Image Processing

Homework 3

Fall 2017

Kynan Justis

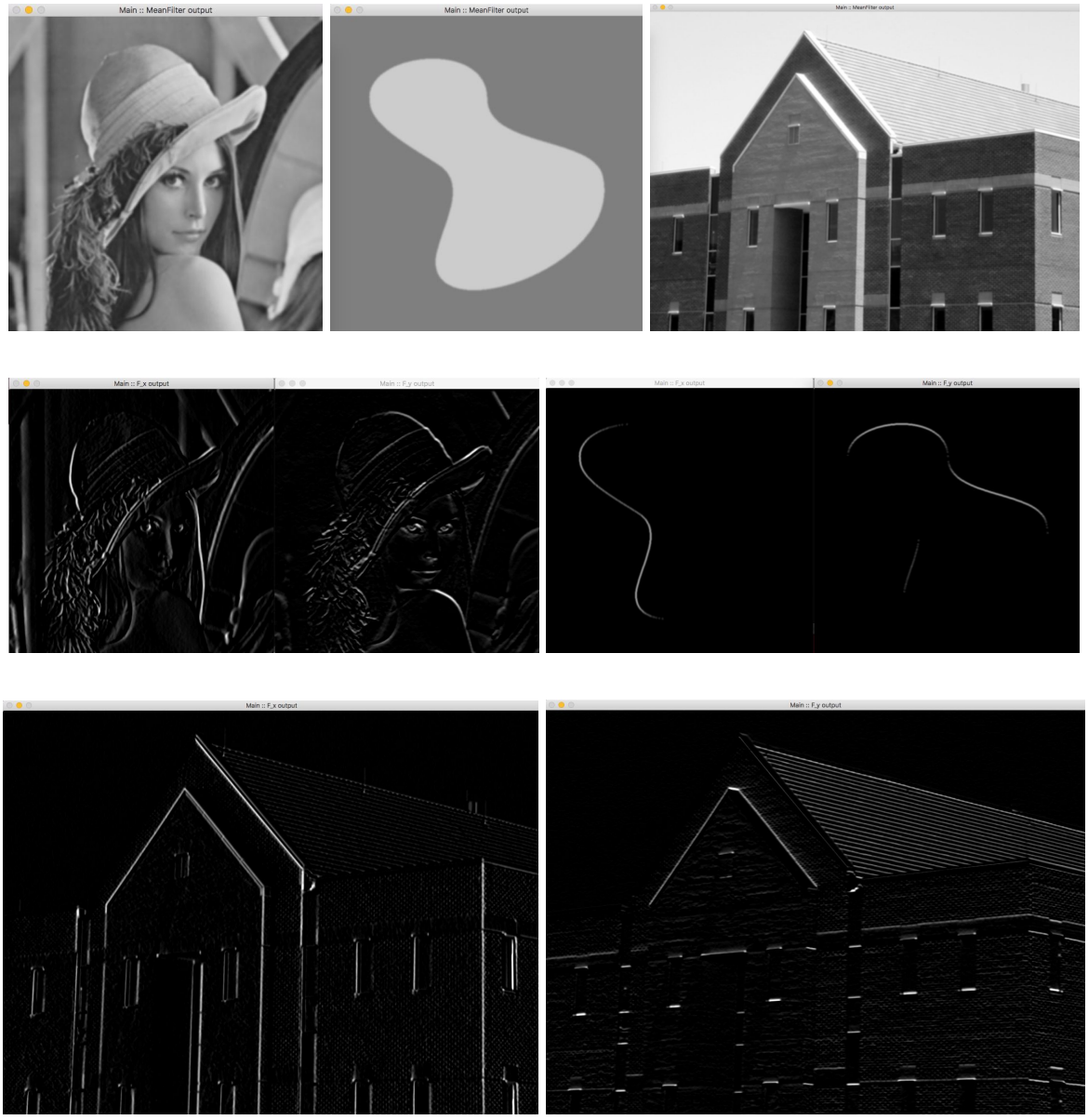
11/07/2017

Overview

The main purpose of this assignment is to implement a Canny edge detector, which itself consists of several parts. First, an input image must be filtered using a Gaussian kernel. Once filtered, the image gradient must be derived in both the x and y directions. Then, the x and y gradients are used to determine the overall gradient magnitude and orientation. Afterwards, non-maximum suppression must be applied in order to create a "thinned-edge" image using the gradient orientation as a guide. Finally, hysteresis thresholding is applied in order to isolate the edges for the output image. The result is a white line showing all of the edges in the original image with a black background, or in other words, the detected edges.

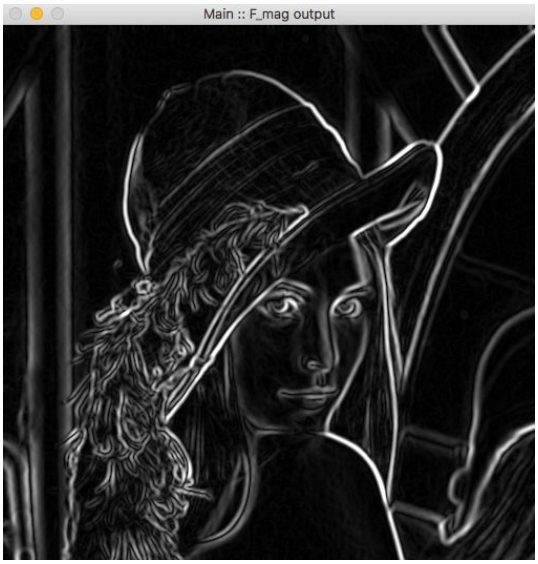
Part 1 - Filtered gradient

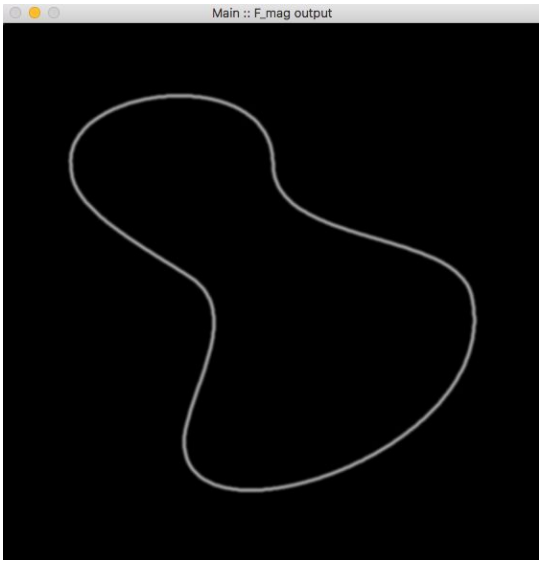
Implementing the filtered gradient was fairly straightforward. Using a similar process to the mean filtering in the last assignment, the filtered image was produced by walking over 3x3 slices of the source image. Within each 3x3 slice, the average pixel value was determined and placed at the center of the slice at the same position in the output matrix. In the provided code, the number of complete passes can be manipulated using the sigma command line input. The difference in this implementation, however, is that instead of using zero padding for the input matrix, values along the edges were duplicated instead. This was done to help prevent black edges along the filtered output image. In this phase, the Fx and Fy matrices were also calculated by simply convolving the Sobel kernel with the output of the mean filter. From this, the edge magnitude and orientation matrices were then calculated just as the formulas from the slides. These are the results of the mean filter and subsequent gradient calculations:

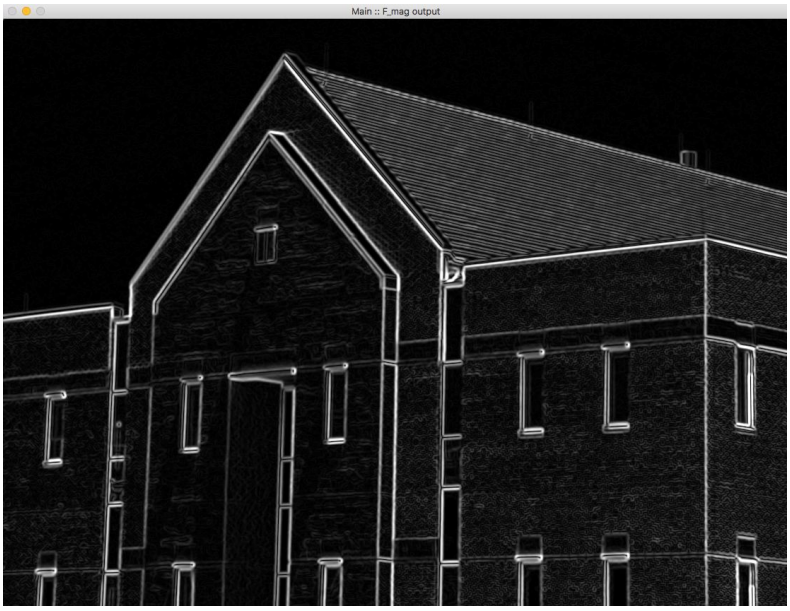


Each of these were ran through the mean filter 3 times before having their x and y gradients calculated.

The following images show the resulting gradient magnitudes:





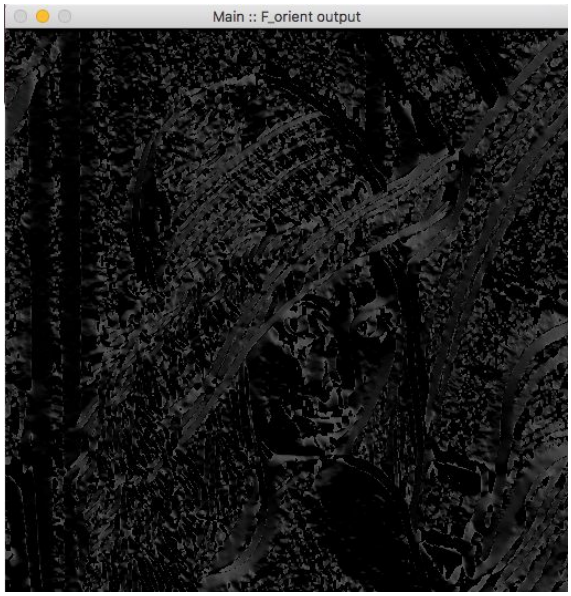


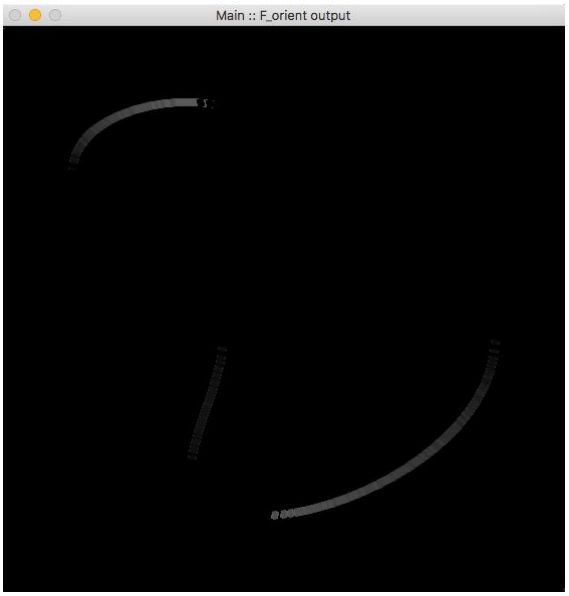
Although the implementation up to this point has garnered good results, there seems to have been an issue with how the gradient orientation is calculated. This is done with the following function:

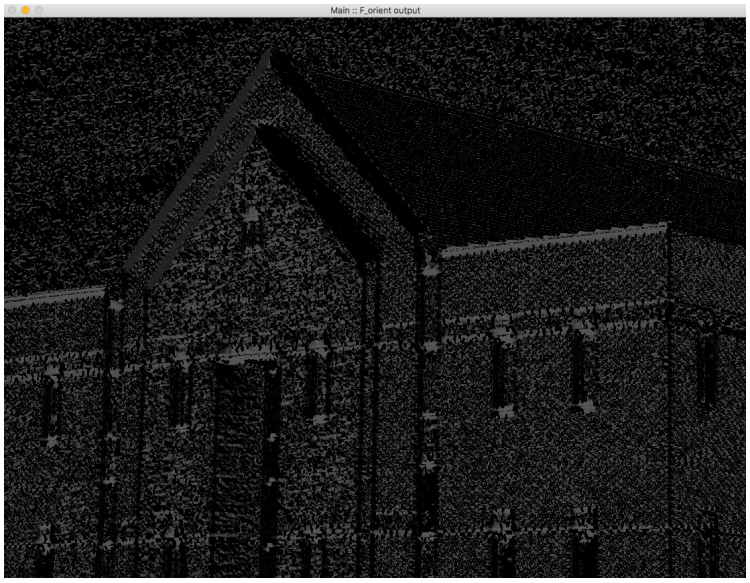
```
cv::Mat GetGradientOrientation(const cv::Mat* f_x, const cv::Mat* f_y) {
	if (f_x == NULL || f_y == NULL) {
		return cv::Mat();
	}

	if (f_x->rows != f_y->rows && f_x->cols != f_y->cols) {
		return cv::Mat();
	}

	cv::Mat F_orient = cv::Mat::Mat(f_x->rows, f_x->cols, CV_64F);

	double value = 0, temp_x = 0;
	for (int i = 0; i < f_x->rows; ++i) {
		for (int j = 0; j < f_x->cols; ++j) {
			// calculate orientation in degrees and store in F_orient
			temp_x = f_x->at<double>(i,j);
			if (temp_x == 0) {
				temp_x = 1.0;
			}
			value = atan(f_y->at<double>(i,j)/temp_x) * 180.0 / PI;
			F_orient.at<double>(i,j) = value;
		}
	}

	return F_orient;
}
```

The main work of this function is handled in the double for-loop, which is responsible for visiting every pixel value between the two passed matrices (f_x and f_y). These two matrices contain the respective x and y gradient values. Since they are the same size, the loop condition only uses the length of the rows/cols for the f_x matrix. Inside, the current pixel value for the divisor is checked and clamped to 1 to make sure division by zero does not occur. Using the formula from the slices, a new pixel value is calculated with arctan and stored in an output matrix after being converted to degrees. However, the output images from this function do not appear to line up with the expected output in the slides. One noticeable detail is in the shoulder area. If the output lena image from this code is compared to the image in the slides, the pixel

intensity values are quite different. Since the formula appears to be correct, there may be a bug in how the pixel values are being stored. Sometimes the orientation calculation would result in negative numbers, but there was not enough time to test how those value played with the rest of the functions versus values that were strictly positive. Further testing would likely help to remove this bug. The resulting orientation outputs can be viewed below.

