Project Report

Edge Detection and Character Recognition

Ernest Carozza, Chandroutie Sankar

ECSE, Rensselaer Polytechnic Institute

Troy, NY, USA

[caroze@rpi.edu](mailto:caroze@rpi.edu)

[sankac@rpi.edu](mailto:sankac@rpi.edu)

**Abstract**: Edge detection involves extracting the edges or outlines of an image. An edge exists when there is a sharp change in pixel intensities (image brightness) of an image i.e. there is a discontinuity in the image intensity function. Edge detection is important because it can provide us with information about the events and changes in the properties of the world. Discontinuities in image brightness can indicate a change in depth, a change in surface orientation, a change in the material’s properties, and/or a variation in scene illumination. Edge detection can also be used in image enhancement, and this will be one application of edge detection in our project. This project involves an exploration and analysis of the different types of edge detectors and a comparison will be made of the different types of methods. In our project we will develop a demonstration that shows the different types of edge detectors. The user can test our various edge detector algorithms using various types of images: images of the real world, images of computer generated shapes, and the user can even use his/her image. The user can choose to do edge detection with or without noise, and can even experiment with different types of smoothing filters. It is our hope that this demo will educate and generate curiosity and interest in edge detection. It is our intention to provide a comprehensive package about edge detection, and the user can explore the different factors of edge detection in its various permutations. The result of our algorithms will then be applied to image enhancement. An image can be enhanced by adding the edges back to the original image, and this process is called crispening. We will also apply of edge detection algorithm to character recognition.

**1. Introduction**

Edge detection is an important aspect of image processing because it allows us to extract information from an image. The information extracted can tell us about events and changes in the properties of the real world. Edge detection can allow us to filter out the relevant information in an image and this can significantly reduce the amount of data that has to be processed. Edge detection involves identifying the edges or outlines of objects in an image and determining its orientation [2]. An edge exists when there is a sharp change in the image intensity or image brightness; i.e. a discontinuity in the brightness occurs from one pixel to the next. However, edge detection is not a simple matter. Images may have false edges, or they may contain edges that have been fragmented. Some images may have a gradual change in intensities and as such would not produce very good edges, or the image can contain many small edges. Additionally, during acquisition, the image can become corrupted by noise. One of the goals of edge detection is to locate the edges most likely generated by the scene elements rather than the noise of the image. This is because image noise can cause intensity variations which can result in spurious edges.

A good edge detector should suppress most of the “false” edges without destroying the true edges. It should also enhance the edges; i.e. increase the pixel values of the edges. In addition, the edge detector should be able to differentiate between the edges caused by noise and the actual edges of the image. This can be done by thinning the wide edges: if it is not a maximum value it is suppressed (non-maximum suppression), and by thresholding: a minimum value is established to declare a local maximum at the edge.

The goal of our project is to investigate the different types of edge detectors. Using the information gleaned, we will create a demonstration program that will use our edge detection algorithms. We will perform edge detection on various types of images: images of the real world, computer generated images, images with different types of noise, and images that have been smoothed by filters of different sizes and types. We will then use the extracted edges to enhance the image. Finally we will use our edge detection algorithms to recognize characters of the alphabet.

**2. Theory of Edge Detection**

Edges can be generated from different physical sources. But regardless of how the edge was generated, the resultant edge exhibits varying degrees of discontinuities in image intensity [2]. Physical edge sources may arise from different surface properties such as change in color of an object, reflective properties, and change in texture of the object. Edges may also be formed as a result of discontinuities in the distance and the object’s orientation. Edges may also be caused by shadows.

There are many different types of edges; an edge may be a step function, it may be a ridge, a ramp function, or a roof edge (triangular) function. Additionally the width of the edge can vary infinitely. Identifying an edge amounts to finding the change in intensity of the image. As such we can find the first and second derivative of the image intensity function. The first derivative of the image intensity functions produces a high where an edge exists. The second derivative identifies the zero-crossings of the image intensity function.

