Image Enhancement Toolbox in MATLAB

By: Tyler Bryk and Michael Eng

CPE462 - Image Processing & Coding

Professor Hong Man

I pledge my honor that I have abided by the Stevens Honor System.

~Tyler Bryk, Michael Eng

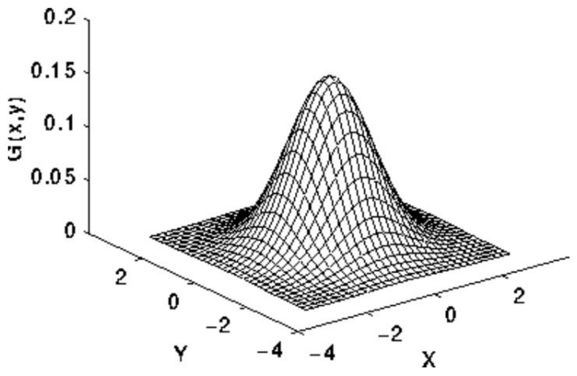
Introduction

Image enhancements play a critical role in modern day image processing. Such enhancements offer better image perception, and increase the overall visibility by highlighting key elements of the image. Currently, most image editors already include standard image enhancement filters. For example, almost every photo editor has a blur tool which can fade out a portion of the background. Other types of image enhancement tools include equalization, noise filters, linear contrast adjustment, and mask filtering. In this project, an Image Enhancement Toolbox was created which encompasses all of the above operations, in addition to Gaussian filtering and edge detection. The Gaussian smoothing operator is a 2-D convolutionary operator that is used to blur images, remove detail, or add noise. In this sense, it is similar to the mean filter, but it uses a different kernel to represent the shape of a Gaussian (bell-shaped) curve. The particular filters and their implementations are discussed within this paper.

Gaussian Filter

The Gaussian filter was implemented to perform image smoothing, noise addition, and noise reduction. The isotropic Gaussian distribution is shown below in figure 1, along with its 2-D equation. In the formula, σ is the standard deviation of the distribution. It is also assumed that the distribution has a mean of zero (*i.e.* it is centered on the line *x* = 0 and *y* = 0).

Figure 1: Gaussian Distribution & Formula



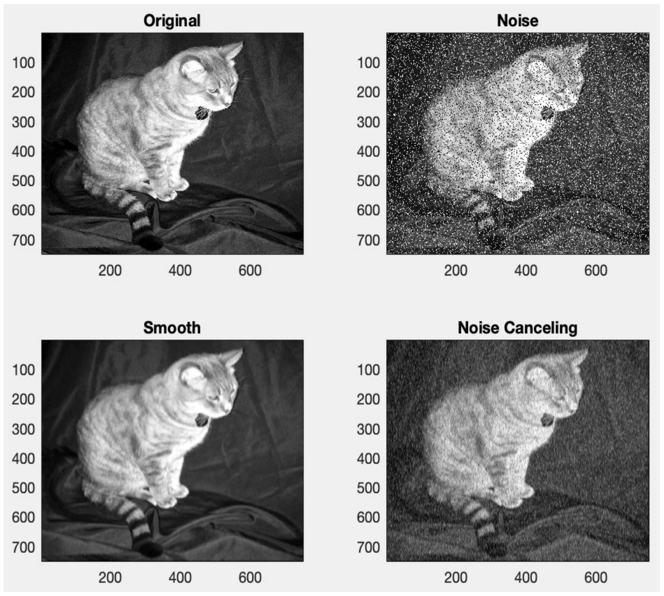
$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The idea of Gaussian smoothing is to use a 2-D distribution as a ‘point-spread’ function, which is achieved by convolution. Since the image is stored as a collection of discrete pixels, a discrete approximation to the Gaussian function needs to be produced before the convolution can be performed. In theory, the Gaussian distribution is non-zero everywhere, which would require an infinitely large convolution kernel, but in practice, it is effectively zero more than about three standard deviations from the mean, therefore, the kernel can be truncated at this point.

Once a suitable kernel has been calculated, the Gaussian smoothing can be performed using standard convolution methods. The convolution can be performed fairly quickly since the equation for the 2-D isotropic Gaussian shown above is separable into *x* and *y* components. Thus, the 2-D convolution can be performed by first convolving with a 1-D Gaussian in the *x* direction, and then convolving with another 1-D Gaussian in the *y* direction.

The Gaussian filter was applied to two different images, the original and the original with added salt & pepper noise. The results of the filter are demonstrated in figure 2. Regardless of the input image, the filter requires input parameters such as the x and y range for the Gaussian curve, the standard deviation with respect to each direction, and the angle of rotation, theta. Once these parameters are inputted, a Matlab function applies them to the equation shown in figure 1. The result of this equation is stored in a separate array and the 2-D convolution is taken to achieve an output. The salt and pepper noise effect was achieved by randomly making pixels either white or black in the image.

Figure 2: Gaussian Noise Removal and Smoothing



Edge Detection Filter

The goal of edge detection is to mark the points in a digital image where the luminous intensity changes sharply. Rapid changes in image properties usually reflect important events and changes in properties of the world. These include discontinuities in depth, discontinuities in surface orientation, and changes in material properties and variations in scene illumination.

Edge detection in an image significantly reduces the amount of data, and filters out information that may be regarded as less relevant, preserving the important structural properties of an image. There are many methods for edge detection, but most of them can be grouped into two categories, search-based and zero-crossing based. The search-based methods detect edges by looking for maxima and minima in the first derivative of the image, usually local directional maxima of the gradient magnitude. The zero-crossing based methods search for zero crossings in the second derivative of the image in order to find edges, usually the zero-crossings of the Laplacian or the zero-crossings of a non-linear differential expression.