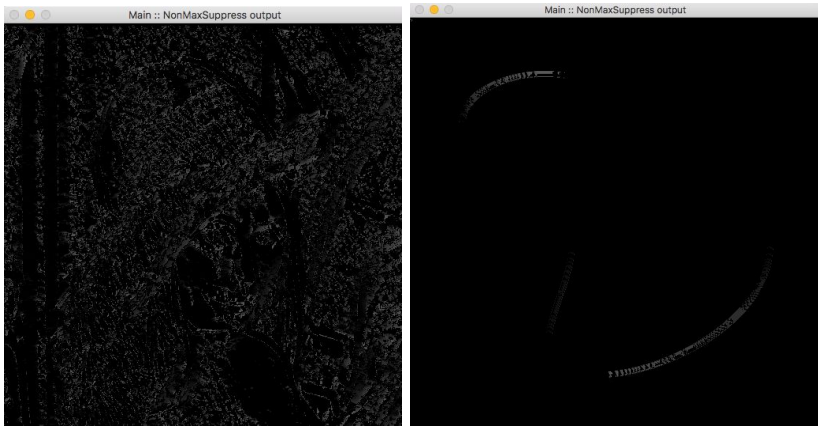


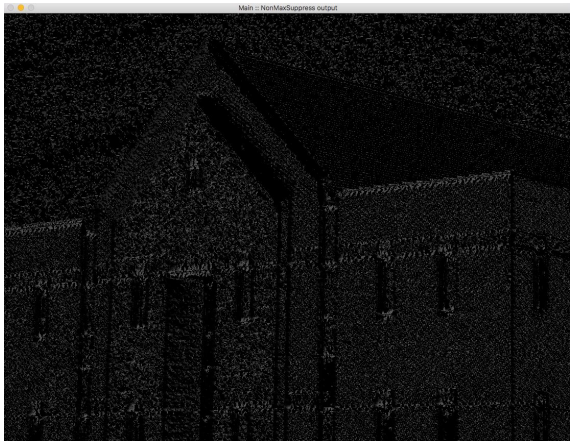




Part 2 - Non-maximum Suppression

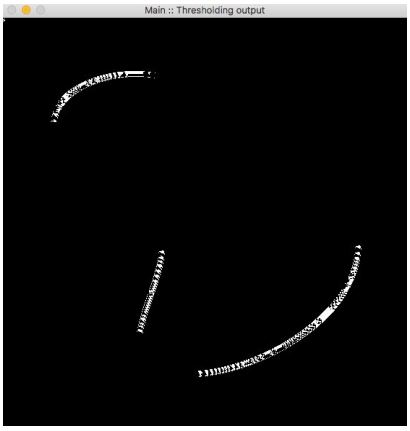
Given the problem with calculating the gradient orientation, there were issues with implementing the non-maximum suppression phase as well. The purpose of this step is to clearly define where the edges lay in the image based on the orientation of the gradient at any given point. By examining along the orientation of the gradient, the largest pixel value can be selected and stored in the resulting image, while the lesser values are simply set to 0. Due to this process, the resulting image should have clearly visible lines, albeit with improper intensity values. Unfortunately, many areas remain quite noisy after this process as shown in the resulting output below. It's possible that the implementation for this step was merely done in incorrect fashion relative to the given class slides, but more testing is needed to rule out other potential bugs.





Part 3 - Hysteresis thresholding

Finally, the last step is called hysteresis thresholding. This thresholding is a process whereby the output from the non-maximum suppression is processed. As each piece of the image is processed, the program follows lines that are greater than a given high threshold and no less than a given low threshold. As long as the orientation of the gradient is the same along a given line, the intensity values are set to maximum whereas all other values are set to 0. Given proper implementations of the previous steps, the resulting output should be the final image containing the detected edges from the source image. However, with the implementations provided, the resulting image is much more cluttered. Noise managed to make it through the non-maximum suppression process, and as such, still exists in the final image. The results are shown below using a high and low threshold of 150 and 100 respectively.



Note

As with the submission for Assignment 1 and Assignment 2, all of the results for the various filters and source images can be found here in the output folder: https://github.com/kfjustis/ip3 If you want to build the project with CMake, assuming there is a valid installation of OpenCV and CMake, download the project, make a folder called “build” inside. Then, change the directory into the build folder, run “cmake ..” and then “make”. The binary will be in “build/src”.