2.1 First order Edge Filter

If *I(c,r)* represents the image intensity at pixel *(c,r)*, then the first order image gradient can be approximated by:

(2-1)

where the partial derivatives are numerically approximated,

(2-2)

(2-3)

Where *hx* computes the first order horizontal image derivative, and hy computes the first order vertical image derivative. And the gradient and magnitude can be defined by:

(2-4)

From (2-4) and (2-2)

(2-6)

(2-7)

Where gx and gy are the image gradients in the x and y directions respectively, and \* represents a 2D convolution operation.

(2-8)

where θ represents the steepest slope direction

These formulae allow us to compute the image derivatives by convolution. Each method of edge detection has its characteristic mask. These masks are sometimes called *stencils*.

In our project we have investigated five first order edge detectors. Other types of edge detectors exist, but it is beyond the scope of this project to explore them all. The edge detectors we chose to implement were chosen because of their accuracy, popularity, and ease of implementation. The edge detectors we chose to implement are:

1. The Roberts Cross Edge detector
2. The Prewitt Edge detector
3. The Sobel Edge detector
4. The Scharr Edge detector
5. The Canny Edge detector

2.1.1 The Roberts Cross Edge Detector

The image derivative filters of the Roberts Cross method are:

, (2-6)

Where *hx* computes the first order horizontal image derivative, and hy computes the first order vertical image derivative. If we take the first order derivatives in 45 and 135 degrees, we obtain the Roberts Cross edge operator:

(2-7)

The Roberts Cross method uses the approximation of the gradient to identify an edge. Convolving with the operator in (2-7) computes the sum of the differences between diagonally adjacent pixels and finds the discrete derivative for each pixel. The output of this convolution produces the gradient of the image. This method identifies a change of intensity in the diagonal direction. We can see from (2-7) that the kernel is small and contains only integers so the edges are simple to compute using this method. However, one of the disadvantages of the Roberts Cross method is that is highly sensitive to noise.

2.1.2. The Prewitt Edge Detector

To make our edge detector more robust to noise, we can first preprocess the image to remove some of the noise. This preprocessing is done using a simple smoothing operator. If *s* is the smoothing operator and *h* represents the difference operator, from equations (2-6) and (2-7), we have

(2-8)

which states that the image gradient can more accurately be found by smoothing the image and then applying the image derivative filter. Note that because of the associative property of convolution, equation (2-8) can be rewritten as:

(2-9)

Hence we can see that is the smoothed difference operator. The simplest smoothing operator is the averaging operator and if an image is smoothed with the following impulse response:

(2-10)

and the resulting image is convolved with the first order horizontal derivative hx in equation (2-6), we have the classical horizontal Prewitt operator.

(2-11)

and the vertical Prewitt operator is generated in a similar fashion

(2-12)

The Prewitt edge detector calculates the maximum response of the set of convolution filters (2-9) & (2-10) to find the local edge orientation for each pixel. One kernel is sensitive to edges in the vertical direction and one is sensitive to edges in the horizontal direction.

2.1.3 The Sobel Edge Detector

The Sobel edge detector is relatively inexpensive in terms of computational complexity. This algorithm is based on finding the gradient of the image intensity at each point which gives the direction of the largest possible increase from light to dark and the rate of change in that direction. The result shows how “abruptly” or “smoothly” the image changes at that point. It also shows how likely that part of the image represents an edge and how the edge is likely to be oriented. So the magnitude (likelihood of an edge) calculation is more reliable.

If instead of the simple averaging filter, we choose to smooth the image with a weighted smoothing filter such as:

(2-13)

where the center pixels are weighted more heavily (x2). So if we convolve *s* and *h*, we get:

(2-14)

(2-15)

And these are the kernel for the Sobel operator.

