Computer Vision
Assignment 5

Alireza Moradi

October 24, 2020

1 Roberts Operator

The Roberts operator performs an easy, fast to calculate, 2-D special gradient activity on a picture. It therefore highlights regions of high special gradient which frequently correspond to edges. In its commonest usage, the input to the operator could be a grayscale image, as is that the output. Component values at every purpose within the output represent the calculable magnitude of the special gradient of the input image at that time.

1	0
0	-1

Gx

0	1
-1	0

Gy

The small size of the masks for the Roberts Operator make it very easy to implement and quick to calculate the mask responses. However, these responses are also very sensitive to noise in the image.
Source 1 Source 2

2 Laplace

1. because we're working with second order derivatives, the laplacian edge detector is extremely sensitive to noise.
2. It does not provide information about edge direction.
3. The thresholded magnitude of Laplacian operator produces double edges.

Source 1 Source 2 Source 3

3 Isotropic

The magnitude of the gradient detects edges in all directions.
Laplacian is an Isotropic operator (equal weights in all directions).
Source

4 Implementations

4.1 Diagonal Edge Detection

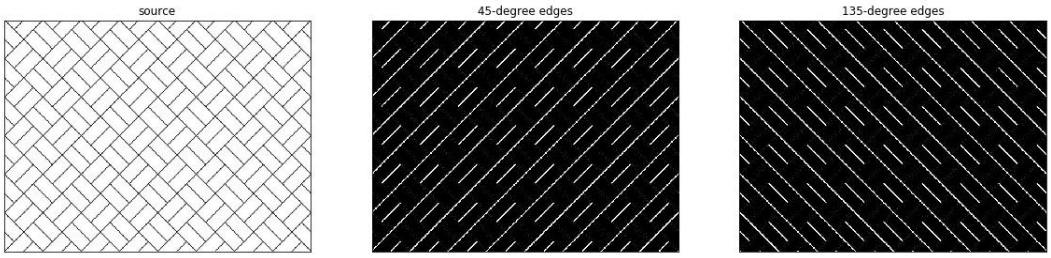
A Sobel operator to detect vertical lines looks like:

-1	0	1
-2	0	2
-1	0	1

If we apply this to a pixel (put this on top of the pixel, and use the values in neighbor pixels, then add up everything), it will only have a non-zero value if pixels in the right have different values as pixels in the left. If that is the case it means there is a vertical edge.

-2	-1	0
-1	0	1
-0	1	2

The biggest values will happen when the diagonal of this matrix has different values, thus a 45 degrees edge.
Source 1 Source 2

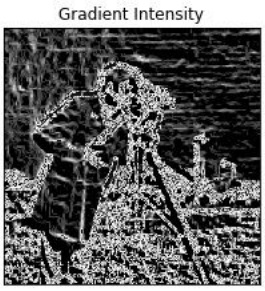


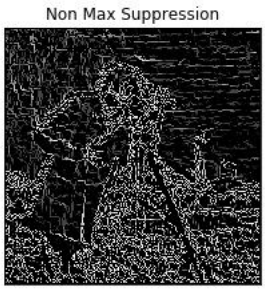

```python
def get_45_edges(image):
    '''
    Returns the image which shows the 45-degree edges.

    Parameters:
        image (numpy.ndarray): The input image.

    Returns:
        edges_45 (numpy.ndarray): The 45-degree edges of input image.
    '''
    kernel = np.array([[ 2,  1,  0],
              [1,  0,  -1],
              [0, -1,  -2]],dtype=float)
    edges_45 = image.copy()

    #Writer your code here
    edges_45 = cv2.filter2D(edges_45,-1, kernel)

    return edges_45
```

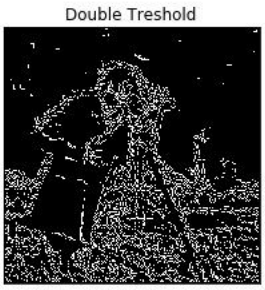
```python
def get_135_edges(image):
    '''
    Returns the image which shows the 135-degree edges.

    Parameters:
        image (numpy.ndarray): The input image.

    Returns:
        edges_135 (numpy.ndarray): The 135-degree edges of input image.
    '''
    kernel = np.array([[ 0,  1,  2],
               [-1,  0,  1],
               [-2, -1,  0]],dtype=float)
    edges_135 = image.copy()

    #Writer your code here
    edges_135 = cv2.filter2D(edges_135,-1, kernel)

    return edges_135
```

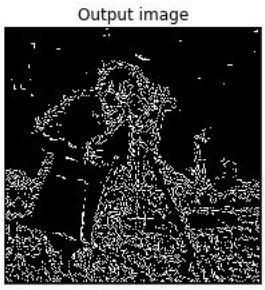
4.2 Canny

Source










```python
def G(x,y, sigma):
    pi = np.pi
    g = (1/(2*pi*sigma))*np.exp(-(x**2 + y**2)/(2 * (sigma**2)))
    return g
```

