

Problem Set 2
CMSC 426
Due September 27, 2005

Written Exercises (10 points each)

1. Consider the image below:

3	4	5	6
4	5	6	7
5	6	7	8
6	7	8	9

a. What is the gradient at the pixel in bold face (with a value of 6)?

(1,1). We can compute the partial derivative in the x direction by taking $(7-5)/2$, and we do the same thing for the y direction.

b. What is the magnitude of the gradient?

This is:

$$\sqrt{1^2 + 1^2} = \sqrt{2}$$

c. What is the direction of the gradient?

We can represent this with a unit vector in the direction of $(1,1)$, which would be:

$$\left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2} \right)$$

2. Suppose the intensities in an image can be described by the equation: $I(x,y) = (x-50)^2 + (y-50)^2$. Answer the following questions analytically, by taking derivatives of this equation.

a. What is the gradient at the position (40,45)?

$$\frac{\partial I}{\partial x} = 2(x-50) \quad \frac{\partial I}{\partial y} = 2(y-50)$$

So we plug in the values 40 and 45, and get: (-20, -10).

b. What is the direction of the gradient there?

It's easiest to do part (c) first. Then we divide the vector by this magnitude, to make it a unit vector:

$$\left(\frac{-2\sqrt{5}}{5}, \frac{-1\sqrt{5}}{5} \right)$$

c. What is the magnitude of the gradient there?

Computing the magnitude of this vector, we get:

$$\sqrt{400 + 100} = 10\sqrt{5}$$

3. Suppose there is an image, I , such that the gradient at (x,y) is always $(x,0)$.
- Give an example of a 3x3 image that would fit this description.

Well, maybe I should have asked for a bigger matrix. For a 3x3 one, we just have to make sure that everything in the second column has a gradient of $(2,0)$. But one nice way to do this is to notice that we'll get this gradient if we have the function: $I(x,y) = x^2/2$. This gives us values like:

1/2	2	9/2
1/2	2	9/2
1/2	2	9/2

b. Suppose $I(3,2) = 7$. What is the intensity at $I(5,2)$?

One way to answer this is to notice that in fact any analytic description of the answer must have the form: $I(x,y) = x^2/2 + ay + b$. So at point $(3,2)$, we have: $9/2 + 2a + b = 7$. At position $(5,2)$ we would get $25/2 + 2a + b = I(5,2)$. If we add $16/2$ to both sides of the first equation, we find: $25/2 + 2a + b = 15$. So $I(5,2) = 15$.

Another way to approach this problem would be discretely. We could say that at $I(3,2)$ the change in intensity with respect to x would be 3, so that at $(4,2)$ we would have $I(4,2) = 10$. Then the partial derivative there would be 4, so we would expect $I(5,2) = 14$. This answer is acceptable, but is different from the first one, and really less accurate, because it is based on a discrete approximation to the true function. It implicitly assumes that the gradient at $(3,2)$ is $(3,0)$, and continues to be 3 all the way until you get to $(4,2)$, where it increases to $(4,0)$. In the continuous formulation, we capture the fact that the gradient continuously increases from $(3,0)$ to $(4,0)$, so, for example, at $(3 \frac{1}{2}, 2)$, the gradient is $(3 \frac{1}{2}, 0)$.

Programming Exercises

The goal of this assignment is to write a simple edge detector based on the Canny edge detector. Your program will identify pixels in which the magnitude of the gradient is a local maxima when compared to two neighboring points in the direction of the gradient. To compute this you must smooth the image prior to finding the gradient. Then you must compute the gradient magnitude and direction. Using the gradient direction, for each pixel, (x,y) , you will compute two points, one in the direction in which the image intensity is increasing most rapidly, the other in the opposite direction, where the intensity is decreasing most rapidly. You will compare the gradient magnitude at (x,y) to the magnitude at these two points. (x,y) is an edge only if its gradient is bigger than at these two points.

Keep in mind that whenever you do image filtering, you should use the option 'replicate' for handling boundary pixels. Note that to use the test functions we describe below, you will have to name your functions the same as ours, or modify our test functions. For example, to use the test function *test_smooth_image* you will have to name your function *smooth_image* in the first problem.