2.1.4 The Scharr Edge Detector

Even though the Sobel operator reduces artifacts associated with a pure central difference operator, it does not have perfect rotational symmetry. The Scharr operator optimizes this property. Scharr operators are formed by an optimization minimizing weighted mean squared angular error in the Fourier domain. The optimization is done using the condition that the resulting filters are numerically consistent. Hence they are derivative kernels as well as symmetrical constraints. The kernels of the Scharr operator are:

(2-16)

(2-17)

2.1.5 The Canny Edge Detector

The Canny edge detector, also known as the optimal edge detector, is probably the most used edge detector. The algorithm is based on an improvement of previously existing edge detectors. Canny Edge detection is performed by doing the following steps:

1. First smooth the image with a Gaussian smoothing filter to remove noise
2. Find the image gradient of each pixel – this highlights the regions with high spatial derivatives
3. Track along these regions and suppress any pixel that is not at a maximum
4. Apply hysteresis – use two thresholds, if the magnitude is below the first threshold, set the current pixel value to zero (make it a non-edge), if the magnitude is above the high threshold, it is determined to be an edge. If there are any magnitudes between the two thresholds, search for a path from this pixel to a pixel with a gradient above the second threshold, if such a path exists, the pixel is defined as an edge, if not, the pixel is defined as a non-edge.

2.2 Second order Edge Filter

The place where the first order image derivative of a step edge is maximum is exactly the place where the second derivative has a zero-crossing. The isotropic generalization of the second derivative in 2D is the Laplacian. The Laplacian of the image intensity *I(c,r)* is:

(2-18)

This Laplacian operation does not provide the strength of the edge point; neither does it give the edge direction. This is typically solved by using the first order image gradient as well. The second order image derivative is more sensitive to noise than the first order image derivative. In our project we explored one second order edge detection algorithm:

2.3 Noise

Noise in imagery greatly affects the ability to detect edges. In terms of the spectrum of the image, noise generally shows itself as additional high frequencies. Unfortunately, since edges are also high frequency, edge detectors encounter ambiguity when attempting to differentiate between edges and noise.

To illustrate this, we will first present some common types of noise found in images, which can be manipulated using the demonstration tool discussed later. We then present three methods of reducing noise in an image, all of which are low pass filters. Note that when attempting to filter out noise, we will also soften the edges of an image, making it more difficult to detect them.

Three types of noise are considered in this project. First, Gaussian white noise is an additive form of noise. A Gaussian random variable is added to each pixel with a certain mean (usually zero) and standard deviation. This value of this random variable is independent of the value which it affects. The second type of noise is salt and pepper noise. This is defined as random pixels being turned on and off with some probability density. The final type of noise considered is speckle noise. Under this multiplicative type of noise, the value of each pixel is multiplied by some small uniform random variable with the defined variance.

In order to cope with noise, three noise reduction, or “smoothing”, algorithms are demonstrated. Under Gaussian smoothing, the image is convolved with a Gaussian kernel of some mean, standard deviation, and size. Another approach, mean filtering, attempts to reduce the effects of noise by replacing the value of a pixel with the average of those around it. This is equivalent to convolving the image with a normalized kernel of ones of a certain size. A third approach, median filtering, replaces the value of a pixel with the median value of those around it.

To evaluate these smoothing functions on noise, the signal to noise ratio is compared for smoothed. The noisy image is Lena, with Gaussian white noise of zero mean and variance 0.01. The SNR is recorded for each filter and kernel size. In the case of Gaussian smoothing, the optimal standard deviation is used.

|  |  |
| --- | --- |
| Filter Type | Signal to Noise Ratio |
| Gaussian 3x3, σ=1.2 | 19.91 |
| Gaussian 5x5, σ=1.1 | 20.47 |
| Gaussian 7x7, σ=1.1 | 20.49 |
| Gaussian 9x9, σ=1.1 | 20.49 |
| Gaussian 9x9, σ=1.1 | 20.49 |
| Median 3x3 | 18.80 |
| Median 5x5 | 19.72 |
| Median 7x7 | 19.28 |
| Median 9x9 | 18.63 |
| Median 11x11 | 18.03 |
| Mean 3x3 | 19.73 |
| Mean 5x5 | 19.41 |
| Mean 7x7 | 18.41 |
| Mean 9x9 | 17.54 |
| Mean 11x11 | 16.81 |

TODO more data, conclusions

2.3.1 Computational Considerations

When implementing these smoothing algorithms, differences in computation time may be of importance. Median and mean filtering require no setup, however, under Gaussian smoothing, the Gaussian kernel must be computed. However, this only needs to be done once for the whole image and could be precomputed. Therefore this is considered an O(1) operation and is not a factor in this discussion.

