# Hieu Nguyen
# GTID: 903185448
# Email: hieu@gatech.edu
# 9/21/15

CS-6475 Computational Photography Assignment 4

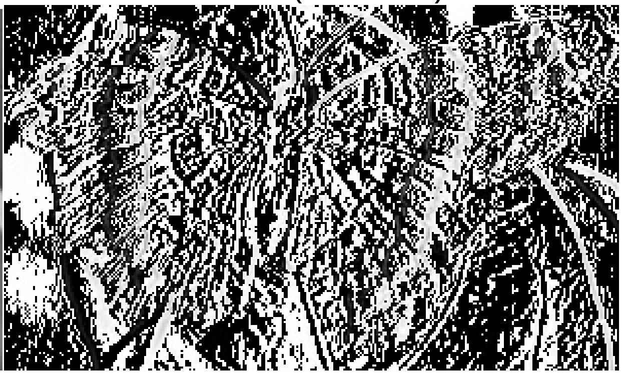
Part 1: Programming Gradients

1. imageGradientX()
   This function differentiates an image in the X direction. My approach for this function was to iterate through the number of columns in the input image and perform the absolute value of the difference between a pixel and the pixel intensity of its column+1 neighbor. Instead of iterating through every single pixel, I took advantage of numpy array slicing to select all indices along the row axis. The result is an image gradient in the X direction with a width that is one less than the original.

```
x_image = np.empty([image.shape[0], image.shape[1]-1])
for col in range(image.shape[1] - 1):
    x_image[:,col] = abs(image[:,col+1] - image[:,col])
return x_image
```

INPUT IMAGE (300x177)

OUTPUT IMAGE (299x177)

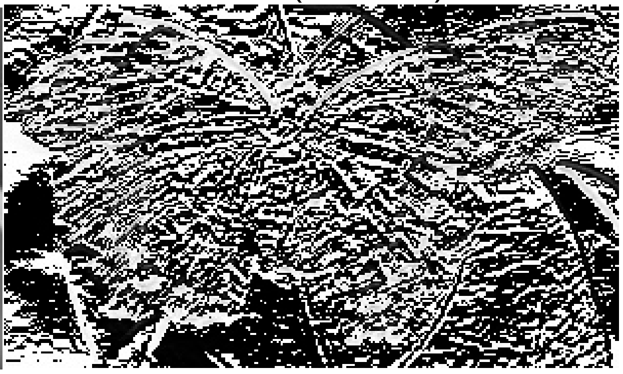


2. imageGradientY()
   This function differentiates an image in the Y direction. My approach for this function was to iterate through the number of rows in the input image and perform the absolute value of the difference between a pixel and the pixel intensity of its row+1 neighbor. Instead of iterating through every single pixel, I took advantage of numpy array slicing to select all indices along the column axis. The result is an image gradient in the Y direction with a height that is one less than the original.

```
y_image = np.empty([image.shape[0]-1, image.shape[1]])
for row in range(image.shape[0] - 1):
    y_image[row,:] = abs(image[row+1,:] - image[row,:])
return y_image
```

INPUT IMAGE (300x177)

OUTPUT IMAGE (300x176)

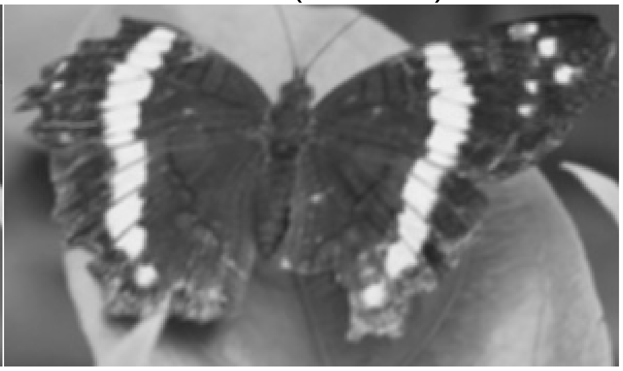


3. computeGradient()
   This function applies a 3x3 kernel to an input image. My approach was a bit convoluted (pun intended)… Because I knew the output array would be two rows and two columns smaller than the original image, I arranged my cross-correlation function to work with axes offsets. The code iterates over each pixel in the output image, and computes the cross-correlation with the input kernel. It then assigns the result of the cross-correlation to the corresponding element in the output array. The overall result is a gradient depending on what kernel is used. For example, using an averaging kernel (`avg_kernel = np.ones((3, 3)) / 9`), the output image is blurred.

```
gradient_image = np.empty([image.shape[0]-2, image.shape[1]-2])
sum = 0
for row in xrange(1, image.shape[0]-1):
    for col in xrange(1, image.shape[1]-1):
        for (kx, ky), value in np.ndenumerate(kernel):
            sum += kernel[kx,ky] * image[(row-1)+kx, (col-1)+ky]
        gradient_image[row-1,col-1] = sum
        sum = 0
return gradient_image
```

INPUT IMAGE (300x177)

OUTPUT IMAGE (298x175)

