Comp Photography (Spring 2016) HW 4

Stewart Boyd

imageGradientX

```python
def imageGradientX(image):
    """ This function differentiates an image in the X direction.

    Note: See lectures 02-06 (Differentiating an image in X and Y) :
    explanation of how to perform this operation.

    The X direction means that you are subtracting columns:
    der. F(x, y) = F(x+1, y) - F(x, y)
    This corresponds to image[r,c] = image[r,c+1] - image[r,c]

    You should compute the absolute value of the differences in ord
    setting a pixel to a negative value which would not make sense.

    We want you to iterate the image to complete this function. You
    any functions that automatically do this for you.

    Args:
        image (numpy.ndarray): A grayscale image represented in a nu

    Returns:
        output (numpy.ndarray): The image gradient in the X directi
                                of the output array should have a w
                                one less than the original since no
                                can be done once the last column is
    """
    # WRITE YOUR CODE HERE.

    (num_rows, num_cols) = image.shape
    new_image = np.zeros((num_rows, num_cols - 1))
    for j in xrange(num_cols - 1):
        for i in xrange(num_rows):
            delta = int(image[i, j + 1]) - int(image[i, j])
            new_image[i, j] = abs(delta)
    return new_image
```

For imageGradientX, I began by creating an empty array with 1 less column then the original image. Then I iterated through the indices of each column and row (sans the last column index) and subtracted the intensity values according to the equations given in the lectures as well as the python doc string shown below:

$$F(x, y) = F(x + 1, y) - F(x, y)$$

I then returned the resulting image

imageGradientY

```
def imageGradientY(image):
    """ This function differentiates an image in the Y direction.

    Note: See lectures 02-06 (Differentiating an image in X and Y
    explanation of how to perform this operation.

    The Y direction means that you are subtracting rows:
    der. F(x, y) = F(x, y+1) - F(x, y)
    This corresponds to image[r,c] = image[r+1,c] - image[r,c]

    You should compute the absolute value of the differences in o
    setting a pixel to a negative value which would not make sens

    We want you to iterate the image to complete this function. Y
    any functions that automatically do this for you.

    Args:
        image (numpy.ndarray): A grayscale image represented in a

    Returns:
        output (numpy.ndarray): The image gradient in the Y direc
                                of the output array should have a
                                one less than the original since
                                can be done once the last row is
    """
    # WRITE YOUR CODE HERE.

    (num_rows, num_cols) = image.shape
    new_image = np.zeros((num_rows - 1, num_cols))
    for j in xrange(num_cols):
        for i in xrange(num_rows - 1):
            delta = int(image[i + 1, j]) - int(image[i, j])
            new_image[i, j] = abs(delta)
    return new_image
```

For imageGradientY, I began by creating an empty array with 1 less row then the original image. Then I iterated through the indices of each column and row (sans the last row index) and subtracted the intensity values according to the equations given in the lectures as well as the python doc string shown below:

$$F(x, y) = F(x, y + 1) - F(x, y)$$

I then returned the resulting image

computeGradient

```python
106 def computeGradient(image, kernel):
107     """ This function applies an input 3x3 kernel to the image, and outputs the
108     result. This is the first step in edge detection which we discussed in
109     lecture.
110
111     You may assume the kernel is a 3 x 3 matrix.
112     View lectures 2-05, 2-06 and 2-07 to review this concept.
113
114     The process is this: At each pixel, perform cross-correlation using the
115     given kernel. Do this for every pixel, and return the output image.
116
117     The most common question we get for this assignment is what do you do at
118     image[i, j] when the kernel goes outside the bounds of the image. You are
119     allowed to start iterating the image at image[1, 1] (instead of 0, 0) and
120     end iterating at the width - 1, and column - 1.
121
122     Note: The output is a gradient depending on what kernel is used.
123
124     Args:
125         image (numpy.ndarray): A grayscale image represented in a numpy array.
126         kernel (numpy.ndarray): A 3x3 kernel represented in a numpy array.
127
128     Returns:
129         output (numpy.ndarray): The computed gradient for the input image. The
130                                 size of the output array should be two rows and
131                                 two columns smaller than the original image
132                                 size.
133     """
134     # WRITE YOUR CODE HERE.
135
136     (num_row, num_col) = image.shape
137     new_image = np.zeros((num_row - 2, num_col - 2))
138     for j in xrange(1, num_col - 1):
139         for i in xrange(1, num_row - 1):
140             new_image[i - 1, j - 1] = (kernel * image[i - 1:i + 2, j - 1: j + 2]).sum()
141     return new_image
142
```

