18MCMI09
Krishnan.S
Colour Image Processing
Assignment No:1

1.

The Cumani Edge Detector is a second order edge detector, like the LoG filter. It finds out where the zero crossings of the second order differential are to locate the edges. Wherever the directional derivative is zero are where the edges are present.

Second order edge detectors are sort of like a compounding of first order derivatives. The edges formed by first order horizontal and vertical detectors are used to compute the second order one.

To start with, Cumani defines the squared local contrast S(P;n), where P is a point (x,y) in the image n = (n_1, n_2) is a unit vector, as the squared norm of the directional derivative of the image (noted as f, a function of RGB vectors) in the direction of n.

$$S(P;n) = \frac{\partial f}{\partial x} * \frac{\partial f}{\partial x}.n_1.n_1 + 2.\frac{\partial f}{\partial x} * \frac{\partial f}{\partial y}.n_1.n_2 + \frac{\partial f}{\partial y} * \frac{\partial f}{\partial y}.n_2.n_2$$

$$= En_1^2 + 2Fn_1n_2 + Hn_2^2$$

E, F and H are the double partial derivatives in the x and y directions. This leads us to a matrix

$$A = \begin{bmatrix} E & F \\ F & H \end{bmatrix}$$

The two eigenvectors and the extreme values of this 2x2 matrix are then calculated.

I think they're used because the eigenvectors gives you the direction of the tangent and the normal to the tangent at a point B (the strongest responses), and the extreme value gives you an idea on how much the intensity or edge strength "spreads". E and H correspond to the gradients in the vertical and horizontal direction, so the eigenvalues show how much the edge contrast is in those directions

The extreme values (λ) are calculated as:

$$\lambda_{\pm} = \frac{E+H}{2} \pm \sqrt{\frac{(E-H)^2}{4} + F^2}$$

The eigenvectors (n) are calculated as:

$$n_{\pm} = (\cos\theta_{\pm}, \sin\theta_{\pm})$$

$$\theta_{+} = \begin{cases} \frac{\pi}{4} & if\,(E-H) = 0, F > 0 \\ \frac{-\pi}{4} & if\,(E-H) = 0, F < 0 \\ undefined & if E = F = H = 0 \\ \frac{1}{2}\tan^{-1}(\frac{2E}{E-H}) & otherwise \end{cases}$$

$$\theta_{-} = \theta_{+} + \frac{\pi}{2}$$

After all this, we can finally get to finding edge points. The edge point is where the directional derivative D_S(P;n) of the maximal squared contrast (the extreme values) in the direction of the maximal contrast (the eigen vectors) is zero.

The directional derivative is given as:

$$D_S(P;n) = E_x n_1^3 + (E_y + 2F_x)n_1^2 n_2 + (H_x + 2F_y)n_1 n_2^2 + H_y n_2^3$$

The x and y gradients for E, F and H are computed, which is what the subscript denotes.

Finally, the edge points are detected by computing zero crossings of D_S(P;n) considering the sign of D_S along a curve tangent to the eigenvectors at the point P.

It has issues in detecting edges when theres an ambiguity in the gradient direction.

The method also applies Gaussian masks of different size to smoothen out any discontinuities in the image, which would otherwise make life hard for the gradient detecting masks (well more that it picks out a lot more things that aren't really proper edges). The two dimensional Gaussian is given by

$$G(x,y) = \frac{1}{2\pi\sigma} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The bigger the variance, the bigger will be the size of the mask and also the effect of the smoothing.
Below are examples on how the operation works on a grayscale picture with various blocks in it.
An increase in variance in the second image gets rid of a lot of the noisy edge like features in the first, and all the strong edges are retained.

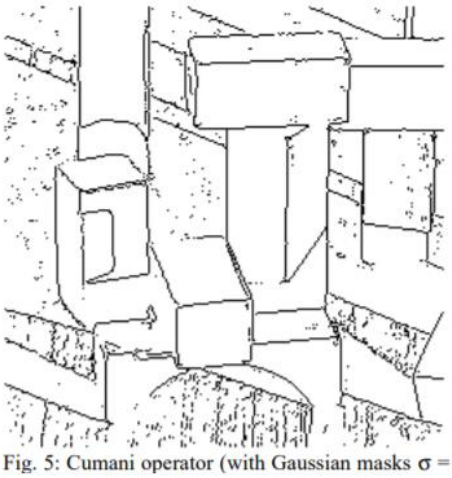


Fig. 5: Cumani operator (with Gaussian masks σ = 0.5).

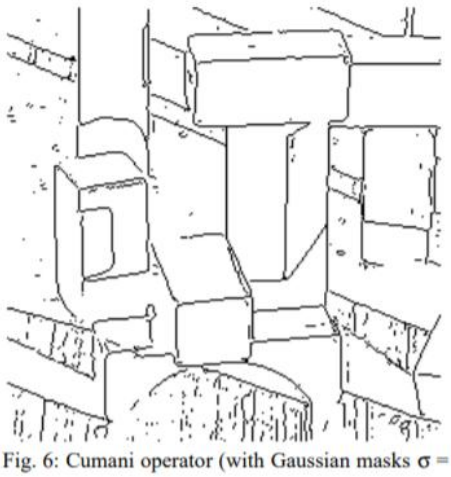


Fig. 6: Cumani operator (with Gaussian masks σ = 1.0).

I got all this information from a paper titled "A Comparative Study On Color Edge Detection" by Andreas Koschan.