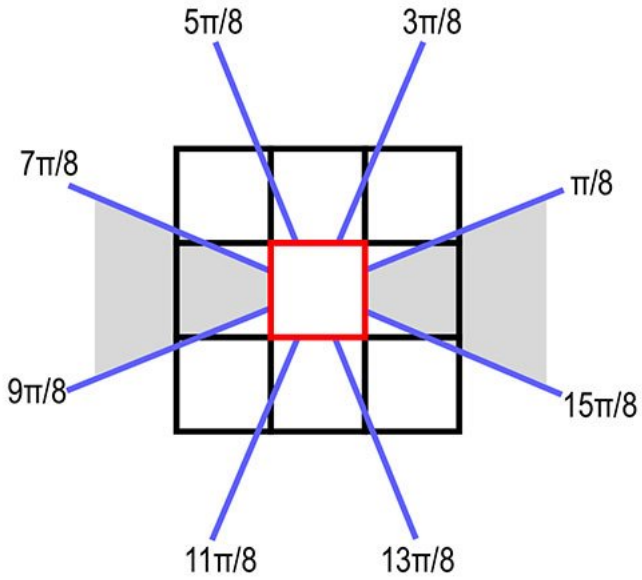
```python
def gaussian_kernel(size, sigma=1):
    '''
    Calculates and Returns Gaussian kernel.

    Parameters:
        size (int): size of kernel.
        sigma(float): standard deviation of gaussian kernel

    Returns:
        gaussian: A 2d array shows gaussian kernel
    '''
```

```
    l = size//2
    gaussian = np.zeros((size,size))
    #Writer your code here
    for i in range(size):
        for j in range(size):
            gaussian[i][j] = G(i-l,j-j,sigma)

    return gaussian
```

`np.arctan2`'s result is in radians so we have to covert it to degrees. Also `np.rad2deg`'s result is between -180 and 180 so we add 180 to make it between 0 and 360.

```
def sobel_filters(image):
    '''
    finds the magnitude and orientation of the image using Sobel kernels.

    Parameters:
        image (numpy.ndarray): The input image.

    Returns:
        (magnitude, theta): A tuple consists of magnitude and orientation of the image gradients.
    '''
    #Writer your code here

    Kx = np.array([[-1,0,1],
                   [-2,0,2],
                   [-1,0,1]])

    Ky = np.array([[1,2,1],
                   [0,0,0],
                   [-1,-2,-1]])

    Ix = cv2.filter2D(image,-1, Kx)
    Iy = cv2.filter2D(image,-1, Ky)

    magnitude = np.sqrt(np.power(Ix,2)+np.power(Iy,2))
    theta = np.rad2deg(np.arctan2(Iy,Ix))
    theta += 180
    return (magnitude, theta)
```

Magnitude orientations in each of these regions are considered to be in the same region.




```python
def non_max_suppression(image, theta):
    '''
    Applies Non-Maximum Suppression.

    Parameters:
        image (numpy.ndarray): The input image.
        theta (numpy.ndarray): The orientation of the image gradients.

    Returns:
        Z(numpy.ndarray): Output of Non-Maximum Suppression algorithm.
    '''
    #Writer your code here
    M, N = image.shape
    Z = np.zeros((M,N), dtype=np.int32)
    diff = 22.5

    for i in range(1,M-1):
        for j in range(1,N-1):
            t = theta[i][j]
            if (360 - diff <= t < 360  or 0 <= t < 0 + diff):
                back = image[i][j-1]
                forw = image[i][j+1]
            elif (0 + diff <= t < 90 - diff or 180 + diff <= t < 270 - diff):
                back = image[i+1][j-1]
                forw = image[i-1][j+1]
            elif (90 + diff <= t < 180 - diff or 270 + diff <= t < 360 - diff):
                back = image[i-1][j-1]
                forw = image[i+1][j+1]
```

```
            elif (90 - diff <= t < 90 + diff or 180 - diff <= t < 180 + diff):
                back = image[i-1][j]
                forw = image[i+1][j]
            if image[i][j] >= back and image[i][j] >= forw:
                Z[i][j] = image[i][j]

    return Z
```

Here pixels with values higher than the high threshold are mapped to 255, Pixels with values lower than low threshold are mapped to 0 and pixels in between are mapped to 100.

```
def hysteresis_threshold(image, lowThreshold=60, highThreshold=180):
    '''
    Finds strong, weak, and non-relevant pixels.

    Parameters:
        image (numpy.ndarray): The input image.
        lowThreshold(int): Low Threshold.
        highThreshold(int): High Threshold.

    Returns:
        result(numpy.ndarray): Output of applying hysteresis threshold.
    '''
    #Writer your code here
    M, N = image.shape
    result = np.zeros((M,N), dtype=np.int32)

    r, c = np.where(image >= highThreshold)
    result[r,c] = 255
    r, c = np.where((lowThreshold <= image) & (image <= highThreshold))
    result[r,c] = 100



    return result
```

I changed filter's size and sigma to have better results.

```
def canny(image, kernel_size = 3, sigma = 1, lowtreshold = 60, hightreshold = 180):
    '''
    Applys Canny edge detector on the input image.

    Parameters:
        image (numpy.ndarray): The input image.
        size (int): size of kernel.
        sigma(float): standard deviation of gaussian kernel.
        lowThreshold(int): Low Threshold.
        highThreshold(int): High Threshold.
```

```
    Returns:
        img_smoothed(numpy.ndarray): Result of applying the Gaussian kernel on the input image.
        gradient(numpy.ndarray): The image of the gradients.
        nonMaxImg(numpy.ndarray): Output of Non-Maximum Suppression algorithm.
        thresholdImg(numpy.ndarray): Output of applying hysteresis threshold.
        img_final(numpy.ndarray): Result of canny edge detector. The image of detected edges.
    '''
    g = gaussian_kernel(5,3)
    img_smoothed = cv2.filter2D(image,-1, g)
    gradient,theta = sobel_filters(img_smoothed)
    nonMaxImg = non_max_suppression(gradient, theta)
    thresholdImg = hysteresis_threshold(nonMaxImg,10,25)
    img_final = thresholdImg

    return img_smoothed, gradient, nonMaxImg, thresholdImg, img_final
```