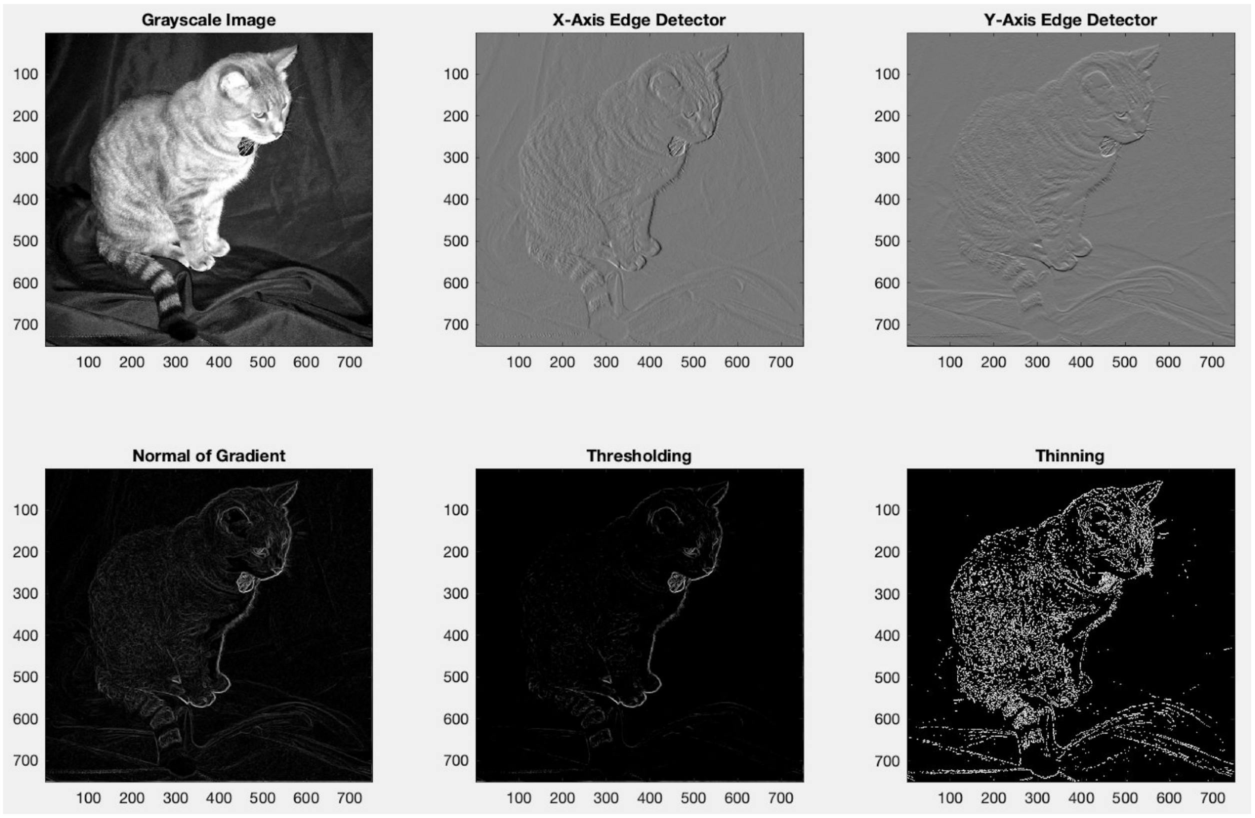
An edge detection module was created as a Matlab function. The function returns a 2-D edge detector in the form of a first derivative of whatever is inputted. The function requires parameters such as x and y range, standard deviations in both directions, and the angle of rotation, theta. The edge detector then takes the first order derivative of the Gaussian function and applies the inputted parameters to achieve a result. This result is stored in an array and can be applied in a variety of ways to achieve different filtering effects. Some of the techniques and results used are shown in figure 3.

For both the X and Y direction edge detectors, the edge detection module explained above is called to instantiate an array of edge detected pixels. A 2-D convolution is then performed between the original image and this newly created edge detection array. For testing purposes, the same parameters were used for both directions, so the same edge detector was applied in each direction to show the side-by-side comparison. It is important to note that this edge detection was performed on a grayscale variant of the original image. To perform a real edge detection technique on the images, the normal gradient of the image was found. The results of the x and y direction edge detectors were saved and the first order derivatives of each were found. To find the normal gradient of the image, the x and y direction edge detectors were multiplied by their respective first derivatives and the square root of that result was taken.

Once the true edge detection was performed on the image, a thresholding filter was applied to further distinguish the edges by removing subtle edges. The thresholding filter requires a user input threshold value to compare all of the pixel values to. For increased accuracy, the user can input a threshold percentage such as 10% and the program will calculate the thresholding value by multiplying the threshold percentage by the median value. The median value can be found by subtracting the minimum pixel value from the maximum pixel value and then by adding that to the minimum value. The thresholding filter offers an enhanced edge detection module which is more distinguished to the human eye.

Lastly, a thinning filter was separately implemented to show a morphological variant of the original image. Similar to an edge detector, a thinning filter removes foreground pixels and displays the essential skeleton of the image. The normal gradient edge detector values were reused to indicate the local maximum in the first derivative of the image. In the thinning process, the local maximum pixels were interpolated with the local minimum pixels and that result was convoluted with the original image. The results of the entire edge detection process are shown in figure 3.

