Computer Vision – Implement Canny Edge Detection

Work in teams of two. Check each other. Submit one assignment per pair. Teach each other.
You are responsible for understanding everything you submit.

How to submit your homework:
A. Create a directory named HWNN_LastName1_LastName2.
B. Put everything in that one directory: code, and PDF of your write-up.
C. Zip up the entire directory, and submit it to the dropbox.

The standard rubric applies.

Start with the provided image. Later you will receive an email indicating another image to use for your homework too.
Use that image for all parts of the homework.

Canny Edge Detection – non maximal suppression and edge propagation:

Write a main function HW_NN_LastName1_LastName2 that calls the other functions.
Write a routine named canny_edges(input_image).

function edge_map = canny_edges(input_image, threshold1, threshold2, sigma_for_gaussian_filter);
This function returns a Boolean image. The Boolean image is true for the pixels that are important edge pixels, and false (or zero) everywhere else.

When you are done with your program, compare your results to the results you get using the
`edge( input_image, 'Canny' )` call in Matlab.

("`doc edge`" and read the documentation on the section for Canny, and look at the examples.)

You will need several variables in your program that contain copies of the image and intermediate copies of the image to use in the process of computing your final image. Do not expect to do everything with only one copy of the image.

1. **(1)** First question, before you begin. How will you know when you have it working correctly?
   Always try to begin with the end in mind. Otherwise you will never know when you are done.

   Canny edge detection can iterate forever if you are not careful. This is a design decision that you have to make.

   This is extremely important here because there are three parameters, and all of the parameters interact with each other. The process involves smoothing the image with a smoothing filter. If you change the filter, you change the resulting edge strengths that are presented to the next stage of the imaging chain.

   Can you compute an image that will tell you when you have everything correct? Can you find an image that will tell you when you have everything correct?

   Or, can you look at an image, and specify which edges must be detected in order for the algorithm you develop to be working correctly? Comparing to the Matlab routine is too stringent a requirement, and is likely to be impractical. You cannot expect to get results that are that good.

   In some cases, computing the test suite or the test images and expected results takes more time to develop then the algorithm itself.

   How will you know when you are done? Discuss this in your write up.

(continued)

2. (3) Implementation of Canny Edge Detection:

a. If the given input image is color convert it to a grayscale representation.

You can do that using something like this:

```
dimensions = size( input_image);
if length(dimensions) > 2
    you have a color image.
else
    you probably have a grayscale image.
end
```

b. Filter the image using a Gaussian filter to remove small nuisance edges:

```
fltr    = fspecial('Gauss', 7, 2 );                          % Play with both parameters here.
im2     = imfilter( input_image, fltr, 'same', 'repl' );     % This call takes a while.
```

c. For each pixel in im2, use a Sobel filter as specified in the lecture, to estimate the df/dx and df/dy of the gradient at each pixel, in each plane of the image.

Define your own sobel edge filters, because nobody else does this correctly:

```
dx = [  -1  0  1 ;
        -2  0  2 ;
        -1  0  1 ] / 8;
and its transpose for dy.
dy = dx.';
```

Use imfilter(im2, …) to find dIdx and dIdy.
dIdx = imfilter(im_in, dx, 'same', 'repl');

d. Compute the edge magnitude at each and every pixel. `im_mag = ( dIdx.^2 + dIdy.^2 ) .^ (1/2);`

e. Compute the edge gradient *direction* at each pixel (the angle). `angle = atan2( dIdy, dIdx ) * 180 / pi;`

f. Convert the edge angle to a multiple of 45 degrees. This means using the round( ) or floor( ) functions. The angle 0 should round to 0. **The angles -22.5 to ~22.5 should round to zero.**
This should give you a few angles to handle: 0, 45, 90, and 135. Check your code!!

Caution. You might want to use –dIdy there because images have a left hand coordinate system. So, test that your angles make sense with a simple images. You might quickly generate an image that has one vertical and horizontal edge in it, and test with that. You can work with angles that are wrong, but you must know what you are doing.

Round the angles to the nearest 45 degrees so that they have the values [0, 45, 90, or 135].
Negative angles can be fixed using code that looks like this:

```
bool_negative_angle = im_angle < 0;
im_angle( bool_negative_angle ) =  im_angle( bool_negative_angle ) + 180;
im_angle( im_angle == 180 ) = 0;
```

g. **Non-maximal suppression:**
For every pixel in the intermediate image, only keep edges if they have a larger edge magnitude than the pixel in front of them and behind them and if it has at least some magnitude (Threshold1). So, if the edge gradient at a location has an angle of 0 degrees, then it is a vertical edge, and you need to compare to the pixels left and right of that location (East and West of that location). If the edge gradient at a location has an angle of 45 degrees, then the edge has a slope of -1, and you need to compare to the edge strengths that are North-East and South-West. If the edge gradient has an angle of 90 degrees then you have a horizontal edge, and you need to compare to the pixels above and below that location (North and South of that pixel).

In all cases, if the edge is a maximum of the surrounding pixel edge strengths, then create an image with that point as a seed point for a new edge.

All other pixels are set to zero. You can initialize them to zero using the call to zeros…

```
seed_points = zeros( size( input_image ) );
```

All of the seed edges that are not maximum, or are below Threshold1, are set to zero as seeds. The seed points are only the local maximum edges.

h. **Apply the first threshold:** These seed edges are only the initial candidates for edge points. You need to test that they are at least as strong as the threshold1. You need to test the edge magnitude associated with each seed point to assure that they are strong enough to pass the first threshold.

i. **Edge propagation:** For every seed point that is at least as strong as the first threshold, you need to propagate the edges as long as the edges are more then the second threshold. And you need to repeat this process as long as there are changes to the edge_map.

   For each point in the current edge_map,
   based on the angle of the edge gradient, check the strength of the edges next to it. If the edges next to it are at least as strong as the second threshold, then add the neighbors to the next version of the edge_map.

   Make sure you propagate the edges in the correct direction. It is easy to propagate in the perpendicular direction by accident. ☺

   Repeat this iteratively as long as there are (significant) differences between the edge_map and the last version of the edge_map. Some people iterate until there are no changes. That might never happen.

3. **(2) Document your Code well.**
   Dr. Kinsman wants to see meaningful variable names and large blocks of code explaining your thinking.

4. **Discussion of things to watch out for:**

   Don't forget to use im2double( ) to convert the image to extra precision when you read them in.

   Test and display your results as you go to assure that you get reasonable results.

   Threshold1 may be <, <=, >, or >= threshold2. The initial restriction on the seed points that the seed point be a local maxima, and over threshold1 might be more stringent then the limitation of threshold2. Play with the values.

5. **(2) In your write-up**, show the input image, using imagesc( ); axis image; colormap(gray(256)), and the output image generated. This means you will need to run the code, and save the output using imwrite( ) to another file.

   Run your routine on two of the images from the last homework assignment. Show before and after images in your writeup.

   Discuss the results that you get with different sized blurring amounts that were applied.
   What impact does larger standard deviations cause? What do smaller standard deviations cause?

   Show different results as the parameters are changed. I would expect at least three resulting images for each of the three different parameters, to show the impact and the trends as the parameters are changed. Describe what you see, and what you learned.

   Demonstrate your learning in writing.

6. **(2) Conclusion:**
   What can you conclude about the entire process?
   What did you learn?
   What challenges did you have to overcome?