COMPUTER VISION

ISTANBUL TECHNICAL UNIVERSITY

HOMEWORK-2 REPORT

OCTOBER 29, 2017

Name: EMRE KÖSE
ID: 150130037

Computer Vision Homework-2
Report

Note: The execution of programs can take a few seconds(5s -10s)

1) Spatial Filtering

a- In this part mean filter was implemented . To implement mean filter, firstly the function was written which finds the neighbors of pixels with intended range. We used the 3*3 range for neighbors.

The code which finds the neighbors of the one pixel(n is range):

```
 def produce_Neighbours(img,i, j, n):
 p_v = []
 row_n = np.size(img, 0)
 column_n = np.size(img, 1)
 for r in range( i-int((n-1)/2),i+(int((n-1)/2))+1,1):
   for c in range(j - int((n - 1) / 2), j + int((n - 1) / 2)+1,1):
     if r >=0 and c >=0 and r<=row_n-1 and c <=column_n-1:
       p_v.append(img[r,c])
     else:
       p_v.append(0)
 return p_v
```

With using of this function, every pixel’s intensity value is changed with the mean value of the neighbors.

Mean filter implementation:

```
def ApplyMeanFilter(img,n):
  row_n = np.size(img, 0)
  column_n = np.size(img, 1)
  for i in range(row_n):
    for j in range(column_n):
      array = produce_Neighbours(img,i,j,n)
      meanValue = round(np.mean(array))
      img[i,j] = meanValue
  return img
```

b- In this part median filter was implemented . Neighbors were found again and the median of these neighbors’ values was assigned to current pixel.

```
def ApplyMedianFilter(img,n):
  row_n = np.size(img, 0)
  column_n = np.size(img, 1)
  for i in range(row_n):
    for j in range(column_n):
      array = produce_Neighbours(img,i,j,n)
      medianValue = round(np.median(array))
      img[i,j] = medianValue
  return img
```

c- In this part mean-median filter was implemented . Neighbors were found again. Then, the mean value of these neighbors was multiplied with alpha value which is taken by function as a parameter and the median value was multiplied with (1-a). The sum of these two value was assigned to current pixel.

```
def ApplyMeanAndMedianFilter(img,n,a):
  row_n = np.size(img, 0)
  column_n = np.size(img, 1)
  for i in range(row_n):
    for j in range(column_n):
      array = produce_Neighbours(img,i,j,n)
      meanValue = round(np.mean(array)*a)
      medianValue = round(np.median(array)*(1-a))
      img[i,j] = meanValue+medianValue

  return img
```

d- Gaussian noise images include noises with the form of more proportionally distributed values. So, in order to fix this kind of images, we can not take one value among them because there are various values and it is so risky situation to take one value with median. Hence, we can more trust to take mean value of the current pixel's neighbors' values.

In contrast, impulse noise images include top and bottom intensity values like 0 and 255, and the rate of these noisy bits is less than the rate of noisy bits in Gaussian noise images. If median filtering is used, the extreme values(noisy pixel values) will not be taken as a pixel to change. So, the image can be fixed with median filtering.

2) Kirsch Compass Operator

a)

In this part, the kirsch compass edge detection algorithm was implemented.

Firstly, eight direction matrices were defined:

```
kirsch_N= np.array([[-3, -3, -3], [-3, 0, -3], [5, 5, 5]])
kirsch_W = np.array([[-3, -3, 5], [-3, 0, 5], [-3, -3, 5]])
kirsch_S = np.array([[5, 5, 5], [-3, 0, -3], [-3, -3, -3]])
kirsch_E = np.array([[5, -3, -3], [5, 0, -3],[ 5, -3, -3]])
kirsch_NW = np.array([[-3, -3, -3], [-3, 0, 5],[ -3, 5, 5]])
kirsch_SW = np.array([[-3, 5, 5], [-3, 0, 5], [-3, -3, -3]])
kirsch_SE = np.array([[5, 5, -3], [5, 0, -3], [-3, -3, -3]])
kirsch_NE = np.array([[-3, -3, -3],[ 5, 0, -3],[ 5, 5, -3]])
```

Then, to produce neighbors of a pixel as a matrix, produce_Neighbours(img, i, j, n) function was defined which is given below:

```
def produce_Neighbours(img, i, j, n):
p_v = np.array([[0, 0, 0], [0, img[i,j], 0], [0, 0, 0]])
row_n = np.size(img, 0)
column_n = np.size(img, 1)
k=0
for r in range(i - int((n - 1) / 2), i + (int((n - 1) / 2)) + 1, 1):
  l=0
  for c in range(j - int((n - 1) / 2), j + int((n - 1) / 2) + 1, 1):
    if r >= 0 and c >= 0 and r <= row_n - 1 and c <= column_n - 1:
      p_v[k,l]=(img[r, c])
    l=l+1
  k=k+1
return p_v
```