Figure 3: Applications of Edge Detection Filtering



Noise Tool and Motion Blur Filters

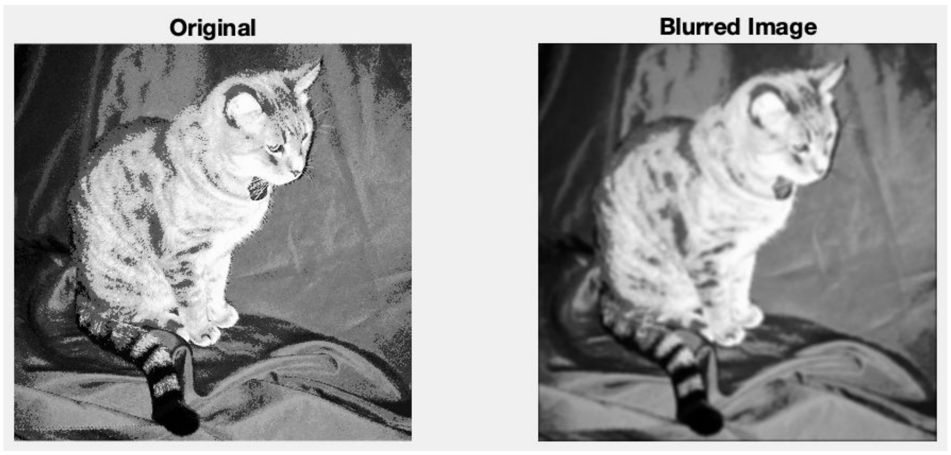
In an effort to accurately test some of the filters designed by the team, noise was required to be present. To solve this problem, the team created a noise tool which can perform several different noise operations on an image. Some of the noise operations include salt & pepper noise, additive white gaussian noise, motion blur, and averaging.

For the salt & pepper noise filter, pixels were randomly changed to be either black or white. The user can input a parameter which specifies the percentage of pixels that should be converted to noise. Using this percentage, a random number was generated between 0 and 100. If the random number was less than the probability, then the pixel was converted to noise. Another random number would be generated to determine if the selected noise pixel would become white or black. The results of the filter are shown in figure 2 along with the Gaussian noise removal tool.

The team added white Gaussian noise to an image in a similar fashion to the salt & pepper noise filter. A normal distribution with mean 1 and variance 1 was created to help generate random numbers. A user defined magnitude or intensity is input to the program and multiplied by the random number generated from the normal distribution. This number is then added to the original pixel value for each pixel in the image. The resultant image seems to be more white in color and contains some white speckles throughout the image.

In terms of development, the motion blur tool is quite similar to the smoothing filter. The inputted image is stored in an array of pixels, and a 3x3 mask filter is used to average the values of all neighboring pixels. The blurred result can be seen in figure 4.

Figure 4: Motion Blur Tool Result



Contrast Enhancement

In modern day image processing, contrast is used to make certain elements of images stand out to the human eye. Applying a contrast filter to an image typically amplifies the dark and light tones of the image while smoothing out the intermediate pixels. The enhanced resulting image will look the same in terms of sharpness and structure, but the focal point of the image should be much clearer and more distinguished.

The team implemented a contrast enhancing filter in Matlab by amplifying the darkest and lightest pixels in the image. As the magnitude of contrast increases, dark pixels get darker and light pixels get lighter. Several different types of contrast enhancing filters were implemented as shown in figure 5.

For the negative filter, the team subtracted 255 from all pixel values in the image and then took the absolute value of that result. Performing that operation essentially inverts the image by making all dark pixels light and all light pixels dark.

For the stretched mid-range filter, the team split the image into two groups containing pixels above the value 128 and below the value 128. The contrast magnitude was then added to the group above 128 and subtracted from the group below 128. The resulting image amplifies both the dark and light tones in a linear fashion.

A logarithmic transform filter takes all of the pixel values and multiplies them by a logarithmic constant known as $c = 255/\log(1+255)$ (The log is base 10). After multiplying each pixel by this constant, the resulting values are rounded down to the nearest integer and the resulting images illustrates a lightened version of the original input.

An intensity slicing filter behaves similarly to the linear contrast filter, however, rather than separating pixels based on a common threshold, they are grouped by pixel value ranges. An arbitrary intensity factor was chosen to multiply the pixel values by. Both the upper and lower 40th percentiles of all pixels were multiplied by this intensity factor. The middle 20% range was multiplied by the intensity factor plus a constant which resulted in an enhanced mid-range contrast. It is important to note that the team chose to break the image up into two groups of 40% and a mid-range of 20%, however, in a typical intensity slicing filter, the ranges can vary in size.

Lastly, a histogram equalization filter was implemented by plotting the original histogram of all pixel values and then by normalizing them. To do this, the team used a predefined function in Matlab which takes all of the pixel values and maps them to a histogram. The results of this filter are shown in both figure 5 and figure 6.

