Assignment #4

CPSC 425

Robert MacEachern

42415091

m9c7

**Question #5:**

Hand in a printed copy of the donkey image after texture synthesis has been used to remove the donkey.



**Question #6:**

Hand in texture synthesis results for 2 new images, where one shows the algorithm performing well and the other shows it performing poorly. Briefly describe why the method failed in the case in which it performed poorly.

**Answer #6:**

Performing well (tan.jpg)

 

Performing poorly (Brick-Wall.jpg)

 

Analysis of Brick-Wall.jpg

The result for this image seemed to break down at certain points, specifically with the vertical alignment of bricks in the centre and bottom parts of the fill area. I believe this may be a result of the algorithm having difficulty distinguishing between the dark edges between bricks and the other dark textures that also occur on individual bricks (such as the cracks, stains, and shadows). Once the vertical alignment of the bricks gets thrown off by a few bad patch selections, it becomes quite apparent to a human viewer, since fairly crisp horizontal lines are expected on a brick wall.

Also, the size of the fill area is quite large relative to the overall image which I believe impacts the perceived effectiveness of the result.

**Question #7:**

Provide an explanation for the effects of the randomPatchSD and patchL parameters. What results can be expected if these values are too small or too large, and why do these results happen?

**Answer #7:**

The randomPatchSD parameter impacts the randomness of the choice from the set of the best texture patches (based on our SSD calculation). The higher the value of randomPatchSD, the less likely we are to choose the absolute 'best' match (based on SSD). This randomness factor will help to add variation to our result when we encounter similar neighbourhoods while constructing our fill region. When randomPatchSD is too high, the algorithm will occasionally choose very poor matches. If randomPatchSD is set too low, the algorithm will always choose the best SSD match, resulting in a much greater chance that multiple instances of the same patch will appear in the result.

The patchL parameter is used to influence the size of the patch (patchSize = 2\*patchL+1) which determines the size of the region filled at each step. When the patchSize is too large, the resulting patches will tend to standout more noticeably and the edges between patches may become more noticeable to a human viewer. When patchSize is too small, some patterns (that don't fit in a patchSize window) may be missed and the resulting image will fail to capture the overall texture in our texture region.

Holefill.py

from PIL import Image, ImageDraw

import numpy as np

import random

import os.path

import pickle

##############################################################################

# Functions for you to complete #

##############################################################################

def ComputeSSD(TODOPatch, TODOMask, textureIm, patchL):

'''

ComputeSSD

Computes the sum of squares difference between the patch and texture image

for each valid location of the patch within the texture image.

TODOPatch - The image patch, of size [2 \* patchL + 1, 2 \* patchL + 1, 3]

TODOMask - Specifies the elements of which elements of TODOPatch are empty and

waiting to be filled in. A 1 represents an empty pixel.

The first two dimensions are the same as TODOPatch, and there is no

third dimension.

textureIm - The texture image. Size is [texImRows, texImCols, 3]

patchL - The length used to define the patch size.

'''

patch\_rows, patch\_cols, patch\_bands = np.shape(TODOPatch)

tex\_rows, tex\_cols, tex\_bands = np.shape(textureIm)

ssd\_rows = tex\_rows - 2 \* patchL

ssd\_cols = tex\_cols - 2 \* patchL

SSD = np.zeros((ssd\_rows,ssd\_cols))

for r in range(ssd\_rows):

for c in range(ssd\_cols):

# Compute sum square difference between textureIm and TODOPatch

# for all pixels where TODOMask = 0, and store the result in SSD

#

# ADD YOUR CODE HERE

#

for patchRow in range(patch\_rows):

for patchCol in range(patch\_cols):

if TODOMask[patchRow][patchCol] == 0:

patchArr = TODOPatch[patchRow][patchCol]

textureArr = textureIm[r + patchRow][c + patchCol]

SSD[r][c] += ((patchArr[0] - textureArr[0] \* 1.0) \*\* 2)

SSD[r][c] += ((patchArr[1] - textureArr[1] \* 1.0) \*\* 2)

SSD[r][c] += ((patchArr[2] - textureArr[2] \* 1.0) \*\* 2)

pass

pass

return SSD

def CopyPatch(imHole,TODOMask,textureIm,iPatchCenter,jPatchCenter,iMatchCenter,jMatchCenter,patchL):

'''

CopyPatch

Copies the patch into the destination image at the specified location.

imHole - The image we are copying the patch into.

TODOMask - Specifies the elements of which elements of TODOPatch are empty and

waiting to be filled in. A 1 represents an empty pixel.