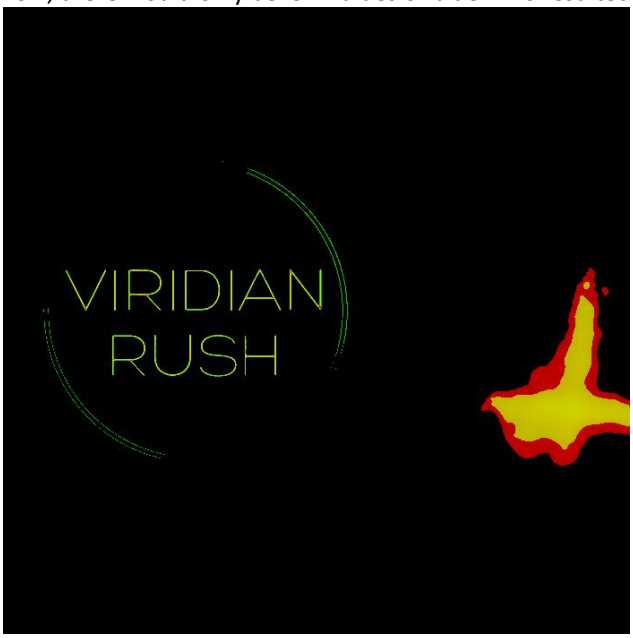
It works better than doing Sobels three time on an RGB image because it doesn't separate the components and treat them independently. The RGB components are treated as a single vector.
The same sort of thing is demonstrated in the fourth question, where some edges are picked up which aren't really edges.

The next couple of operations were implemented in python3 using OpenCV and NumPy. The code for each of the questions are in the folders labeled "Question X", along with the input and output images.

2.
As long as the intensity of the red, green or blue component independently was between the range r_c ± r_bw, g_c ± g_bw or b_c ± b_bw, it was retained. This kind of colour ranging makes it a little hard to isolate a single kind of color. If we try to get yellow, reds and greens get picked out as well. The HSV version in the next question does this better.

Erroneous ranges were clipped down to the range (0,255) using the function np.clip(), which takes the variable/array in question, and the min and maximum ranges it can have. So if r_c = 100 and r_bw = 300, it would clip it to (0,255).
The input for the operation was:

To pick out certain colours from the image, I opened it up in photoshop and found how the RGB values changed in different places.
To get the flame like part on the right side of the image, I gave the parameters (200, 20, 200, 20, 0, 0). Since its mostly yellow, there would only be low values of blue. This resulted in the following:



The white parts of the image get captured too since they contain a high red and green values. It's turned slightly yellowish green.
Next I tried to pick out just the logo, which is somewhere close to white. The parameters I gave were (225, 15, 225, 15, 225, 20), which resulted in :

The logo is slightly bluish since I retained more blue values than red or green.

3.
In this one there's only one range, which is h_c ± h_bw for the hue, and the other two parameters s_c and v_c are thresholds for saturation and value. The input image was converted into HSV using the cvtColor() function and then converted back into RGB after the ranging operation was performed. Ranging with Hue makes it a lot easier to pick out colors from the image.
Unlike RGB, Hue ranges from 0-180, Saturation from 0-255 and Value from 0-255. np.clip() was used again to make sure nothing goes out of bounds. So Red is at 0, Green at 60 and Blue at 120.
The input for the operation was a bunch of colourful pokemon figurines as shown below.

The difficulty with picking things out from this image was that the background is almost white. It stuck around while trying to pick out most colors.

The first thing I tried to do was to pick out Charizard, which is the orange dragon like pokemon on the bottom right. It's a nice saturated orange. The parameters I used were (15, 3, 160, 180). Hue changes rapidly since it's between 0-180, so the bandwidth is small. The result was :

The hue was set to zero if it didn't fall inside the range. So all the highly saturated ones that didn't fall in the hue range show up as the red parts in the image. Other than that, the orange charizard stands out nicely.
The next one I tried to pick out was Lucario, the jackal like pokemon on the top left. Nice saturated light blue, somewhere in between pure green and blue but closer to blue. The parameters I gave were(100, 5, 150, 170). The result was:

Unfortunately one of the eyes of the other pokemon was also picked out. Again charizard shows up completely red because of how saturated it was.

4.

A) The scalar median filter was applied by taking the R, G, B neighbours separately in a window of size 3x3 and then replacing the middle pixel with the median of each of the color components.
The vector median filter was applied by replacing the middle pixel vector with the vector in it's 3x3 neighbourhood that was closest to the mean of the vectors in that neighbourhood.
The input for the operations was:

This is a really noisy image, but both median filters manage to denoise this quite significantly.
The result of applying the above operations were:

Scalar Median Filter

Vector Median Filter

The results look almost the same but the vector median filter has actually removed more noise than the scalar version even though it isn't as apparent. On closer inspection you can see that some of the noise still present in the scalar median filter one isn't present in the vector median filter one (For example, the area with the reflection on the green balloon). In the vector version, it also feels like the noise intensity is less.

B) The edge detector I chose to implement was the Prewitt's Edge detector, in the horizontal and vertical direction. The mask in the horizontal and vertical look like

$$G_H = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \qquad G_V = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

The grayscale version was applied by convolving the above masks with R,G,B channels. It finds a gradient for each separately.
The vector version of Prewitts takes the averages of the three neighbouring vectors on the right and the average of the the three neighbouring vectors on the left and replaces the middle pixel vector with the difference of the two averages. This is the same operation as what the mask does, but the components are taken together instead of them being separated. This makes a really big difference.
The input for the operation was myself holding a kitten:

On applying the grayscale and vector versions of the horizontal mask , we get:

Convolved Horizontal Edges

Horizontal Vector Edge Detector

The gray scale version, because it calculates the edges for R,G,B separately, seems to be picking up the wrong edges, and the thickness of the edges is a lot more.
The vector version however does a really good job at figuring out the edges. Similarly, the vertical version of the above two operations are shown below:

Convolved Vertical Edges

Vertical Vector Edge Detector