Figure 5: Contrast Enhancement Results

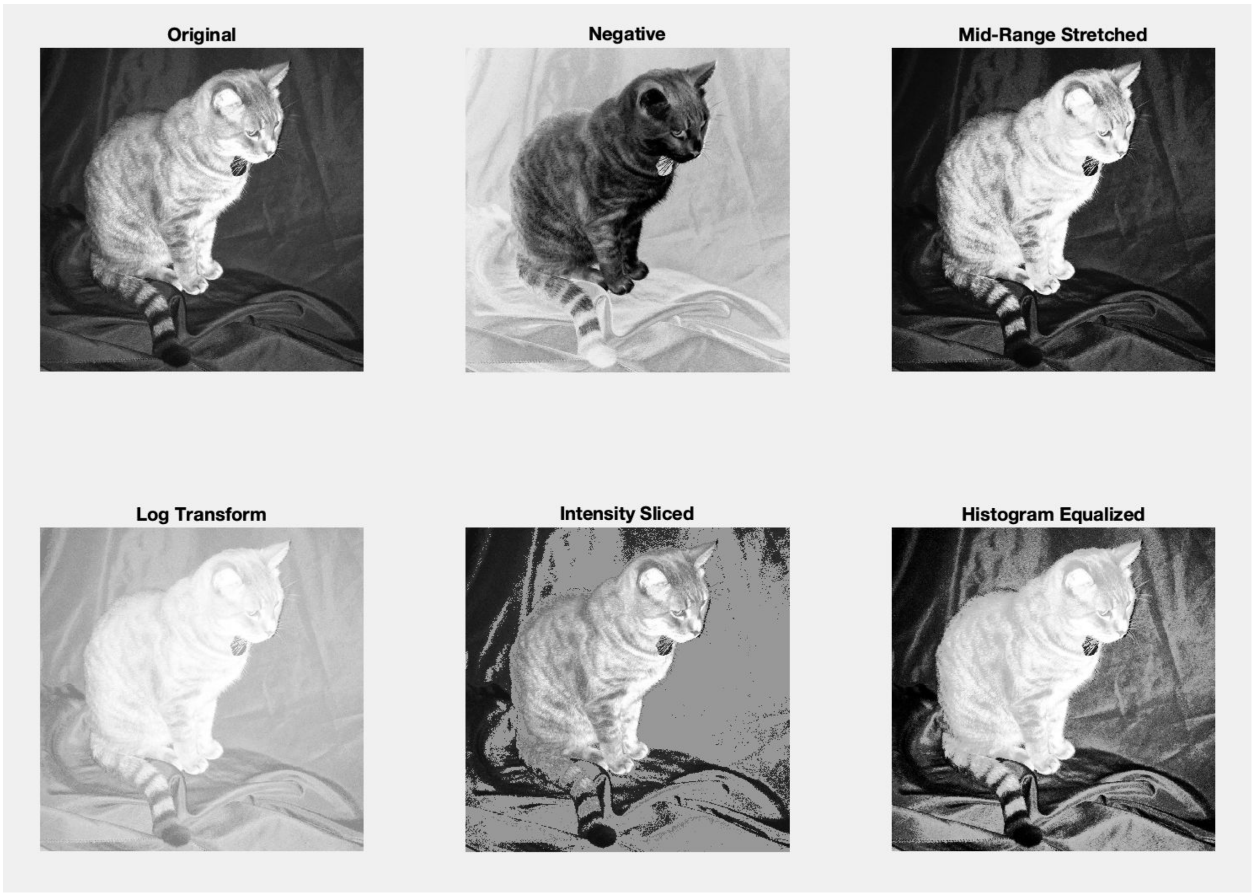
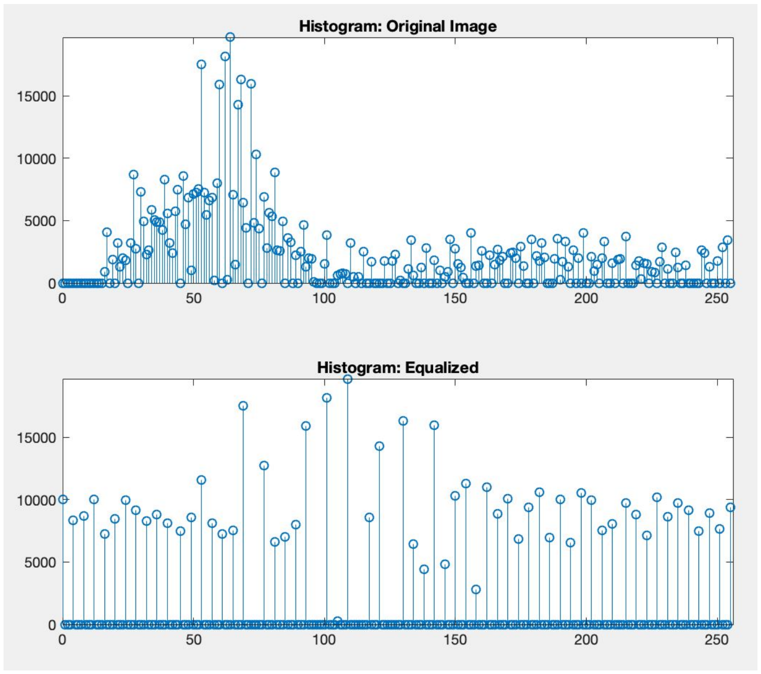


Figure 6: Contrast Enhancement Histogram

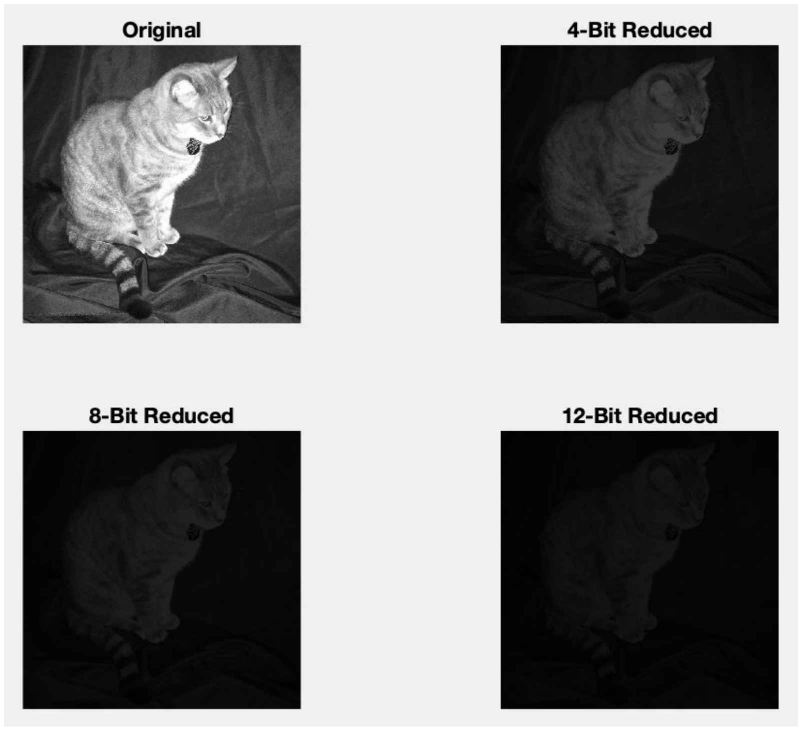


Bit Depth Reduction Filter

High quality images often require large amounts of disk space to store. In an effort to reduce the amount of disk space required by these images, compression filters are used to reduce the number of bits in an image while attempting to maintain as much detail as possible. Some compression filters are vastly complicated and use techniques such as run-length coding. For the scope of this project, a more simplistic filter was applied which reduces the bit depth of the image from n-bits to a smaller number of bits per layer.