In each algorithm, a computation must be performed on each pixel to generate its smoothed version. For Gaussian smoothing with kernel size k x k, each pixel in the image will require k2 multiplications. Any arithmetic operations inexpensive. Therefore, Gaussian smoothing can be done in O(nk2) time for an image with n pixels.

Median filtering requires finding the median number in the window of k2 elements. This can be accomplished by sorting the numbers in the window and taking the middle element. Efficient sorting algorithms exist to sort m numbers in O(m log(m)) time. Since m=k2, smoothing the image with median filtering will require O(nk2log(k)) time. This is somewhat less efficient than Gaussian smoothing.

The final method, mean filtering, has the advantage that it is easy to compute. Calculating each pixel requires just k2 additions and a single division. Therefore the entire image can be smoothed in O(nk2). This appears the same as Gaussian filtering, though it was previously assumed multiplication is O(1). The lack of multiplication instructions in the mean filtering method would in fact make it faster.

2.4 Character Recognition

One of the goals of this project was to explore typeset character recognition, and edge detection’s applications to this problem. Due to time constraints and the limited experience of the group, it was decided to develop an algorithm to classify images corresponding to the 26 capital letters of the Latin alphabet. In order to do this, the task was treated as a supervised learning problem. We needed to develop and train a model to classify characters based on a set of features found in such images. To do this, we generated a sample set of images, computed a set of features, trained a neural network, and evaluated its performance.

2.4.1 Sample Data Set Generation

On a high level, the approach of supervised learning is to train a model to detect differences between different classes of data sets. In order to train our model, we needed a set of images of characters from which our model could learn. We wanted our model to be smart enough to not be affected by changes in font style, font size, or stroke thickness. Two steps were taken to address these concerns. First, all images are preprocessed as described below, in order to remove the data’s dependence on font or stroke size. To ensure accuracy across fonts, it was logical for the data set to contain images of multiple fonts. Therefore, our model would learn to see different font styles

To generate said images, a MATLAB script was developed that produced all the capital letters in a variety of font styles and sizes. The data set consisted of images with fonts Arial, Times New Roman, Courier, Helvetica, and Calibri, of sizes 12, 14, 16, 18, 32, 40. Generally, the larger the data set, the more accurate the model will be, so it was desirable for the model to learn on many types of letters.

2.4.1 Preprocessing

To reduce noise in our results, all character images are first run through a preprocessing algorithm. The purpose of this is to maintain uniformity across our data. First, to simplify the process, all images are thresholded to ensure they are binary. Next, all characters should be the same size and have the same stroke thickness. To accomplish the first piece, images are cropped and resized to a 32x32 frame. More complicated is the task of keeping constant stroke thickness. The most straightforward approach was to reduce each character to a “skeleton” of its former self. That is, reduce the stroke thickness of any character to no more than one to two pixels in any one location. After researching various algorithms, we decided to implement the thinning algorithm proposed by Zhang and Suen. The preprocessing algorithm is demonstrated on a thick letter ‘E.’ The first frame represents the original image. The second represents the cropped and resized version. Finally, the skeleton of the image is found.



It should be noted that not all of our features use the skeletonized version. Others used the original image, or the image consisting of the edges.

2.4.2 Identifying Features

In order to differentiate amongst characters, we needed to develop a number of features. Mathematically, a feature is a function that maps an image to a number. Enough features should be defined that any pair of letters can be consistently differentiated. However, an overly complicated model does not generalize well out of sample. Therefore, feature selection must be done cautiously to avoid overfitting.