After these steps convolution of every pixel with its neighbors was implemented:

```
def convolution
(array,kirsch_N,kirsch_W,kirsch_S,kirsch_E,kirsch_NW,kirsch_SW,kirsch_SE,kirsch_NE):
 conv_ValueN = (kirsch_N * array).sum()
 conv_ValueW = (kirsch_W * array).sum()
 conv_ValueS = (kirsch_S * array).sum()
 conv_ValueE = (kirsch_E * array).sum()
 conv_ValueNW = (kirsch_NW * array).sum()
 conv_ValueSW = (kirsch_SW * array).sum()
 conv_ValueSE = (kirsch_SE * array).sum()
 conv_ValueNE = (kirsch_NE * array).sum()

return conv_ValueN,conv_ValueW,conv_ValueS,conv_ValueE,

            conv_ValueNW,conv_ValueSW,conv_ValueSE,conv_ValueNE
```

After the convolution function was implemented, eight different matrix was produced by **applyKirsch**() function and all of these matrices were shown in Q2.py file.

b) In this part, **Jmag** image was created with finding the maximum response among each pixel's convolution values of directions.

The maximum response was found with the below code:
In this code, "max" value holds the maximum value of the response for each pixel and all max. responses produce the final image **Jmag**.

```
def gradient_direction(eight_direction):
 Jmag = np.zeros(eight_direction[0].shape, dtype=np.int)
 Jdir = np.zeros(eight_direction[0].shape, dtype=np.int)
 max = 0
 dir=0
 for row in range(Jmag.shape[0]):
   for col in range(Jmag.shape[1]):
     for i in range(8):
       if eight_direction[i][row][col] > max:
         max =eight_direction[i][row][col]
         dir = i
     Jmag[row][col] = max
     Jdir[row][col] = dir+1
     max = 0
 return Jmag,Jdir
```

Then threshold operation was applied like below with the appropriate value (0.3)

```
Jmag = gradient_direction(eight_direction)
Jmag = exposure.equalize_adapthist(Jmag/np.max(np.abs(Jmag)), clip_limit=0.3)
plt.title("JMAG")
plt.imshow(Jmag, cmap='gray')
plt.show()
```

c) In this part, **Jdir** image was created in the `gradient_direction()` function with same time **Jmag** image. According to maximum response the direction of edge was determined and the numbers were assigned from 1 to 8 with the appropriate directions.

Then according to these encodings, 2D vector field U(x,y) and V(x,y) was filled by related values with the implementation in below:(But I could not visualize the directions with)

```
def produceUandV(img):
):
  U = np.zeros(eight_direction[0].shape, dtype=np.float32)
  V = np.zeros(eight_direction[0].shape, dtype=np.float32)
  for row in range(img.shape[0]):
    for col in range(img.shape[1]):
      if img[row][col] == 1: #North
        U[row][col] = 0
        V[row][col] = 1
      elif img[row][col] == 2: #e
        U[row][col] = 1
        V[row][col] = 0
      elif img[row][col] == 3:#s
        U[row][col] = 0
        V[row][col] = -1
      elif img[row][col] == 4:#w
        U[row][col] = -1
        V[row][col] = 0
      elif img[row][col] == 5:#ne
        U[row][col] = np.sqrt(2)/2
        V[row][col] = np.sqrt(2)/2
      elif img[row][col] == 6:#se
        U[row][col] = np.sqrt(2)/2
        V[row][col] = -np.sqrt(2)/2
      elif img[row][col] == 7:#nw
        U[row][col] = -np.sqrt(2)/2
        V[row][col] = np.sqrt(2)/2
      elif img[row][col] == 8:#sw
        U[row][col] = -np.sqrt(2)/2
        V[row][col] = -np.sqrt(2)/2
  return U,V
```

d) The gradients were found but I couldn't understand what will I do after the Imag image was created:

```
gx, gy = np.gradient(I)
mag = cv2.addWeighted(gx, 0.5, gy, 0.5, 0)
plt.title("magnitude")
plt.imshow(mag, cmap='gray')
plt.show()
```

f) In this part, Gaussian filter was implemented. Firstly, Gaussian kernel was produced with the sigma value(0.6):

```
#Q2 PART F - GAUSSIAN FILTERING
def produceG(x,y,sigma):
  pi = np.pi
  return (1/(2*pi*(sigma * sigma))) * np.exp(-(x*x + y*y)/(2*sigma*sigma))
def gaussian_kernel(X, Y, sig):
  kernel = np.zeros((3, 3))
  for row in range(3):
    for column in range(3):
      kernel[row, column] = produceG(X[row][column],Y[row][column],sig)
  return kernel
```

Then the image was filtered with the convolution of the image:

```
def gaussianFilter(img):

  X = [[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]]
  Y = [[-1, -1, -1], [0, 0, 0], [1, 1, 1]]
  gauss_kernel = gaussian_kernel(X, Y, 0.6)
  GaussianFilteredImage=  np.zeros(img.shape, dtype=np.int)
  for r in range(img.shape[0]):
    for c in range(img.shape[1]):
      matrix = produce_Neighbours(img, r, c, 3)
      conv_Value = (matrix*gauss_kernel).sum()
      GaussianFilteredImage[r,c]= conv_Value
  return GaussianFilteredImage
```