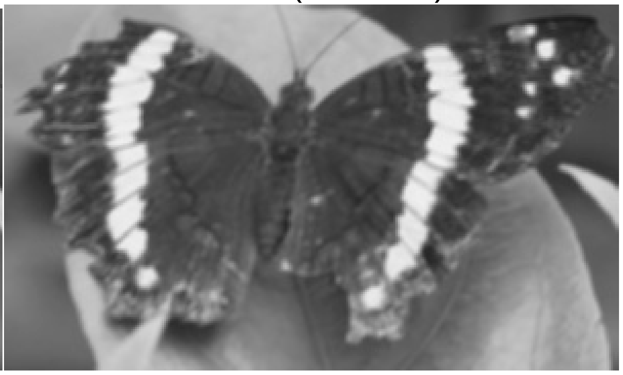


I also played around with using an OpenCV function to simplify things. It produced the correct result, however the output array had the same shape as the input array, which did not align with the problem specification.

```
gradient_image_filter2D = cv2.filter2D(image, -1, kernel)
return gradient_image_filter2D
```

INPUT IMAGE (300x177)

OUTPUT IMAGE (300x177)



Part 2: Edge Detection

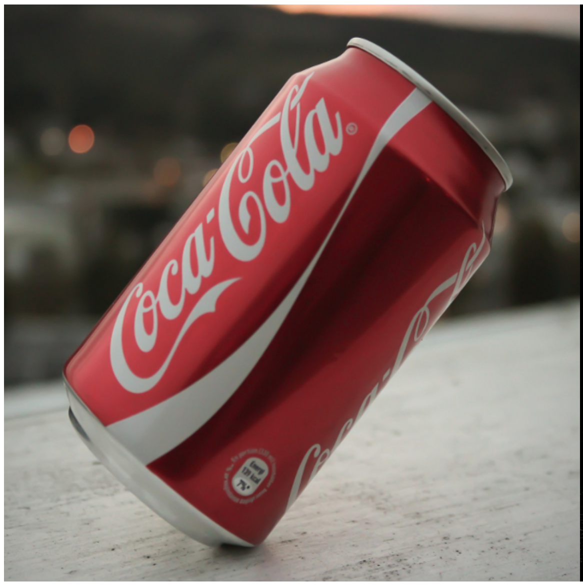
This open-ended part of the assignment called for producing an edge detected image. I explored a few different approaches to obtain the final image.

1. computeGradient()
   This approach utilized the computeGradient() function I coded in part 1 to obtain an image gradient. I utilized a kernel to generate an edge detection effect. I then ran the resultant gradient image in my convertToBlackAndWhite() function (from assignment 2) to create a binary image. I initially tested with a threshold value of 128, but some of the gradient edge information was being lost. After trial and error, I settled on a threshold of 50 to maintain the most relevant edge information in the final binary image.

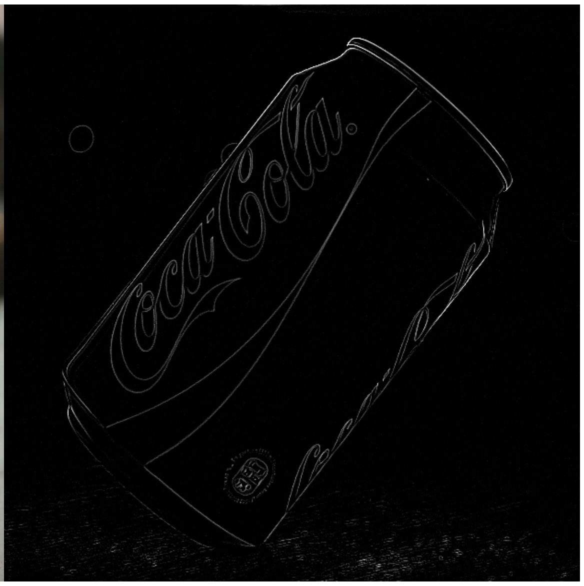
```
Kernel:
[[-1 -1 -1]
 [-1  8 -1]
 [-1 -1 -1]]

testImage = cv2.imread("coke.jpg", cv2.IMREAD_GRAYSCALE)
cv2.imwrite('gradient_image.jpg', computeGradient(testImage,
kernel))
gradientImage = cv2.imread("gradient_image.jpg",
cv2.IMREAD_GRAYSCALE)
cv2.imwrite('bw_image.jpg',
convertToBlackAndWhite(gradientImage))
```