The first two dimensions are the same as TODOPatch, and there is no

third dimension.

textureIm - The texture image. Size is [texImRows, texImCols, 3]

iPatchCenter - The row index of the patch center in the destination image

jPatchCenter - The column index of the patch center in the destination image

iMatchCenter - The row index of the match center in the texture image.

jMatchCenter - The column index of the patch center in the texture image.

patchL - The length used to define the patch size.

'''

patchSize = 2 \* patchL + 1

for i in range(patchSize):

for j in range(patchSize):

# Copy the selected patch selectPatch into the image containing

# the hole imHole for each pixel where TODOMask = 1.

# The patch is centred on iPatchCenter, jPatchCenter in the image imHole

#

# ADD YOUR CODE HERE

#

if TODOMask[i][j] == 1:

holeLoc = imHole[iPatchCenter - patchL + i][jPatchCenter - patchL + j];

textureLoc = textureIm[iMatchCenter - patchL + i][jMatchCenter - patchL + j];

holeLoc[0] = textureLoc[0];

holeLoc[1] = textureLoc[1];

holeLoc[2] = textureLoc[2];

pass

pass

return imHole

##############################################################################

# Some helper functions #

##############################################################################

def UserDefinedRegion(imRows, imCols):

print "A polygon consists of straight lines between the given coordinates,"

print "plus a straight line between the last and the first coordinate."

print "Enter row, column coordinates, comma separated, one point per line."

print "End with -1,-1"

fillPolyPoints = []

Done = False

while not Done:

user\_input = raw\_input()

coords = [int(x) for x in user\_input.split(",")]

assert (len(coords) == 2 and

-1 <= coords[0] < imRows and

-1 <= coords[1] < imCols), "Bad point coordinates"

if coords[0] == -1 and coords[1] == -1:

Done = True

if not Done:

fillPolyPoints.append(coords[1])

fillPolyPoints.append(coords[0])

assert len(fillPolyPoints) >= 6, "A polygon requires at least 3 points"

img = Image.new('L', (imCols, imRows), 0)

ImageDraw.Draw(img).polygon(fillPolyPoints, outline=1, fill=1)

fillRegion = np.array(img, dtype=np.uint8)

return fillRegion

def DrawBox(im,x1,y1,x2,y2):

draw = ImageDraw.Draw(im)

draw.line((x1,y1,x1,y2),fill="white",width=1)

draw.line((x1,y1,x2,y1),fill="white",width=1)

draw.line((x2,y2,x1,y2),fill="white",width=1)

draw.line((x2,y2,x2,y1),fill="white",width=1)

del draw

return im

def Find\_Edge(hole\_mask):

[cols, rows] = np.shape(hole\_mask)

edge\_mask = np.zeros(np.shape(hole\_mask))

for y in range(rows):

for x in range(cols):

if (hole\_mask[x,y] == 1):

if (hole\_mask[x-1,y] == 0 or

hole\_mask[x+1,y] == 0 or

hole\_mask[x,y-1] == 0 or

hole\_mask[x,y+1] == 0):

edge\_mask[x,y] = 1

return edge\_mask

##############################################################################

# Main script starts here #

##############################################################################

#

# Constants

#

# Change patchL to change the patch size used (patch size is 2 \*patchL + 1)

patchL = 10

patchSize = 2\*patchL+1

# Standard deviation for random patch selection

randomPatchSD = 1

# Display results interactively

showResults = True

#

# Read input image

#

im = Image.open('donkey.jpg').convert('RGB')

im\_array = np.asarray(im, dtype=np.uint8)

imRows, imCols, imBands = np.shape(im\_array)

#

# Define hole and texture regions. This will use regions.pkl if it exists,

# but otherwise will allow the user to select the regions.

if os.path.isfile('regions.pkl'):

regions\_file = open('regions.pkl', 'rb')

fillRegion = pickle.load(regions\_file)

textureRegion = pickle.load(regions\_file)

regions\_file.close()

else:

# let user define fill region

print "Specify the fill region by entering coordinates of the bounding polygon"

fillRegion = UserDefinedRegion(imRows,imCols)

# let user define texture region

print "Specify the region to use as the texture sample"

print "Note: This region will made rectangular"

textureRegion = UserDefinedRegion(imRows,imCols)

# Note: we should save these results to avoid user retyping

regions\_file = open('regions.pkl', 'wb')

pickle.dump(fillRegion, regions\_file, -1)

pickle.dump(textureRegion, regions\_file, -1)

regions\_file.close()