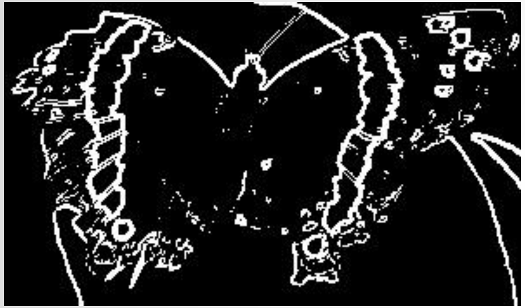
…	…	…	…	…	…
…	i-1,j -1	i-1,j	i-1,j +1	…	…
…	i,j -1	i,j	i,j +1	…	…
…	i+1,j -1	i+1,j	i+1,j +1	…	…
…	…	…	…	…	…

For computeGradient, I began by creating an empty array with 2 less columns and 2 less rows then the original image. Then I iterated through the indices of each column and row (sans the first and last column and row index) . In line 140 of code I do the following things:

1. Use array slicing to get a 3X3 subset image array of the ndarray passed as *image* argument surroudning the current index i, j (see image graphic above for illustration)
2. Use element by element multipication (* operator) to calculate 3 X 3 matrix of kernel * subimage
3. Use *sum* method to sum up individual elements of kernel * subimage and place at the index I – 1, j -1 (because looping started at 1 index instead of 0).

After looping finishes the *new_image* is returned

Edge Detection using a Sobel Kernel and a threshold of 150



```python
import assignment4
import cv2
import numpy as np
import os
def convertToBlackAndWhite(image, threshold = 128):
    for elem in np.nditer(image, op_flags = ['readwrite']):
        elem[...] = 255 if elem > threshold else 0
    return image
def convert_and_write(image, image_name, threshold, outdir):
    im = convertToBlackAndWhite(image, threshold = threshold)
    if not os.path.isdir(outdir):
        os.makedirs(outdir)
    cv2.imwrite(os.path.join(outdir, image_name.format(threshold)), im)
#---Set Sobel X Gradient Kernel
kernel_sobelx = np.ndarray((3,3), buffer=np.array([[-1,-0,1], [-2, 0, 2], [-1, 0, 1]]), dtype=int)
#---Set Sobel Y Gradient Kernel
kernel_sobely = np.ndarray((3,3), buffer=np.array([[-1,-2, -1], [0, 0, 0], [1, 2, 1]]), dtype=int)
#---Calculate Sobel X Gradient
im3x = assignment4.computeGradient(im, kernel_sobelx)
#---Calculate Sobel Y Gradient
im3y = assignment4.computeGradient(im, kernel_sobely)
#---Calculate Gradient Magnitude
im3mag = np.sqrt(im3x**2 + im3y**2)
#---Write image for viewing
convert_and_write(im3mag, 'sobelmag_bw-{}.jpg', threshold = 150, outdir = 'SOBELMAG')
```

The image to the left is my attempt at edge detection. I attempted a couple of different variations of threshold and kernels and this one was my most satisfying result. To arrive at this image I used the sobel X gradient and sobel Y Gradient Kernels:

1. Sobel X : $\begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix}$

2. Sobel Y: $\begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix}$

I then took the magnitude of those two resulting arrays and input them into convertToBlackAndWhite (method from assignment 2) with a threshold of 150 (above which were converted to white pixels below were converted to black)

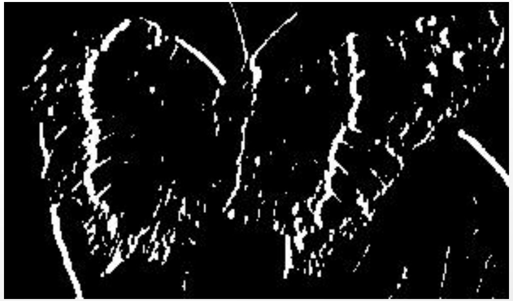
The code snippet included shows the script used to generate the image