2.4.3 Feature Set

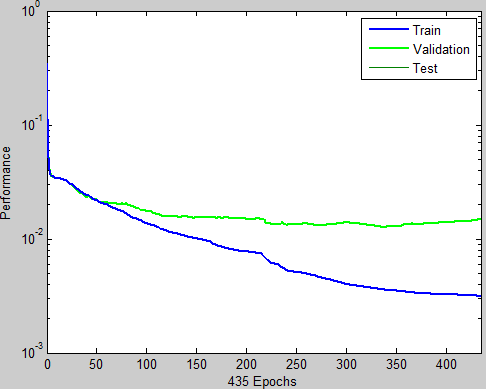
A total of nine features were used to differentiate the capital letters of the Latin alphabet:

* The number of white pixels in the skeleton
* The maximum number of pixels reflected around a vertical axis of symmetry
* The number of “holes” in the image – connected black pixels surrounded by white
* The longest continuous vertical line of white pixels
* The longest continuous horizontal line of white pixels
* The number of vertical lines
* The number of horizontal lines
* Maximum value in cross correlation with template intended to match the “hook” on the letter G. This feature specifically designed to differentiate G from C
* The number of lines in the skeleton, found using the Hough transform

2.4.3 Training the Model

Given that we needed a powerful model to fit a complex set of data, it was decided that a neural network would be an appropriate choice. Neural networks are powerful enough to fit any data set, while regularization can be easily added to improve generalization to out of sample data.

The approach in training a neural network is to generate a differentiable function for the in sample error, and then to use gradient descent to improve the performance. In this case, scaled conjugate gradient was used to help the model “learn,” decreasing the in sample error. To improve generalization, early stopping is used. In this technique, 15% of the data set is left out during training and used as a validation set. After each iteration of gradient descent, the error in the validation set is calculated. When the validation error is determined to have reached its minimum value, training ends. If the model were allowed to train further, it would continue to lower the in sample error, while overfitting. That is, it will place too much weight on noise in the sample data in order to correctly classify that data. Because this noise is not consistent with that found in out of sample data, generalization suffers. This observation is demonstrated below:



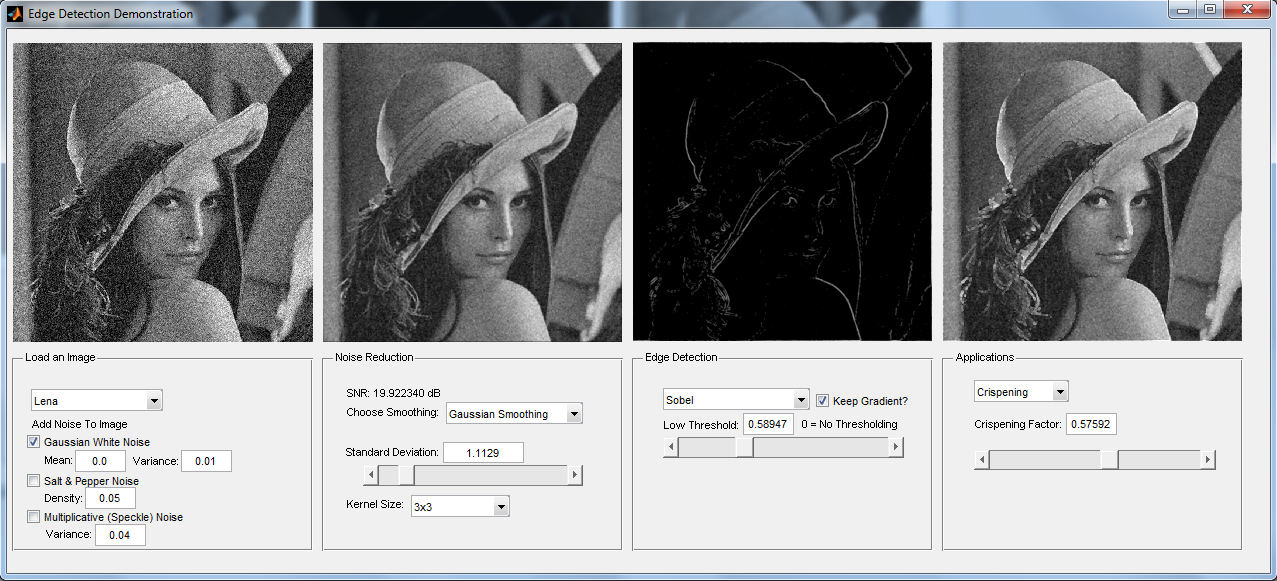
2.4.4 Implementation

The OCR program was implemented as MATLAB script. MATLAB’s Neural Network Toolbox was used to train the model discussed here. TODO continue this