Hint: It will be a good idea to turn your image into a matrix of floating point numbers using the *double* function. If not, you will get into trouble, because images are constrained to be non-negative integers below 255. If, for example, the gradient is sometimes negative, it cannot be stored in an image.

1. **Image Smoothing (15 points)** Write a function that will smooth two images with a Gaussian filter. The function should take as input an image and a value for sigma that will be used to smooth the image. You may use the function *fspecial* to create the Gaussian filter, and the function *imfilter* to perform correlation with it. You should be sure that the width of your filter is at least four sigma, to fully capture the Gaussian. For example, if sigma is 1, you might use a filter that has a length of 5 (the first odd number after $4*1$). For sigma = 2, you might use a filter of length 9.

You can test your function using the test function that we provide, *test_smooth_image*. When we execute this test function, we get the following result:

```
>> test_smooth_image
ans =
    0.0053    0.0682    0.3497    0.8090    1.0279    0.8090    0.3497    0.0682
    0.0053
    0.0682    0.8744    4.4855    10.3763    13.1832    10.3763    4.4855    0.8744
    0.0682
    0.3497    4.4855    23.0104    53.2302    67.6296    53.2302    23.0104    4.4855
    0.3497
```

```

    0.8090 10.3763 53.2302 123.1381 156.4484 123.1381 53.2302 10.3763
0.8090
    1.0279 13.1832 67.6296 156.4484 198.7695 156.4484 67.6296 13.1832
1.0279
    0.8090 10.3763 53.2302 123.1381 156.4484 123.1381 53.2302 10.3763
0.8090
    0.3497 4.4855 23.0104 53.2302 67.6296 53.2302 23.0104 4.4855
0.3497
    0.0682 0.8744 4.4855 10.3763 13.1832 10.3763 4.4855 0.8744
0.0682
    0.0053 0.0682 0.3497 0.8090 1.0279 0.8090 0.3497 0.0682
0.0053

```

Your results might be slightly different, if you produce a filter of a slightly different length than we do. But they should be quite similar. Hand in your code and the result of this test.

Sorry, there was an error in the original description of the problem. You should make the filter at least 8 sigma in length, not 4. But if you did 4, that's fine.

```

function Is = smooth_image(I, sigma)
if sigma == 0
    Is = double(I);
    % There will be no smoothing. This is especially useful for debugging
    % later functions. fspecial doesn't work with sigma = 0, so we need to
    % handle this separately.
else
    n = 1+2*ceil(4*sigma);
    % The length of the filter will be odd, and at least eight times sigma.
    Is = imfilter(double(I), fspecial('gaussian', n, sigma), 'replicate');
end
% We convert the image to double, so we don't have problems when we compute
% the gradient.

```

2. **(20 points)** Now write a function that will compute the gradient of an image. Your function should take an image as input and return two matrices. Each matrix will be the size of the original image. One will contain the x component of the image gradient for each pixel, the other will contain the y component of the gradient.

You can test your function using the test function that we provide, `test_image_gradient`. When we execute this test function we get the following result:

```

>> [Dx Dy] = test_image_gradient
Dx =

```


should return three matrices, which contain the magnitude of the gradient at each pixel, and the x and y components of the direction of the gradient. That is, the second and third results will contain the cosine of the direction of the gradient and the sine of the direction of the gradient.

You can test your function using the test function that we provide, `test_gradient_magnitude_direction`. When we execute this test function we get the following result:

```
[R, X, Y] = 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
```

Notice that with these values, we can find in Matlab:

```
X.^2+Y.^2
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
```

This is because X and Y contain the cosine and sine of the direction of the gradient, and cosine squared plus sine squared equals one. Hand in your code and the result of this test.

```
function [R, X, Y] = gradient_magnitude_direction(Dx, Dy)
R = sqrt(Dx.^2 + Dy.^2);
% The magnitude of the gradient
Rnozero = R + (R == 0);
% We will divide each gradient by its magnitude. But, if the magnitude is
% 0, we don't want to divide by 0.
X = Dx./Rnozero;
```

$Y = Dy./Rnozzero;$

4. **(20 points)** Now combine these functions together to create a function that will find positions in the image in which the gradient is a local maxima, and reasonably large. This function will take three inputs: an image, sigma, and t. sigma is the parameter used in *smooth_image* to determine how much to smooth the image. t is a threshold. We will only detect edges at pixels in which the magnitude of the gradient is greater than this threshold. The output of this function will be a binary matrix. For every pixel, there will be a 1 if there is an edge at that pixel, and a zero if there is no edge.

