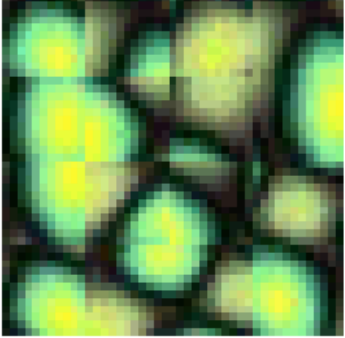
	Rows / columns added: 10	Rows / columns added: 20
	 Neighborhood radius: 2 Rows / columns added: 15	 Neighborhood radius: 10 Rows / columns added: 10
	 Neighborhood radius: 3 Rows / columns added: 15	 Neighborhood radius: 10 Rows / columns added: 10

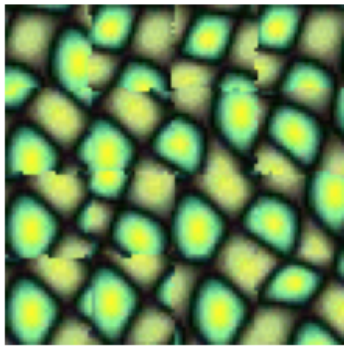

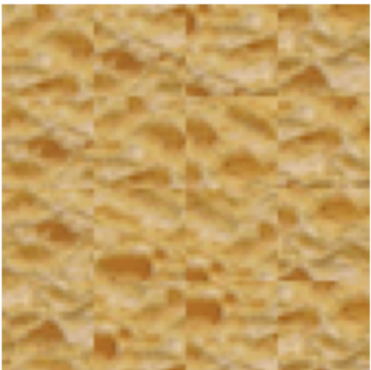




Because I did not like the noisy output of this method, I also explored image quilting. I was inspired by another paper from Berkeley (<https://people.eecs.berkeley.edu/~efros/research/quilting/quilting.pdf>) and a YouTube explanation (<https://youtu.be/m9WAIJfmyLM>) – and I wrote all of the code myself. Here is an overview of how it works:






- User chooses patch size, overlap between patches (it will be the same horizontally and vertically), number of patches to generate from the starting texture image, and number of patches to go into the final image.

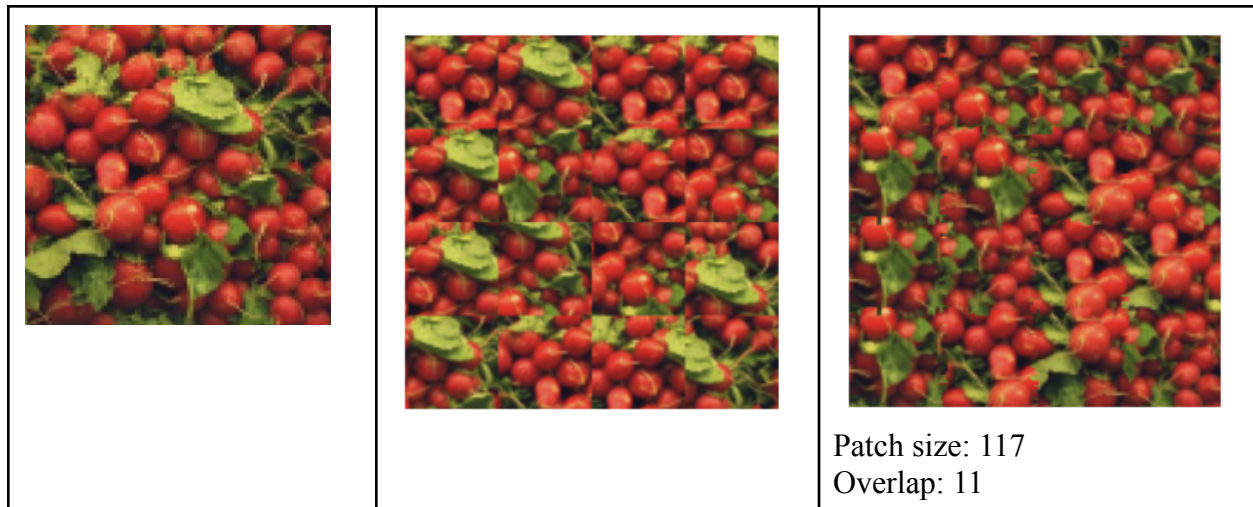
- The function `tilePatches` produces the first output – it randomly chooses patches from the list of generated ones to paste together into an image. Output from this function is in the middle column of the table below – the function ultimately creates undesirable output, and the YouTube video suggested writing it as a point of comparison for the next function.
- The function `tilePatchesV2` starts with a “working image” of the desired size, where pixels are placeholder values, and it does two main things:
 - Finds the best patch to go next: iterates over the list of generated patches; calculates the overlap SSD – the differences between the top and left borders of the patch and the corresponding parts on the working image; chooses an appropriate patch randomly from a few with the smallest SSDs.
 - Splices the patch into the working image: for the top of the patch and the region it overlaps in the working image, the code finds the squared difference between each pixel and stores it in a 2d array. For each column in this array, it finds the row of minimum difference; in the working image, all rows before this retain their original values, and all below take values from the new patch. (A similar process occurs for the left side of the current patch – but with rows and columns reversed.) Here is an image I generated during testing that shows a cut based on the left part of patches:



Initial Image (all are from: https://people.eecs.berkeley.edu/~efros/research/EfrosLeung.html and https://people.eecs.berkeley.edu/~efros/research/quilting/quilting.pdf)	Random patching; borders visible (used for comparison with right column)	The “final product” – SSD-chosen patching & patches cut along line of smallest difference
		 Patch size: 38 Overlap: 3
	 	 Patch size: 12 Overlap: 3 

		<p>Patch size: 12 Overlap: 1</p>  <p>Patch size: 25 Overlap: 2</p>
		 <p>Patch size: 21 Overlap: 2</p>
		 <p>Patch size: 51 Overlap: 10</p>

		 <p>Patch size: 68 Overlap: 13 (By random chance, this one is repetitive on the top and left)</p>
		 <p>Patch size: 68 Overlap: 13</p>  <p>Patch size: 51 Overlap: 5</p>



I think the results for the image quilting were pretty good; the sponge, rocks, cherries, and peppers could pass as non-synthetic images if one were looking quickly. I could also see this method being enhanced by other algorithms, such as ones that connect lines (which would help the pattern errors in the green and black images) or blend in parts that stick out from their surroundings. However, I think it would be difficult to fix the bricks / to achieve high performance on highly regular patterns. This version of image quilting performs best on images that are mostly monochrome and slightly random. Finally, image quilting introduced me to many things that I had not seen in previous exercises. I learned how to tile images together using `numpy.copyto()`, and I learned a method for finding a good line to cut along when merging two images together.

Code for Non-parametric Sampling

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Nov 12 18:06:01 2022

@author: emmacivello
```



```

"""
# yellow sponge image from:
https://people.eecs.berkeley.edu/~efros/research/EfrosLeung.html
# https://people.eecs.berkeley.edu/~efros/research/NPS/alg.html
# pseudocode for Efros and Leung

import numpy
from matplotlib import image as matImg
from matplotlib import pyplot as plt
import math
from PIL import Image
import torchvision.transforms as transforms
import random
import cv2

# =====
# General functions
# =====

def getImgData(name):
    image = matImg.imread(name, format="PNG")
    data = numpy.asarray(image)

    #transform to range 0-1
    if(numpy.issubdtype(numpy.uint8, data[0][0][0])):
        data = data.astype(float) / 255

    return data

def showImage(data):
    plt.imshow(data)
    plt.axis('off')
    plt.show()

def arrEquality(arr1, arr2):
    for i in range(arr1.size):
        if arr1[i] != arr2[i]:
            return False
    return True

```

```

def countSubArr(arr, subArr):
    rows, cols = arr.shape[0:2]
    instances = 0
    for i in range(len(arr)):
        if arrEquality(arr[i], subArr):
            instances += 1
    return instances

# =====
# SSD functions
# =====

def findSSD(possibleMatch, neighborhood):
    rows, cols = possibleMatch.shape[0:2]
    ssd = 0
    for i in range(rows):
        for j in range(cols):
            if not arrEquality(neighborhood[i][j],
numpy.asarray([-1,-1,-1,-1])):
                diff = possibleMatch[i][j] - neighborhood[i][j]
                ssd += diff * diff
    return ssd

def findBestMatch(neighborhood, possibleMatches):
    SSDs = []
    for i in range(len(possibleMatches)):
        newSSD = numpy.sum(findSSD(possibleMatches[i], neighborhood))
        # print(newSSD)
        SSDs.append(newSSD)
    return numpy.argmin(SSDs)

# =====
# Patch and seed functions
# =====

def getPatch(img, middleRow, middleCol, radiusR, radiusC): #called def
getNeighborhoodWindow(): in pseudocode -> function reused from my stereo
exercise

```

```

        return numpy.asarray(img[numpy.amin([middleRow-radiusR,
0]):middleRow+radiusR+1,
numpy.amin([middleCol-radiusC,0]):middleCol+radiusC+1])

```

