**Write-up for Programming Part**

**of Problem Set 1**

Jason Filippou

UID:112-47-72-57

E-mail:jasonfil@cs.umd.edu

**Image Smoothing**

The image smoothing routine, in file “smooth\_image.m”, is pretty straightforward. The MATLAB functions ‘fspecial’ and ‘imfilter’ are used to create a normalized 3 x 3 Gaussian filter and perform convolution with the image. It should be mentioned that the image should be converted into ‘double’ format before imfilter is used, or else ‘conv2’, called as a subroutine of imfilter, complains. Here’s the result of the test contained in “test\_smooth\_image”.

>> test\_smooth\_image

ans =

0.0050 0.0661 0.3397 0.7858 0.9983 0.7858 0.3397 0.0661 0.0050

0.0661 0.8708 4.4776 10.3582 13.1602 10.3582 4.4776 0.8708 0.0661

0.3397 4.4776 23.0227 53.2587 67.6658 53.2587 23.0227 4.4776 0.3397

0.7858 10.3582 53.2587 123.2040 156.5322 123.2040 53.2587 10.3582 0.7858

0.9983 13.1602 67.6658 156.5322 198.8760 156.5322 67.6658 13.1602 0.9983

0.7858 10.3582 53.2587 123.2040 156.5322 123.2040 53.2587 10.3582 0.7858

0.3397 4.4776 23.0227 53.2587 67.6658 53.2587 23.0227 4.4776 0.3397

0.0661 0.8708 4.4776 10.3582 13.1602 10.3582 4.4776 0.8708 0.0661

0.0050 0.0661 0.3397 0.7858 0.9983 0.7858 0.3397 0.0661 0.0050

**Image gradient**

The implementation is contained in “image\_gradient.m”. The derivative filter [0.5, 0, -0.5], corresponding to the 2nd order Taylor Series approximation of the 1D diffusion equation, is used in both dimensions of the image. Programmatically speaking, it is somewhat curious that the MATLAB function ‘conv2’ does not allow for an easy way to “pixel-pad” the image, and the programmer has to do it herself. The function padarray is used to accomplish this, and then conv2 does the rest. The superfluous rows and columns generated by padarray are dropped before the function returns.

Here are the results of the test in “test\_image\_gradient”:

>> test\_image\_gradient

Dx =

-3.5000 -6.0000 -4.0000 -2.0000 0 2.0000 4.0000 6.0000 3.5000

-3.5000 -6.0000 -4.0000 -2.0000 0 2.0000 4.0000 6.0000 3.5000

-3.5000 -6.0000 -4.0000 -2.0000 0 2.0000 4.0000 6.0000 3.5000

-3.5000 -6.0000 -4.0000 -2.0000 0 2.0000 4.0000 6.0000 3.5000

-3.5000 -6.0000 -4.0000 -2.0000 0 2.0000 4.0000 6.0000 3.5000

-3.5000 -6.0000 -4.0000 -2.0000 0 2.0000 4.0000 6.0000 3.5000

-3.5000 -6.0000 -4.0000 -2.0000 0 2.0000 4.0000 6.0000 3.5000

-3.5000 -6.0000 -4.0000 -2.0000 0 2.0000 4.0000 6.0000 3.5000

-3.5000 -6.0000 -4.0000 -2.0000 0 2.0000 4.0000 6.0000 3.5000

Dy =

-3.5000 -3.5000 -3.5000 -3.5000 -3.5000 -3.5000 -3.5000 -3.5000 -3.5000

-6.0000 -6.0000 -6.0000 -6.0000 -6.0000 -6.0000 -6.0000 -6.0000 -6.0000

-4.0000 -4.0000 -4.0000 -4.0000 -4.0000 -4.0000 -4.0000 -4.0000 -4.0000

-2.0000 -2.0000 -2.0000 -2.0000 -2.0000 -2.0000 -2.0000 -2.0000 -2.0000

0 0 0 0 0 0 0 0 0

2.0000 2.0000 2.0000 2.0000 2.0000 2.0000 2.0000 2.0000 2.0000

4.0000 4.0000 4.0000 4.0000 4.0000 4.0000 4.0000 4.0000 4.0000

6.0000 6.0000 6.0000 6.0000 6.0000 6.0000 6.0000 6.0000 6.0000

3.5000 3.5000 3.5000 3.5000 3.5000 3.5000 3.5000 3.5000 3.5000

**Gradient magnitude and direction**

Nothing special here. Since we have both the x and y components of the gradient in every pixel, the question of finding the magnitudes and trigonometric functions of the gradient consists of simple arithmetic operations at every pixel. MATLAB allows those operations to be computed at the matrix level in a very fast manner. The results of the test are (“ans” is the squared sum test suggested in the project description):

