Multimedia Image Processing

Assignment 5

Gwangmin Kim(김광민)
2018112540

Department of Multimedia Engineering
Dongguk University

```
import cv2
import numpy as np

def getImage(grayscale = False, scale = 0.5):

  if grayscale:
    grayscale = 0
  else:
    grayscale = 1
  original = cv2.imread('dgu_gray.png', grayscale)
  gt = cv2.resize(original, (0,0), fx = scale, fy = scale)
  return [original, gt]


def addNoise(image, mean = 0,  sigma = 0.3):

  sigma *= 255 #Since the image itself is not normalized
  noise = np.zeros_like(image)
  noise = cv2.randn(noise, mean, sigma)
  ret = cv2.add(image, noise) #generate and add gaussian noise
  return ret


# 논 로컬 민
def nonLocalMeans(noisy, params = tuple(), verbose = True):
  '''
  Performs the non-local-means algorithm given a noisy image.
  params is a tuple with:
  params = (bigWindowSize, smallWindowSize, h)
  Please keep bigWindowSize and smallWindowSize as even numbers
  '''

  bigWindowSize, smallWindowSize, h  = params
  padwidth = bigWindowSize//2
  image = noisy.copy()

  # The next few lines creates a padded image that reflects the border so that the
big window can be accomodated through the loop
```

```
  paddedImage = np.zeros((image.shape[0] + bigWindowSize,image.shape[1] +
bigWindowSize))
  paddedImage = paddedImage.astype(np.uint8)
  paddedImage[padwidth:padwidth+image.shape[0], padwidth:padwidth+image.shape[1]] =
image
  paddedImage[padwidth:padwidth+image.shape[0], 0:padwidth] =
np.fliplr(image[:,0:padwidth])
  paddedImage[padwidth:padwidth+image.shape[0],
image.shape[1]+padwidth:image.shape[1]+2*padwidth] =
np.fliplr(image[:,image.shape[1]-padwidth:image.shape[1]])
  paddedImage[0:padwidth,:] = np.flipud(paddedImage[padwidth:2*padwidth,:])
  paddedImage[padwidth+image.shape[0]:2*padwidth+image.shape[0], :]
=np.flipud(paddedImage[paddedImage.shape[0] - 2*padwidth:paddedImage.shape[0] -
padwidth,:])

  iterator = 0
  totalIterations = image.shape[1]*image.shape[0]*(bigWindowSize -
smallWindowSize)**2

  if verbose:
    print("TOTAL ITERATIONS = ", totalIterations)

  outputImage = paddedImage.copy()
  smallhalfwidth = smallWindowSize//2

  # For each pixel in the actual image, find a area around the pixel that needs to
be compared
  for imageX in range(padwidth, padwidth + image.shape[1]):
    print("imageX: ", imageX)
    for imageY in range(padwidth, padwidth + image.shape[0]):

      bWinX = imageX - padwidth
      bWinY = imageY - padwidth

      #comparison neighbourhood
      compNbhd = paddedImage[imageY - smallhalfwidth:imageY + smallhalfwidth +
1,imageX-smallhalfwidth:imageX+smallhalfwidth + 1]

      pixelColor = 0
      totalWeight = 0

      # For each comparison neighbourhood, search for all small windows within a
large box, and compute their weights
      for sWinX in range(bWinX, bWinX + bigWindowSize - smallWindowSize, 1):
        for sWinY in range(bWinY, bWinY + bigWindowSize - smallWindowSize, 1):
          #find the small box
          smallNbhd = paddedImage[sWinY:sWinY+smallWindowSize +
1,sWinX:sWinX+smallWindowSize + 1]
          euclideanDistance = np.sqrt(np.sum(np.square(smallNbhd - compNbhd)))
          #weight is computed as a weighted softmax over the euclidean distances
          weight = np.exp(-euclideanDistance/h)
```

```
          totalWeight += weight
          pixelColor += weight*paddedImage[sWinY + smallhalfwidth, sWinX +
smallhalfwidth]
          iterator += 1

          if verbose:
            percentComplete = iterator*100/totalIterations
            if percentComplete % 5 == 0:
              print('% COMPLETE = ', percentComplete)

      pixelColor /= totalWeight
      outputImage[imageY, imageX] = pixelColor

  return
outputImage[padwidth:padwidth+image.shape[0],padwidth:padwidth+image.shape[1]]


def denoise(verbose = False, gaussian = True, salted = True):  # 가우시안 노이즈 더한
이미지, 노이즈 제거한 이미지 반환

  scale = 2 #Scale factor of the image
  [original, gtImg] = getImage(grayscale = True, scale = scale)

  # Noise parameters
  sigma = 0.15 #Gaussian sigma

  gNoised = addNoise(gtImg, sigma = sigma)

  # Parameters for denoising using gaussian filter
  kernelSize = 3
  kernel = (kernelSize , kernelSize)

  #NLM filter parameters
  gParams = {
    'bigWindow' : 20,
    'smallWindow':6,
    'h':14,
    'scale':scale,
  }

  #perform NLM filtering
  nlmFilteredGNoised = nonLocalMeans(gNoised, params = (gParams['bigWindow'],
gParams['smallWindow'],gParams['h']), verbose = verbose)

  return [original, gNoised, nlmFilteredGNoised]

imgs = denoise()
original = imgs[0]
gaussian = imgs[1]
noiseRemoval = imgs[2]
```

```
cv2.imshow('original image', original)
cv2.imshow('Gaussian noise Add', gaussian)
cv2.imshow('Gaussian noise Removal', noiseRemoval)

cv2.waitKey(0)
cv2.destroyAllWindows()
```