```

def getCardinalNeighbors(img, row, col):
    rows, cols = img.shape[0:2]
    neighbors = []
    if(row-1 > -1):
        neighbors.append(img[row-1][col])
    if(row+1 < rows):
        neighbors.append(img[row+1][col])
    if(col-1 > -1):
        neighbors.append(img[row][col-1])
    if(col+1 < cols):
        neighbors.append(img[row][col+1])
    return numpy.asarray(neighbors)

```

```

def getNonSqSeed(img, sRows, sCols):
    iRows, iCols = img.shape[0:2]
    cornerRow = random.randint(0, iRows-sRows)
    cornerCol = random.randint(0, iCols-sCols)
    return numpy.asarray(img[cornerRow:cornerRow+sRows,
cornerCol:cornerCol+sCols])

```

```

def findMatches(neighborhood, img):
    possibleMatches = []
    rows, cols = neighborhood.shape[0:2]
    numPatchesToTry = 100
    for i in range(numPatchesToTry):
        newSeed = getNonSqSeed(img, rows, cols)
        possibleMatches.append(newSeed)
    return possibleMatches

```

```

# =====
# Image growing functions
# =====
def makeStartingImg(rows, cols, seed):
    img = []

```

```

for i in range(rows):
    img.append([])
    for j in range(cols):
        img[i].append([-1,-1,-1,-1])

sRows, sCols = seed.shape[0:2]
imgRow = int(rows/2) - int(sRows/2)
imgCol = int(cols/2) - int(sCols/2)
for i in range(imgRow, imgRow+sRows):
    for j in range(imgCol, imgCol+sCols):
        img[i][j] = seed[i-imgRow][j-imgCol]
return numpy.asarray(img)

def getUnfilledWithFilledNeighbors(img, windowSize): #assume that an unfilled
value is -1
    unfilled = []
    unfilledWFilledNeighbors = []
    rows, cols = img.shape[0:2]
    for i in range(rows):
        for j in range(cols):
            if(arrEquality(img[i][j], numpy.asarray([-1,-1,-1,-1]))):
                unfilled.append([i,j])
    print("UNFILLED LEN",len(unfilled))
    for i in range(len(unfilled)):
        centerR = unfilled[i][0]
        centerC = unfilled[i][1]
        neighborhood = getCardinalNeighbors(img, centerR, centerC)
        numUnfilled = countSubArr(neighborhood, numpy.asarray([-1,-1,-1,-1]))
        numFilled = len(neighborhood) - numUnfilled
        if numFilled > 0:
            unfilledWFilledNeighbors.append(unfilled[i])
    print("W NEIGHBORS",len(unfilledWFilledNeighbors))
    return unfilledWFilledNeighbors

def growImage(sampleImage, img, windowSize): #sampleImage = existing texture,
image = empty with 3x3 center seed, windowSize = size of neighborhood
    # function adapted from pseudocode from:
https://people.eecs.berkeley.edu/~efros/research/NPS/alg.html

```

```

newImg = img.copy()
filled = False
while not filled:
    pixelList = getUnfilledWithFilledNeighbors(newImg, windowSize)
#windowSize = squRadius
    if(len(pixelList) == 0):
        filled = True
    else:
        for i in range(len(pixelList)):
            middleRow = pixelList[i][0]
            middleCol = pixelList[i][1]
            neighborhood = getPatch(newImg, middleRow, middleCol,
windowSize, windowSize)
            possibleMatches = findMatches(neighborhood, sampleImage)
            bestMatch = findBestMatch(neighborhood, possibleMatches)
            chosen = possibleMatches[bestMatch]
            middleRow2, middleCol2 = chosen.shape[0:2]
            middleRow2 = int(middleRow2/2)
            middleCol2 = int(middleCol2/2)
            newImg[middleRow][middleCol] = chosen[middleRow2][middleCol2]
#apply blur here?
            showImage(newImg)
        return newImg

# =====
# Main
# =====

def main():
    img = getImgData("textureStart8.png")
    rows, cols = img.shape[0:2]
    showImage(img)
    startImg = makeStartingImg(rows+10, cols+10, img)
    showImage(startImg)
    endImg = growImage(img, startImg, 3)
    showImage(endImg)

main()

```


Code for Image Quilting

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 18 18:06:55 2022
https://people.eecs.berkeley.edu/~efros/research/quilting/quilting.pdf
explanation of the above^ algorithm: https://youtu.be/m9WaiJfmyLM
@author: emmacivello
"""

import numpy
from matplotlib import image as matImg
from matplotlib import pyplot as plt
import random

# =====
# General functions
# =====

def getImgData(name):
    image = matImg.imread(name, format="PNG")
    data = numpy.asarray(image)
    #transform to range 0-1
    if(numpy.issubdtype(numpy.uint8, data[0][0][0])):
        data = data.astype(float) / 255
    return data

def showImage(data):
    plt.imshow(data)
    plt.axis('off')
    plt.show()

# =====
# Sum of squared differences
# =====

def findSSD(patch1, patch2): # function adapted from (I'd never calculated SSD
before -> didn't know if it was pixel by pixel or sum of first patch minus sum
```

of second -> first one is the right method):