>> test\_gradient\_magnitude\_direction

R =

1.4142 2.2361 3.1623 4.1231

2.2361 2.8284 3.6056 4.4721

3.1623 3.6056 4.2426 5.0000

4.1231 4.4721 5.0000 5.6569

X =

0.7071 0.8944 0.9487 0.9701

0.4472 0.7071 0.8321 0.8944

0.3162 0.5547 0.7071 0.8000

0.2425 0.4472 0.6000 0.7071

Y =

0.7071 0.4472 0.3162 0.2425

0.8944 0.7071 0.5547 0.4472

0.9487 0.8321 0.7071 0.6000

0.9701 0.8944 0.8000 0.7071

ans =

1.0000 1.0000 1.0000 1.0000

1.0000 1.0000 1.0000 1.0000

1.0000 1.0000 1.0000 1.0000

1.0000 1.0000 1.0000 1.0000

**Finding peaks with non-maximum suppression**

This is where MATLAB is a big help for us. NMS is implemented in “find\_peaks.m”. All previous subroutines are used, in conjunction with the interpolation routine provided for us. After we have a list of all intensities, as well as the neighbor intensities along the local direction of the gradient, NMS is a one-liner consisting of a short-circuited AND operation between three operands: intensity above the threshold, above the “positive” neighbor and the “negative” neighbor. The following 4 example runs show that the entire process takes about a tenth of a second in our machine.

>> tic; find\_peaks(I, 2, 15); toc;

Elapsed time is 0.098106 seconds.

>> tic; find\_peaks(I, 2, 15); toc;

Elapsed time is 0.117181 seconds.

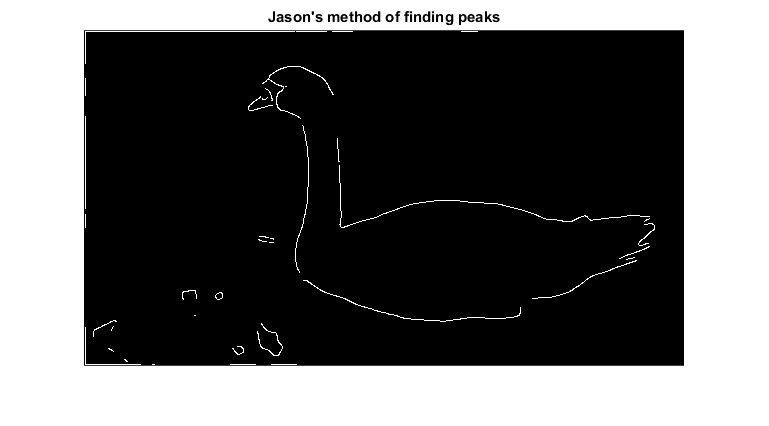
>> tic; find\_peaks(I, 2, 15); toc;

Elapsed time is 0.103951 seconds.

>> tic; find\_peaks(I, 2, 15); toc;

Elapsed time is 0.127548 seconds.

Here’s an image that shows our results:

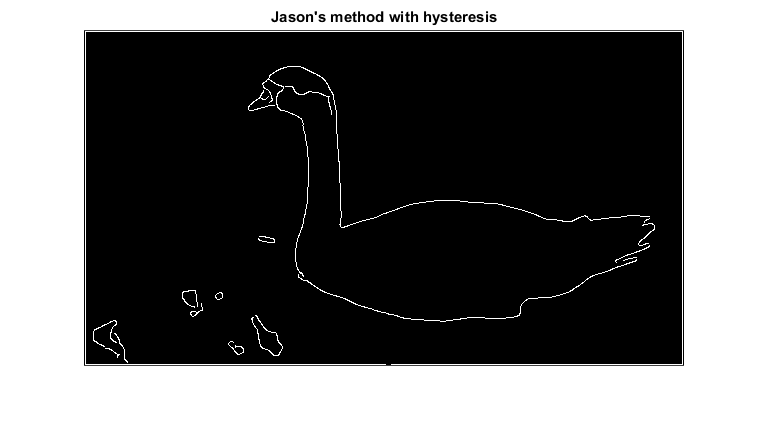


The results are quite pleasing, and we appear to be avoiding some localized artifacts present in the example image given to us. However, the algorithm doesn’t avoid considering some pixels in the border of the image as edge pixels.

**Hysteresis**

We now implement the full Canny edge detector, including hysteresis. The implementation is contained in “hysteresis.m” and is somewhat interesting. The definition of hysteresis might be naturally recursive, yet a recursive implementation blows up the stack fast. The reason is that, in the general image, including the “swan” image provided to us, it will take many recursions to find a “base case” for the recursion to stop. In our experiments, we increased the MATLAB stack to more than 10000 frames, yet the recursion always ended up overflowing the stack (or, in some more extreme cases, crashed MATLAB altogether).

So a different approach was required. We implement hysteresis in a bottom-up manner. Starting from the “hard” edges found by NMS, we subtract the image from its morphological dilation to build a square of ‘1’s around every detected edge pixel. This yields a set of “candidate pixels” . We then check whether the intensity of all of those pixels is above the provided threshold and a local maximum. The routine stops when there are no more candidate pixels to examine. The implementation is highly vectorized and runs in just under 8 seconds in our machine. Here’s the result of applying hysteresis in the swan image:



The results are similar to the ones provided, and are of high fidelity.

**Some more results with hysteresis**

We show some more results of hysteresis with MATLAB 2014b’s built-in images: 