The team created a bit depth reduction filter in Matlab by first reading the image into a Matlab array. The resulting pixel values were then divided by a constant known as the bit reduction rate which would yield in a reduced bit depth for the image. A sample of the teams bit reduction filter can be seen in figure 7. The test image was sampled and reduced by factors of 4-bits, 8-bits, and 12-bits. As expected, the image gets darker as more bit layers are reduced because the pixel values are essentially decreasing by that factor.

Figure 7: Bit Depth Reduction Results



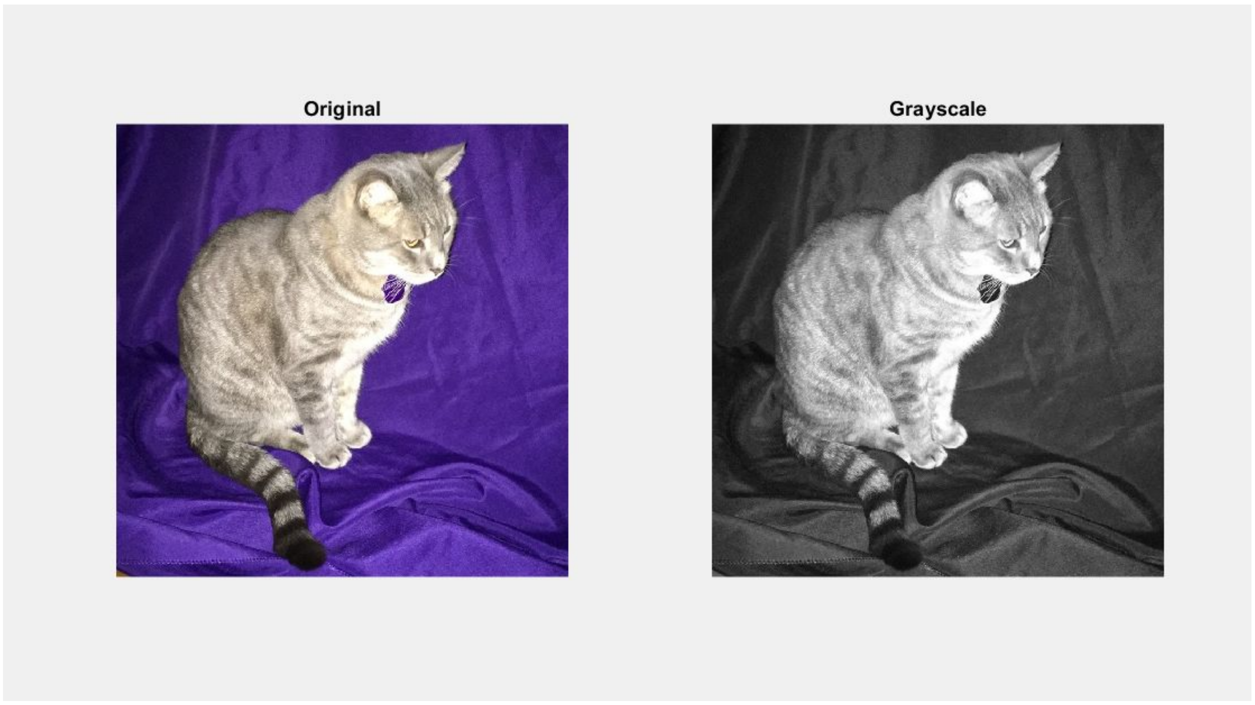
Grayscale Filter

A grayscale filter is used to reduce the image to only black and white values. It is important because other filters such as Laplacian, Gaussian, Motion Blur and Noise filters require a grayscale filter to operate. The grayscale filter replaces the RGB value of each pixel with its hue value. The hue of a pixel is the actual wavelength of light that should be emitted from a diode. Every pixel contains a value between 0 and 255 which represents the ratio between the wavelength of violet light at about 350nm and red light at about 780nm. This is important so that the other filters only need to operate on the hue value instead of 3 color values for each pixel.

The code for the grayscale filter is very simple, each of the three RGB values are multiplied by a specific number, according to the ITU standard, then added together resulting in the hue of that pixel.

The result is a grayscale image which keeps the quality of the original image while only keeping one value for each pixel instead of 3. The results of the grayscale filter can be found in figure 8 .

Figure 8: Grayscale Filter Results



Median Filter

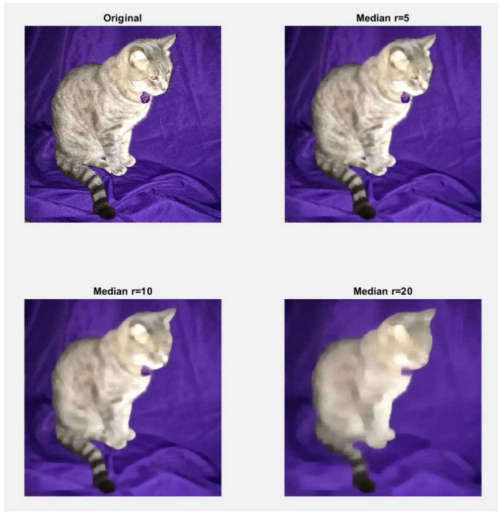
The Median Filter is a noise reduction filter which is able to keep edges. It assigns the median or middle value of group of pixels to the center pixel. Various shapes can be used, such as a circle or square of pixels around the main pixel. A square was used for this project as it is easier to change the radius of the square. If a circle was used, the group of pixels measured would have to be mapped for every radius.