<https://stackoverflow.com/questions/26011224/how-does-sum-of-squared-difference-algorithm-work>

```

    rows, cols = patch1.shape[0:2]
    ssd = 0
    for i in range(rows):
        for j in range(cols):
            diff = patch1[i][j] - patch2[i][j]
            diff = numpy.sum(diff)
            ssd += diff * diff
    return ssd

```

```

def findSSDsOfTopOverlap(possibleMatches, overlap, givenTopPatch):
    rows, cols = possibleMatches[0].shape[0:2]
    SSDs = []
    for i in range(len(possibleMatches)):
        topOfCrtPatch = possibleMatches[i][0:overlap, 0:cols]
        SSDs.append(findSSD(topOfCrtPatch, givenTopPatch))
    return SSDs

```

```

def findSSDsOfLeftOverlap(possibleMatches, overlap, givenLeftPatch):
    rows, cols = possibleMatches[0].shape[0:2]
    SSDs = []
    for i in range(len(possibleMatches)):
        leftOfCrtPatch = possibleMatches[i][0:rows, 0:overlap]
        SSDs.append(findSSD(leftOfCrtPatch, givenLeftPatch))
    return SSDs

```

```

# =====
# Find patches
# =====

```

```

def getPatch(img, topLeftRow, topLeftCol, rows, cols): #called def
getNeighborhoodWindow(): in pseudocode -> function reused from my stereo
exercise
    return numpy.asarray(img[topLeftRow:topLeftRow+rows,
topLeftCol:topLeftCol+cols])

```

```

def findRandomPatches(img, n, patchSize):
    rows, cols = img.shape[0:2]
    randomPatches = []
    for i in range(n):
        crtRow = random.randint(0,rows-patchSize-1) # -1 ??
        crtCol = random.randint(0,cols-patchSize-1)
        randomPatches.append(getPatch(img, crtRow, crtCol, patchSize,
patchSize))
    return randomPatches

def findBestPatch(possiblePatches, upperLeftRow, upperLeftCol, overlap,
workingImg):
    rows, cols = possiblePatches[0].shape[0:2]
    topPatch = workingImg[upperLeftRow:upperLeftRow+overlap,
upperLeftCol:upperLeftCol+cols]
    leftPatch = workingImg[upperLeftRow:upperLeftRow+rows,
upperLeftCol:upperLeftCol+overlap]

    topSSDs = findSSDsOfTopOverlap(possiblePatches, overlap, topPatch)
    leftSSDs = findSSDsOfLeftOverlap(possiblePatches, overlap, leftPatch)
    sumSSDs = numpy.add(topSSDs, leftSSDs)
    sumSSDsWIndices = []

    for i in range(len(sumSSDs)):
        sumSSDsWIndices.append([sumSSDs[i],i])
    sumSSDsWIndices = numpy.asarray(sumSSDsWIndices)
    sumSSDsWIndices = sumSSDsWIndices[sumSSDsWIndices[:,0].argsort()]
    smallestSumSSDsWIndices = sumSSDsWIndices[0:2,1]
    indexOfGoodPatch = int(numpy.random.choice(smallestSumSSDsWIndices))

    return indexOfGoodPatch

# =====
# Create image by tiling patches (v1 - random, v2 - SSD and feathered together)
# =====

def tilePatches(patches, horizontalTiles, verticalTiles):

```

```

#on concatenating images:
https://note.nkmk.me/en/python-pillow-concat-images/
    rows, cols = patches[0].shape[0:2]
    horizPixels = cols*horizontalTiles
    vertPixels = rows*verticalTiles

    newImg = []
    for i in range(vertPixels):
        newImg.append([])
        for j in range(horizPixels):
            newImg[i].append([0.0,0.0,0.0,0.0])

    newImg = numpy.asarray(newImg)
    patchesIndex = 0
    for i in range(verticalTiles):
        for j in range(horizontalTiles):
            startRow = i * rows
            endRow = (i+1) * rows
            startCol = j * cols
            endCol = (j+1) * cols
            crtPatch = patches[patchesIndex]
            patchesIndex += 1
            numpy.copyto(newImg[startRow:endRow, startCol:endCol], crtPatch,
casting='safe', where=True)
            showImage(newImg)
            print("v1 shape",newImg.shape)
    return newImg

def tilePatchesV2(patches, horizontalTiles, verticalTiles, overlap):
    #on concatenating images:
https://note.nkmk.me/en/python-pillow-concat-images/
    rows, cols = patches[0].shape[0:2]
    horizPixels = cols*horizontalTiles - overlap*(horizontalTiles-1)
    vertPixels = rows*verticalTiles - overlap*(verticalTiles-1)

    newImg = []
    for i in range(vertPixels):
        newImg.append([])