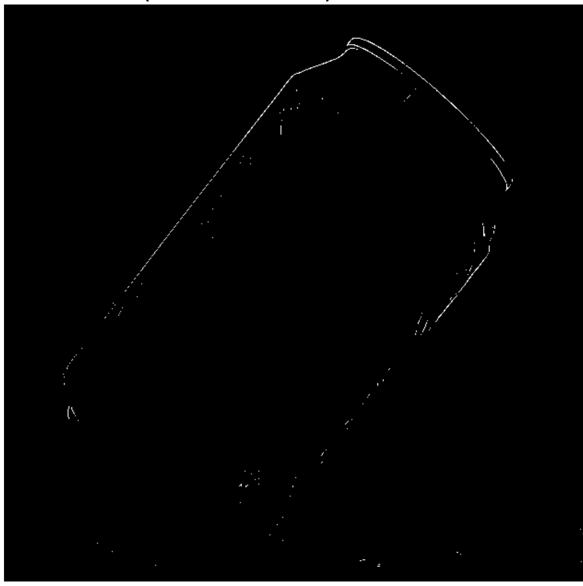
INPUT IMAGE



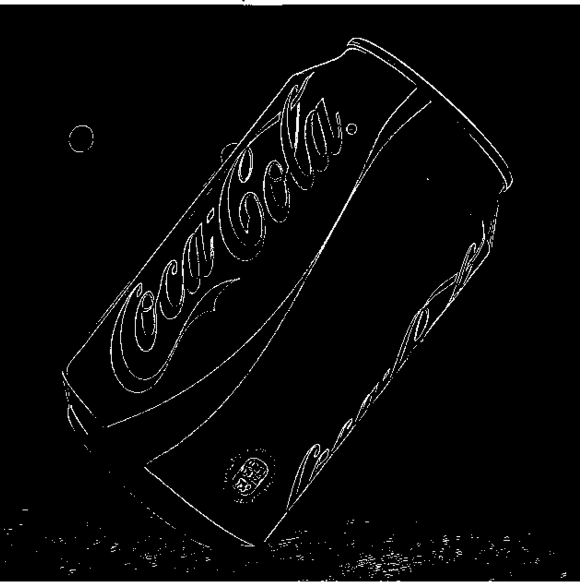
GRADIENT IMAGE



BW IMAGE (threshold=128)



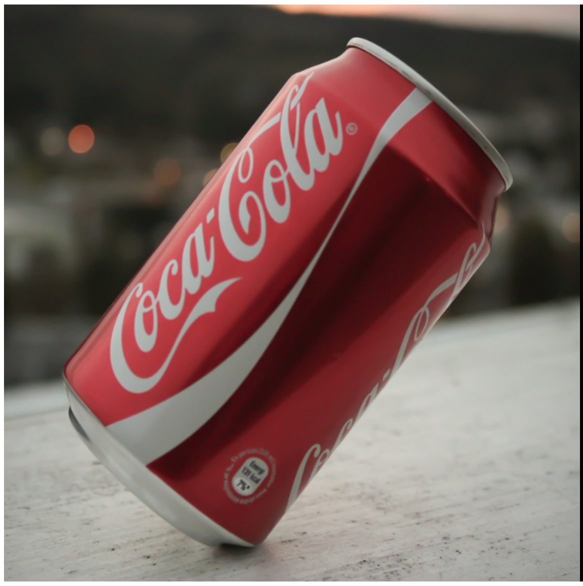
BW IMAGE (threshold=50



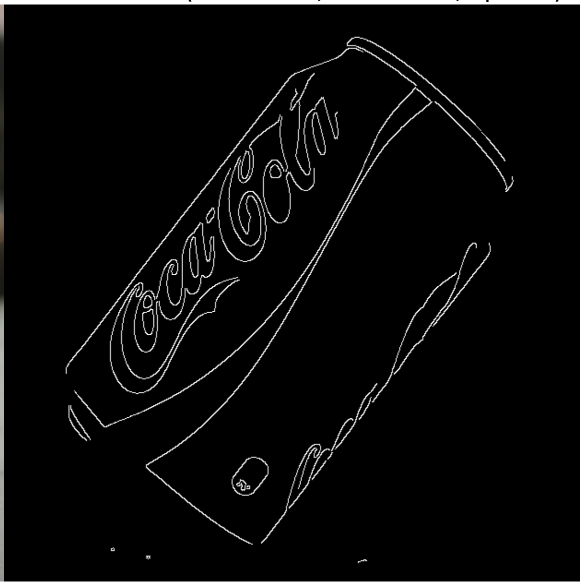
2. OpenCV’s Canny Edges
   I also experimented using Canny edge detection in OpenCV. The Canny function finds edges in the input image and marks them in the output map using the Canny algorithm. This multi-stage algorithm involves applying a Gaussian filter to smooth the image and remove noise, finding the intensity gradients of the image, applying non-maximum suppression to rid of spurious response to edge detection, applying double threshold to determine potential edges, and tracking the edge by hysteresis. The result is a much cleaner image favoring the strongest edges. I also tuned the parameters to make the logo more visible.

```
testImage = cv2.imread("coke.jpg", cv2.IMREAD_GRAYSCALE)
edges = cv2.Canny(testImage, 2000, 4000, apertureSize=5)
cv2.imwrite('canny_image.jpg', edges)
```

INPUT IMAGE



CANNY IMAGE (thr1=2000, thr2=4000, apSz=5)



CANNY IMAGE (thr1=900, thr2=3200, apSz=5)