**3. Methodology**

In order to evaluate these different edge detection techniques, as well as to study their applications, we developed a MATLAB demonstration program. The purpose of the application is twofold: to give us a means by which we can test different algorithms and compare them easily and to allow future students to retrace our steps.

The program can be started by running the demo\_gui.m file in the program directory.



The interface is divided into four panels. The first panel allows the user to load an image. Some images are provided; others can be loaded from a file. Next, the three types of noise discussed can be added to the image. The relevant parameters can also be adjusted here. Upon changing these values, the images will automatically be updated accordingly.

In the second panel, the user has the opportunity to explore the three noise reduction algorithms presented in this paper. Gaussian smoothing, median filtering, and mean filtering can be selected with configurable neighborhood sizes and standard deviation (in the case of Gaussian smoothing). Changing these settings will update the image above as well as in later frames. The signal to noise ratio displayed on this frame will also change automatically upon adjusting values in the form.

The third panel allows the user to use the edge algorithms discussed here. The four kernel operators, Roberts, Prewitt, Sobel, and Scharr, can all be used to find the magnitude of the gradient of the image. To determine what constitutes an edge, the user can set a threshold. If the magnitude of the gradient is below this threshold at some point, it is determined to not be an edge and is colored black. For values above the threshold, the gradient value can be maintained or converted to a one, resulting in a binary edge image. In addition, the Canny and Laplacian of Gaussian methods can be explored on this panel. The parameters for these methods can be changed here as well.

The final panel demonstrates crispening. The edge image on the third panel is added to the smoothed image on the second panel to produce a “crispened” image. The amount of the high pass image that is added to the smoothed image is controlled using the slider on this panel.

**4. Results**

4.1 Noise

Of the three types of noise discussed, the additive Gaussian white noise and the multiplicative speckle noise had nearly the same effect on the image. Examining the two images, the differences are not obvious.

The difference becomes clearer after applying Gaussian smoothing and finding the Sobel gradient:

The multiplicative noise, not surprisingly, has a stronger impact on whiter areas. However, the impact on these white areas in edge detection is not much different from Gaussian noise. Setting the threshold to an appropriate level (threshold = 0.8):

The effects of the noise are now diminished greatly. From this it is concluded that for edge detection, the effects of additive and multiplicative noise are not significantly different. The above analysis was performed using Gaussian smoothing. Median and mean filtering yielded the same conclusion, with slightly different images.

More interesting is salt and pepper noise and its effects on an image. Because its effects on a pixel are more obvious and drastic, a smart filtering algorithm can easily detect many bad pixels. First consider Gaussian smoothing and mean filtering on an image with salt and pepper noise:

**

*Left: Lena with S&P Noise (SNR=7.74)*

*Middle: Gaussian smoothed (stdev=1.8, 7x7 kernel) (SNR=17.24)*

*Right: Mean filtered (5x5 neighborhood) (SNR=15.27)*

Both yield almost the same result. The noise is still very obvious and edges have been softened significantly – not a very impressive result. Now, median filtering with a 3x3 neighborhood performed on the same image:

** 

*Left: Lena with S&P Noise (SNR=7.74). Right: Median Filtered Version (SNR=23.95)*

A far more impressive result, it is hard to believe the image on the right was produced from the image on the left. From this it is clear that an image should be tested for salt and pepper noise before continuing with Gaussian smoothing or mean filtering. A perhaps smarter, though more expensive, filter would check if the center pixel was more than some standard deviation multiple from the mean of the neighborhood, and only replace with the median in this case.

**5. Conclusion**