The variable n is used to represent the radius of pixels from the main pixel that is extended from the matrix. The matrix is a square with width and height of 2n+1;
so if n = 3, a 7x7 box of pixels around the pixel of operation are collected. To keep the filter clean and robust, the teams median filter still applies to edge pixels, however any pixel outside of the range will default to 0. It would be more effective to alternate pixels between 0 and 255 as to keep the median as the actual median.

Four nested for-loops were used to calculate the median for each pixel. The first two for-loops were used to operate on each pixel thus iterating as many times as there are pixels in the image. The third and fourth loops would capture the matrix around each pixel from the first two for-loops. The number of iterations for the third and fourth for-loops depends on the size of n. Inside every iteration of each for-loop, three matrices would collect data, a matrix for each red, green, and blue color values.

The images in figure 9 show the original image quality. A median filter with parameters of n = 5, n = 10 and n = 20 is then applied to the image. It is important to note that although the image becomes more blurry as the n value is increased, the edges are still very distinguishable.

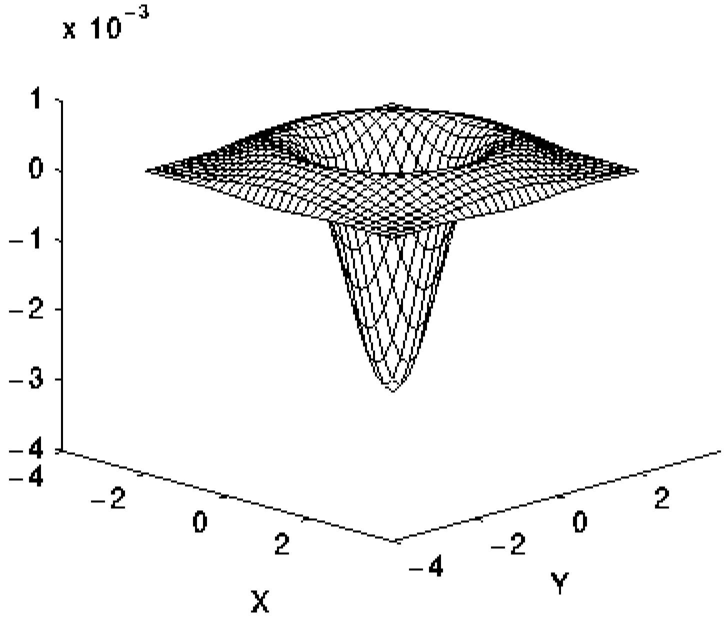
Figure 9: Median Filter Results



Laplacian Filter

Unlike computers, humans are able to distinguish objects in an image with ease. Using a Laplacian filter, however, a computer can distinguish the edges of objects. The Laplacian filter performs a convolution between a mask, containing the integer estimation of the Laplacian curve found in figure 10, and an area around each pixel to create an image that contains the edges of objects from the original image. Each edge is the value of the second derivative of the convolution. This means that one side of the edge will contain large negative values, while the other side will contain large positive values. It is also important to note that the farther away the edge is from the current pixel, the lower the values get.

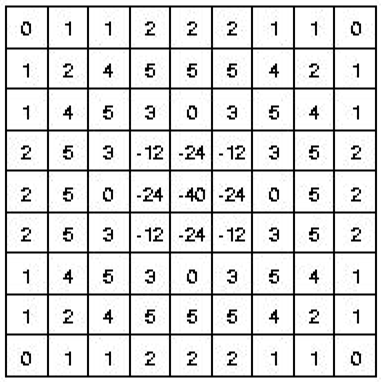
Figure 10: Laplacian Curve



The convolution part of the filter operates on pixels around a central pixel. It takes the summation of each cell in the mask multiplied by that cells corresponding pixel. If the value is negative, the area around the pixel is darker than the center pixel. If there is an edge somewhere in the scope of the mask, the value of the convolution will be either a large negative integer or large positive integer.

Two masks were used to demonstrate the effectiveness of the filter. The 3x3 mask is very simple as there is a central value -8, surrounded by 1. It is hard to estimate a function as complex as the Laplacian Curve effectively with such a small matrix, so the larger 9x9 matrix illustrated in figure 11 was also used.

Figure 11: Laplacian Curve Estimation Matrix



0	1	1	2	2	2	1	1	0
1	2	4	5	5	5	4	2	1
1	4	5	3	0	3	5	4	1
2	5	3	-12	-24	-12	3	5	2
2	5	0	-24	-40	-24	0	5	2
2	5	3	-12	-24	-12	3	5	2
1	4	5	3	0	3	5	4	1
1	2	4	5	5	5	4	2	1
0	1	1	2	2	2	1	1	0

Initially, two images were generated: one for positive values, and one for negative values. However, this method was incorrect as the result of the convolution greatly exceeded the 8-bit value each pixel can hold. In extreme cases, the result of the convolution can be positive or negative 2,040 for a 3x3 convolution matrix. This value was originally entered directly into the resulting image which meant that a lot of data was lost. The result must be scaled, or normalized, from the range of -2,040 through 2,040 to 0 through 255.

The code for the Laplacian Filter is very similar to that of the median filter. Four for-loops are nested, two to iterate through each pixel, and two to perform a 2D convolution of the 9x9 Laplacian Curve Estimation Matrix from figure 11 on the pixels around each pixel from the original image. For every iteration of the mask, the summation of each pixel convolution is taken only if the mask does not extend outside of the image. For instance, on the first pixel, columns and rows 1 through 4 will be outside of the image thus not adding to the total sum.

Two of these four nested for-loops are needed to normalize the result. The purpose of the first massive nest of for loops is only to find the most positive and most negative convolution summations for the image. For the 9x9 matrix, the possible range of summations would be from -46,920 to 41,820, however, most images will not approach even close to this range. In fact, the image that was used to demonstrate the filter, achieved a maximum positive value of 15,898 and negative value of 20,506. If the result were to be normalized using the largest possible range, the edges would not be as pronounced.