#

# Get coordinates for hole and texture regions

#

fill\_indices = fillRegion.nonzero()

nFill = len(fill\_indices[0]) # number of pixels to be filled

iFillMax = max(fill\_indices[0])

iFillMin = min(fill\_indices[0])

jFillMax = max(fill\_indices[1])

jFillMin = min(fill\_indices[1])

assert((iFillMin >= patchL) and

(iFillMax < imRows - patchL) and

(jFillMin >= patchL) and

(jFillMax < imCols - patchL)) , "Hole is too close to edge of image for this patch size"

texture\_indices = textureRegion.nonzero()

iTextureMax = max(texture\_indices[0])

iTextureMin = min(texture\_indices[0])

jTextureMax = max(texture\_indices[1])

jTextureMin = min(texture\_indices[1])

textureIm = im\_array[iTextureMin:iTextureMax+1, jTextureMin:jTextureMax+1, :]

texImRows, texImCols, texImBands = np.shape(textureIm)

assert((texImRows > patchSize) and

(texImCols > patchSize)) , "Texture image is smaller than patch size"

#

# Initialize imHole for texture synthesis (i.e., set fill pixels to 0)

#

imHole = im\_array.copy()

imHole[fill\_indices] = 0

#

# Is the user happy with fillRegion and textureIm?

#

if showResults == True:

# original

im.show()

# convert to a PIL image, show fillRegion and draw a box around textureIm

im1 = Image.fromarray(imHole).convert('RGB')

im1 = DrawBox(im1,jTextureMin,iTextureMin,jTextureMax,iTextureMax)

im1.show()

print "Are you happy with this choice of fillRegion and textureIm?"

Yes\_or\_No = False

while not Yes\_or\_No:

answer = raw\_input("Yes or No: ")

if answer == "Yes" or answer == "No":

Yes\_or\_No = True

assert answer == "Yes", "You must be happy. Please try again."

#

# Perform the hole filling

#

while (nFill > 0):

print "Number of pixels remaining = " , nFill

# Set TODORegion to pixels on the boundary of the current fillRegion

TODORegion = Find\_Edge(fillRegion)

edge\_pixels = TODORegion.nonzero()

nTODO = len(edge\_pixels[0])

while(nTODO > 0):

# Pick a random pixel from the TODORegion

index = np.random.randint(0,nTODO)

iPatchCenter = edge\_pixels[0][index]

jPatchCenter = edge\_pixels[1][index]

# Define the coordinates for the TODOPatch

TODOPatch = imHole[iPatchCenter-patchL:iPatchCenter+patchL+1,jPatchCenter-patchL:jPatchCenter+patchL+1,:]

TODOMask = fillRegion[iPatchCenter-patchL:iPatchCenter+patchL+1,jPatchCenter-patchL:jPatchCenter+patchL+1]

#

# Compute masked SSD of TODOPatch and textureIm

#

ssdIm = ComputeSSD(TODOPatch, TODOMask, textureIm, patchL)

# Randomized selection of one of the best texture patches

ssdIm1 = np.sort(np.copy(ssdIm),axis=None)

ssdValue = ssdIm1[min(round(abs(random.gauss(0,randomPatchSD))),np.size(ssdIm1)-1)]

ssdIndex = np.nonzero(ssdIm==ssdValue)

iSelectCenter = ssdIndex[0][0]

jSelectCenter = ssdIndex[1][0]

# adjust i, j coordinates relative to textureIm

iSelectCenter = iSelectCenter + patchL

jSelectCenter = jSelectCenter + patchL

selectPatch = textureIm[iSelectCenter-patchL:iSelectCenter+patchL+1,jSelectCenter-patchL:jSelectCenter+patchL+1,:]

#

# Copy patch into hole

#

imHole = CopyPatch(imHole,TODOMask,textureIm,iPatchCenter,jPatchCenter,iSelectCenter,jSelectCenter,patchL)

# Update TODORegion and fillRegion by removing locations that overlapped the patch

TODORegion[iPatchCenter-patchL:iPatchCenter+patchL+1,jPatchCenter-patchL:jPatchCenter+patchL+1] = 0

fillRegion[iPatchCenter-patchL:iPatchCenter+patchL+1,jPatchCenter-patchL:jPatchCenter+patchL+1] = 0

edge\_pixels = TODORegion.nonzero()

nTODO = len(edge\_pixels[0])

fill\_indices = fillRegion.nonzero()

nFill = len(fill\_indices[0])

#

# Output results

#

if showResults == True:

Image.fromarray(imHole).convert('RGB').show()

Image.fromarray(imHole).convert('RGB').save('results.jpg')