Edge Detection Continued

I tried other kernels (Prewitt, Roberts) with varying thresholds (50, 100, 128, 150 and 200) as the cutoffs. I've included the script on the next page as well a couple of the outputted images here

Kernel = $\begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{matrix}$ Threshold = 50

Kernel = $\begin{matrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{matrix}$ Threshold = 50



Kernel = $\begin{matrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{matrix}$ Threshold = 200

Edge Detection Script

```python
import assignment4
import cv2
import numpy as np
import os

def convertToBlackAndWhite(image, threshold = 128):
    # WRITE YOUR CODE HERE.

    #---modified in place but returned to match api
    for elem in np.nditer(image, op_flags = ['readwrite']):
        elem[...] = 255 if elem > threshold else 0
    return image

def convert_and_write(image, image_name, threshold, outdir):
    im = convertToBlackAndWhite(image, threshold = threshold)
    if not os.path.isdir(outdir):
        os.makedirs(outdir)
    cv2.imwrite(os.path.join(outdir, image_name.format(threshold)), im)

im = cv2.imread('test_image.jpg', 0)
test_thresholds = (50, 100, 128, 150, 200)
for index, threshold in enumerate(test_thresholds):
    print 'start threshold = {}'.format(threshold)
    #---pewitt
    kernel_pewitt = np.ndarray((3,3), buffer=np.array([[-1,-0,1], [-1, 0, 1], [-1, 0, 1]]), dtype=int)
    im3 = assignment4.computeGradient(im, kernel_pewitt)
    if index == 0:
        cv2.imwrite('prewitt.jpg', im3)
    convert_and_write(im3, 'prewitt_bw-{}.jpg', threshold, outdir = 'PEWITT')

    #---sobel
    kernel_sobelx = np.ndarray((3,3), buffer=np.array([[-1,-0,1], [-2, 0, 2], [-1, 0, 1]]), dtype=int)
    kernel_sobely = np.ndarray((3,3), buffer=np.array([[-1,-2, -1], [0, 0, 0], [1, 2, 1]]), dtype=int)
    im3x = assignment4.computeGradient(im, kernel_sobelx)
    im3y = assignment4.computeGradient(im, kernel_sobely)
    im3mag = np.sqrt(im3x**2 + im3y**2)
    if index == 0:
        cv2.imwrite('sobelx.jpg', im3x)
        cv2.imwrite('sobely.jpg', im3y)
        cv2.imwrite('sobelmag.jpg', im3mag)
    convert_and_write(im3x, 'sobelx_bw-{}.jpg', threshold, outdir = 'SOBELX')
    convert_and_write(im3y, 'sobely_bw-{}.jpg', threshold, outdir = 'SOBELY')
    convert_and_write(im3mag, 'sobelmag_bw-{}.jpg', threshold, outdir = 'SOBELMAG')

    #---roberts
    kernel_roberts = np.ndarray((3,3), buffer=np.array([[0, 0, 0], [0, 0, 1], [0, -1, 0]]), dtype=int)
    im3 = assignment4.computeGradient(im, kernel_roberts)
    if index == 0:
        cv2.imwrite('roberts.jpg', im3)
    convert_and_write(im3, 'roberts_bw-{}.jpg', threshold, outdir = 'ROBERTS')

#---Compare to canny edge
canny_edge = cv2.Canny(im, 100, 200)
cv2.imwrite('canny_edge.jpg', canny_edge)
```

Canny Edge Vs Assignment 4 Algorithm

I also included, for comparison, the output of the Canny Edge algorithm (shown on the left). As can be seen the Canny Edge algorithm has resulted in thinner lines and more distinct edges. I think a key difference in the two algorithms is the hysteresis thresholding used by the Canny Edge algorithm. The thresholding implemented in my algorithm was simple a binary cutoff, however the Canny Edge Algorithm has a mimumn and maximum threshold input and then makes decisions in the middle of those thresholds based on connectivity.

The other difference, a Gaussian kernel, run over the image before hand, I don't think makes a huge difference in this particular image as there wasn't a great deal of noise in the moth test image.