This range must now be normalized to match the range of amplitudes a pixel can contain which is 0 to 255. To do so, the adjusted range in this case -20,506 through 15,898 is shifted to 0 through the summation of the absolute value of each max. The resulting range would be from 0 to 36,404. The scaling of this range to the range of a pixel can be completed in the second group of nested for-loops once the quotient of the new adjusted max by 255 is found, which is 142 for the image.

The second group of nested for-loops uses the most negative summation of the image and the scaling factor to normalize each resulting pixel. Let's say the sum of the convolution for a random pixel was 5,000. The value of 20,506 would be added to 5,000 resulting in 25,506. This value would then be divided by 142 to be normalized to a final pixel value of 179.

The final image will contain pixels amplitudes from 0 to 255 where 0 and 255 portray an edge, or sharp contrast between a pixel and its neighbouring pixels. Pixels which contain amplitudes in the middle near 127 will represent areas of no variation. The top middle images in figures 12 and 13 are the result of the 3x3 and 9x9 Laplacian Curve matrices respectively. It is important to note that the edges are a lot more defined in the latter matrix. This is as expected because the matrix covers more area and is a more accurate estimate of the Laplacian Curve than the former matrix. It is still hard to see the edges in these images. Humans are not good at visualizing these results, so to understand where the edges actually are, pixels with amplitudes between plus or minus n of 127 are assigned with the value 127. For the bottom 3 images in both figures 12 and 13 the value of n was increased to filter out non-edges. As expected, the resulting edges from the 9x9 matrix are a lot easier to see as the middle pixels are filtered out. The result of the 3x3 matrix, where pixels with values 102 to 152 are changed to 127(n = 25), looks very similar to the result where n = 0. This is vastly different to the resulting image from the 9x9 matrix where n = 25 as only sharp edges are visible.

Figure 12: 3x3 Laplacian Filter

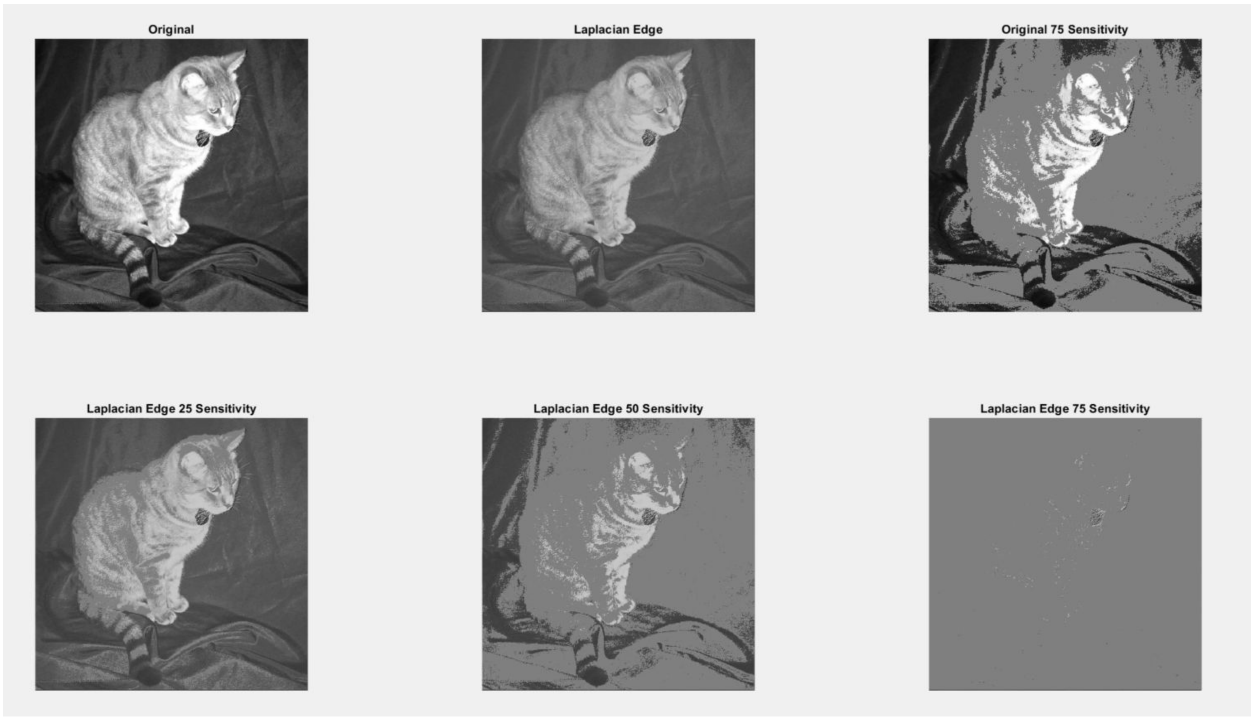
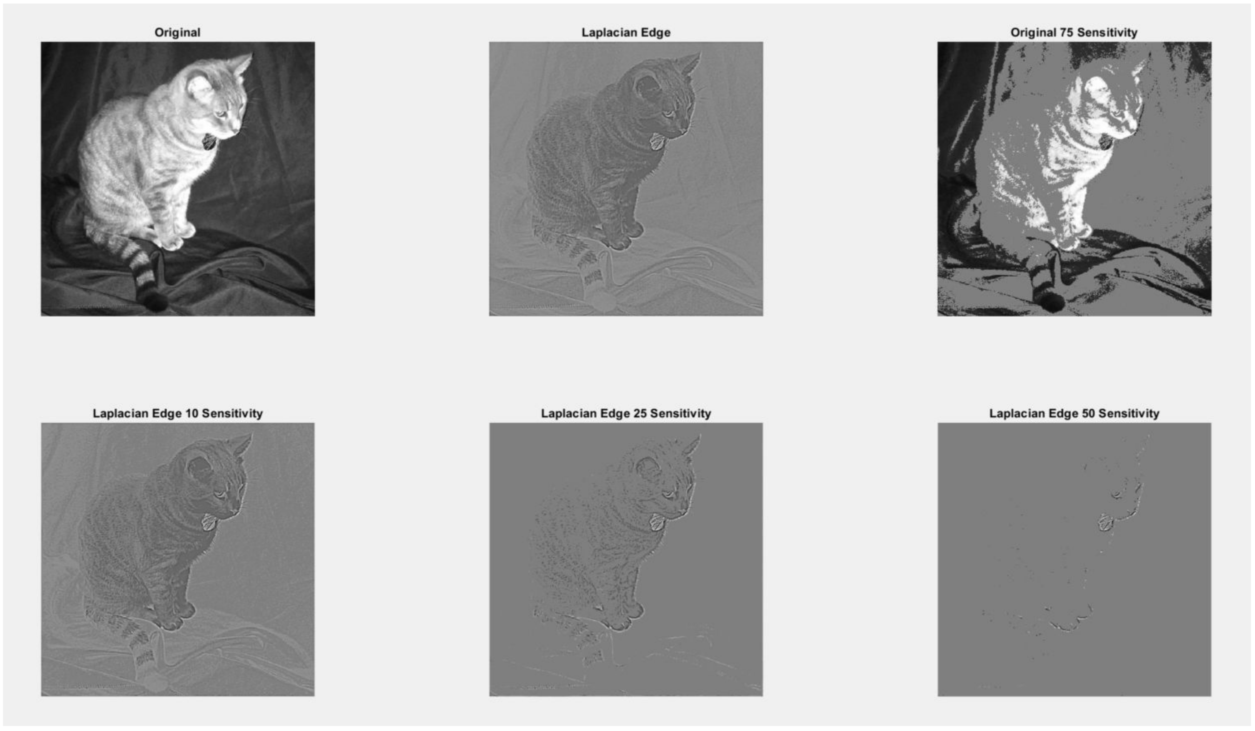