A pixel will be an edge if it satisfies two criteria. First, it must be a local maxima in gradient magnitude. That means that its gradient magnitude must be bigger than that at two neighboring locations. These must be locations that are a distance of one pixel away, and that are in the directions at which the gradient is changing most rapidly. For example, suppose pixel (10,10) has a gradient direction that is 45 degrees from the x axis. We would describe this gradient direction as (.7071, .7071). Then we can only have an edge at (10,10) if the magnitude of the gradient there is bigger than the magnitude of the gradient at (10.7071,10.7071), and also greater than the gradient magnitude at (9.2929, 9.2929). Note that these locations do not have integer coordinates, so we must interpolate the gradient magnitude to estimate its value at these locations. Since we haven't discussed how to do this, we will provide you with a Matlab function that performs this interpolation. This function is called: *interpolate_gradients*. Look at the comments in this function to see how to use it.

Test your function using the image of the swan in swanbw.jpg. We have provided a second image, swanedges.jpg, which shows the output of our code when we run on this image, using a value of 2 for sigma, and a value of 15 for t. Turn in your code, and an image that shows the edges that your code produces when you run with these same values. You may find it useful to use the Matlab function *imwrite*.

```
function J = local_max_gradients(I, sigma, t)
[Dx, Dy] = image_gradient(smooth_image(I,sigma));
[R, X, Y] = gradient_magnitude_direction(Dx, Dy);
J1 = interpolate_gradients(R, X, Y);
J2 = interpolate_gradients(R, -X, -Y);
% We need to find the gradients in the direction (X,Y), and the opposite
% direction, (-X,-Y).
J = (R>J1) & (R>J2) & (R>t);
% As pointed out by a student, R might have NaNs on its boundary. But
% since Matlab returns 0 for any comparison with NaN, this comparison does the right
thing anyway.
```

5. **Challenge Problem (20 points):** Now add hysteresis to your edge detector. It should take two thresholds as input, along with the image and sigma. A pixel is an edge if the gradient magnitude is greater than the first threshold, but also if the gradient magnitude is bigger than the second threshold and it has a neighbor that is an edge (note that this is a recursive definition). The image `swanedges_h.jpg` shows the results of running this function with the swan image, with `sigma = 2`, and with thresholds of 15 and 2. The results are not too different from those in `swanedges.jpg`, but a little better in the beak and head of the swan and the bottom of its body.

```
function J = edges_hysteresis(I, sigma, t1, t2)
% We use two thresholds. All pixels that are local maxima of the gradient
% magnitude are edges if they have gradient magnitude greater than t1.
% They can also be maxima if they have gradient magnitude greater than t2,
% if they have neighbors that are edges.
% This essentially implements the Canny edge detector, including
% hysteresis. This implementation is rather inefficient. It would be
% faster to trace out all the neighbors of edges explicitly, but this
% approach is very simple.
J = local_max_gradients(I, sigma, t1);
Jlo = local_max_gradients(I, sigma, t2);
cont = 1;
while cont
    N = imfilter(J, ones(3,3)) > 0;
    % N now has a value of 1 if it is an edge in J, or if it's a neighbor
    % of an edge in J.
    Jnew = N & Jlo;
    % These are all the edges in Jlo, found with a low threshold, that are
    % either also in J, or that are neighbors of edges in J, or neighbors
    % of neighbors....
    diff = sum(sum(Jnew ~= J))
    % This is the number of pixels that have become edges in Jnew that
    % weren't in J.
    if diff == 0
        cont = 0;
        % We stop iterating this when we don't get any new edges.
    end
    J = Jnew;
end
```