```

```

        for j in range(horizPixels):
            # newImg[i].append([-1.0,-1.0,-1.0,-1.0])
            newImg[i].append([0.0,0.0,0.0,0.0])

newImg = numpy.asarray(newImg)
for i in range(verticalTiles):
    for j in range(horizontalTiles):
        startRow = i * rows - i*overlap
        endRow = (i+1) * rows - i*overlap
        startCol = j * cols - j*overlap
        endCol = (j+1) * cols - j*overlap
        patchIndex = findBestPatch(patches, startRow, startCol, overlap,
newImg)

        crtPatch = patches[patchIndex]
        splicedPatch = mergePatchWithImage(newImg, crtPatch, startRow,
startCol, overlap)

        numpy.copyto(newImg[startRow:endRow, startCol:endCol],
splicedPatch, casting='safe', where=True)

        showImage(newImg)
        return newImg

# =====
# Merge patches together
# =====
def mergePatchWithImage(newImg, crtPatch, startRow, startCol, overlap):
    updatedPatch = crtPatch.copy()
    rows, cols = updatedPatch.shape[0:2]

    if startCol > 0:
        updatedPatch = mergeLeft(newImg, updatedPatch, startRow, startCol,
overlap)
    if startRow > 0:
        updatedPatch = mergeTop(newImg, updatedPatch, startRow, startCol,
overlap)
    # print(rows," / ",cols)
    return updatedPatch

def mergeTop(newImg, patch, startRow, startCol, overlap):

```



```

mergedPatch = patch.copy()

rows, cols = patch.shape[0:2]
topOfImg = newImg[startRow:startRow+overlap, startCol:startCol+cols]
topOfPatch = patch[0:overlap, 0:cols]

difference = numpy.subtract(topOfImg, topOfPatch)
squared = numpy.square(difference)
overlapRows, overlapCols = squared.shape[0:2]

summed = []
for i in range(overlapRows):
    summed.append([])
    for j in range(overlapCols):
        summed[i].append(numpy.sum(squared[i][j]))
summed = numpy.asarray(summed)

for j in range(overlapCols):
    rowSplit = numpy.argmin(summed[:, j])
    for i in range(rowSplit):
        mergedPatch[i][j] = newImg[startRow+i][startCol+j]

return mergedPatch

def mergeLeft(newImg, patch, startRow, startCol, overlap):
    mergedPatch = patch.copy()

    rows, cols = patch.shape[0:2]
    leftOfImg = newImg[startRow:startRow+rows, startCol:startCol+overlap]
    leftOfPatch = patch[0:rows, 0:overlap]

    difference = numpy.subtract(leftOfImg, leftOfPatch)
    squared = numpy.square(difference)
    overlapRows, overlapCols = squared.shape[0:2]

    summed = []
    for i in range(overlapRows):

```

```

        summed.append([])
        for j in range(overlapCols):
            summed[i].append(numpy.sum(squared[i][j]))
        summed = numpy.asarray(summed)

    for i in range(overlapRows):
        colSplit = numpy.argmax(summed[i])
        for j in range(colSplit):
            mergedPatch[i][j] = newImg[startRow+i][startCol+j]
    return mergedPatch

# =====
# Main function
# =====

def main():
    img = getImgData("rocks.png")
    showImage(img)
    rows, cols = img.shape[0:2]

    # patchSize = int(numpy.amin(img.shape[0:2]))
    patchSize = int(numpy.amin([rows,cols]) * 0.3)
    overlap = int(patchSize*0.1) #pixels
    print("patch size ",patchSize)
    print("overlap ",overlap)

    horizontalTiles = 4 #patches
    verticalTiles = 4 #patches
    numRandomPatches = horizontalTiles * verticalTiles * 12
    print("numRandomPatches ",numRandomPatches)

    patches = findRandomPatches(img, numRandomPatches, patchSize)
    tilePatches(patches, horizontalTiles, verticalTiles)
    tilePatchesV2(patches, horizontalTiles, verticalTiles, overlap)

main()

```