Original
Laplacian Edge
Original 75 Sensitivity
Laplacian Edge 25 Sensitivity
Laplacian Edge 50 Sensitivity
Laplacian Edge 75 Sensitivity

Figure 13: 9x9 Laplacian Filter


Original
Laplacian Edge
Original 75 Sensitivity
Laplacian Edge 10 Sensitivity
Laplacian Edge 25 Sensitivity
Laplacian Edge 50 Sensitivity

Conclusion and Analysis

Overall, the project was successfully deployed as an image enhancement toolbox that contained multiple different enhancing filters. Some of the enhancements provided in the toolbox were filters that were previously discussed in class while others were new concepts that the team had to research. For example, the team researched the Gaussian filter independently to more accurately model a Gaussian filter that could offer noise addition, noise removal, and smoothing. The Gaussian filter was effectively implemented and the team concluded that the magnitude required by the program was relatively large in comparison to what was expected. For instance, after adding white noise and salt & pepper speckles, the magnitude required by the filter to remove this noise was nearly 70% of the filters highest capable magnitude. The team expected this to be much lower which would result in a more effective filter. Overall, the filter worked but not to the degree of accuracy that the team was shooting for. Thankfully, Matlab offered support by making it easy for images to be decomposed of into individual pixel matrices. It is fairly intuitive to perform operations on this pixel matrices based on the knowledge presented in class. The team did notice, however, that the run time for the Gaussian filter and edge detection filter was noticeably longer than that of the other image enhancement filters.

Based on the topics covered in class, the edge detection module was instantiated with the help of the previously defined Gaussian filter. The team derived the Gaussian equation provided in class and further expanded upon the filter by implementing a thinning and thresholding function. The team was surprised by how well the filter was able to remove noise from an image and correctly identify the edges. The gradient of the edge detector came out fairly noisy, however, applying the thresholding function to the image greatly reduced the amount of noise and yielded a very distinguishable variant of the image. The team also noticed that as the input parameters were varied, as in the Gaussian range was increased, the image diminished in quality. Equally, as the standard deviations were increased, the image lessened in quality. Overall, the edge detection filter was optimized to work for a range of n1 = 10 and n2 = 10 with standard deviations of 1 and a thresholding value of 127.

In full, the image enhancement toolbox worked as the team expected and was fairly intuitive to implement. Some filters such as the Gaussian filter and Laplacian filter were more complicated to design and they certainly took more CPU power to process, as well as time to compute. Other filters like the contrast enhancement and blur tool were simplistic to create, however, it took several iterations to find the optimal magnitude that would yield in the expected result. The remaining filters worked without exception and performed in a desirable manner. If more time was given, the team would work on bettering the toolbox by incorporating all of the filters into a GUI that displays the result in real time and allows for magnitude and parameter adjustment. Overall, the toolbox was successful and the team was satisfied with the final results.

Resources

[1] Alderson, Edward H, et al. “Pyramid Methods in Image Processing.” *MIT*, MIT, 1984, persci.mit.edu/pub_pdfs/RCA84.pdf.

[2] Fisher, Robert. “Gaussian Smoothing.” *Spatial Filters - Gaussian Smoothing*, 2003, homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm.

[3] Fisher, Robert. “Laplacian/Laplacian of Gaussian.” *Spatial Filters - Laplacian/Laplacian of Gaussian*, 2000, homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm.

[4] Michigan. “Laplacian of Gaussian Filter.” *Laplacian of Gaussian Filter*, Michigan University, 14 Feb. 2001, academic.mu.edu/phys/matthysd/web226/Lab02.htm.

[5] MIT. “Image Filtering & Edge Detection.” *MIT Alumni*, 2007, alumni.media.mit.edu/~maov/classes/vision09/lect/09_Image_Filtering_Edge_Detection_09.pdf.