

Feature Detection in Grayscale Aerial Images

by

Katherine Treash

B.A., Mathematics (1997)
Carleton College

Submitted to the Department of Civil and Environmental Engineering
in Partial Fulfillment of the Requirements for the
Degree of Master of Science

at the

Massachusetts Institute of Technology

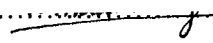
February 1999

©1999 Massachusetts Institute of Technology
All rights reserved

Signature of Author.....

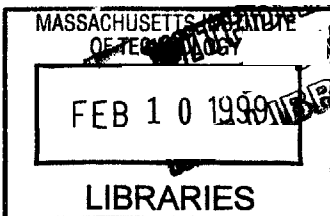
Department of Civil and Environmental Engineering
January 15, 1999

Certified by.....


Kevin Amaratunga
Assistant Professor of Civil and Environmental Engineering

Accepted by.....


Andrew J. Whittle
Chairman, Departmental Committee on Graduate Studies



Feature Detection in Grayscale Aerial Images

by

Katherine Treash

Submitted to the Department of Civil and Environmental Engineering
on January 15, 1999 in Partial Fulfillment of the Requirements for
the Degree of Master of Science

ABSTRACT

Feature detection in aerial images entails a number of specific problems, depending on the feature to be detected as well as the method to be used. This thesis focuses on the problem of automatically detecting roads in grayscale aerial images. The challenges of this particular problem are discussed and two systems are proposed as solutions. Both systems are edge-based methods and have two key steps: an edge detection step followed by an edge linking procedure. One system uses a variant of the Nevatia-Babu edge detector for the first step. This edge detection method revolves around convolutions of the aerial image with a series of masks and is quite simple to implement. The other system applies a wavelet edge detector to the images. Wavelets are briefly introduced and then a set of wavelet filters, developed by Mallat and Zhong, are detailed. In both road detection systems, the edge linking technique is the same. A new edge linking algorithm is developed using a zone-based technique, and is designed to link the long, low-curvature edges which represent roads.

The results of applying each edge detection technique individually, followed by the edge linking procedure, to three test images are displayed. These results are compared and contrasted to try to determine: (i) whether a simple two-step edge-based model has potential for being the initial steps in a full road detection system; (ii) which edge detection method performs better when combined with the edge linking algorithm; and (iii) whether this is a reasonable application for wavelets. All three issues are discussed, as well as possible further steps and improvements to the systems.

Thesis Supervisor: Kevin Amaratunga

Title: Assistant Professor of Civil and Environmental Engineering

ACKNOWLEDGEMENTS

I would like to thank the following people for their help and support:

Professor Kevin Amaratunga for his wisdom and guidance through my entire time at M.I.T. His deep knowledge of wavelets, as well as many other topics, and his ability to teach complex ideas very well made the research and writing of this thesis more fulfilling and enjoyable.

The Suksapattana Foundation, Bangkok, Thailand, for their generous financial support in the form of a research grant to the Intelligent Engineering Systems Laboratory, M.I.T.

My friends at I.E.S.L., Emma, Abel, JP, Hai, Eric, Adriana, and long lost Gilbert, for being an extremely fun source of knowledge and information, and for making the long days in the lab bearable. My friends beyond M.I.T. for keeping me grounded and in tune with the outside world.

My amazing parents, Gordon Treash and Christine Cunningham, and all my family for their support and encouragement. I would never be here without them.

TABLE OF CONTENTS

ABSTRACT	3
ACKNOWLEDGEMENTS	5
TABLE OF CONTENTS	7
LIST OF FIGURES AND TABLES	9

CHAPTER 1.

INTRODUCTION	11
1.1 NARROWING THE FEATURE DETECTION PROBLEM	11
1.2 AUTOMATIC ROAD DETECTION IN GRAYSCALE AERIAL IMAGES	14
1.3 ORGANIZATION OF THE REMAINING CHAPTERS	17

CHAPTER 2.

NEVATIA-BABU METHOD FOR EDGE DETECTION	19
2.1 INTRODUCTION	19
2.2 DETERMINATION OF EDGE ANGLES AND EDGE VALUES	20
2.3 SELECTION OF EDGE POINTS	23
2.4 EDGE THINNING	25
2.5 CONCLUSION	26

CHAPTER 3.

WAVELETS FOR EDGE DETECTION	29
3.1 INTRODUCTION TO WAVELETS	29
3.1.1 Background	29
3.1.2 Wavelet basics	32
3.1.3 Discrete wavelet transform for 2-D signals	39
3.2 RELATION OF WAVELET TRANSFORM TO CANNY EDGE DETECTOR	42
3.3 MALLAT-ZHONG FILTERS FOR EDGE DETECTION	46
3.4 SELECTION OF EDGE POINTS	50
3.4.1 Calculating the modulus image	50
3.4.2 Finding the local maxima	52
3.4.3 Introducing local grayscale variance	54
3.5 CONCLUSION	56

CHAPTER 4.

EDGE LINKING.....	59
4.1 INTRODUCTION.....	59
4.2 OVERVIEW OF EDGE LINKING TECHNIQUE	60
4.3 DETAILS OF EDGE LINKING TECHNIQUE	61
4.3.1 <i>Determining end points</i>	61
4.3.2 <i>Tracing the tree</i>	62
4.3.3 <i>Finding primary and secondary directions</i>	63
4.3.4 <i>Searching for links & edge linking</i>	65
4.4 CONCLUSION	68

CHAPTER 5.

RESULTS AND DISCUSSION.....	71
5.1 INTRODUCTION.....	71
5.2 RESULTS FROM USING NEVATIA-BABU EDGE DETECTOR FOLLOWED BY LINKING	73
5.3 RESULTS USING WAVELET EDGE DETECTOR FOLLOWED BY LINKING	74
5.4 COMPARISON OF RESULTS	78
5.5 CONCLUSIONS	84

CHAPTER 6.

FUTURE WORK AND CONCLUSION.....	87
6.1 POSSIBLE EXTENSIONS TO THE SYSTEM	87
6.1.1 <i>Edge pairing</i>	87
6.1.2 <i>Automatic vs. semi-automatic systems</i>	90
6.1.3 <i>Detection of features other than roads</i>	91
6.2 CONCLUSION	92

BIBLIOGRAPHY	95
---------------------------	-----------

APPENDIX A: MATLAB CODE.....	97
-------------------------------------	-----------

LIST OF FIGURES AND TABLES

Figure 1.1. Examples of test images	15
Figure 2.1. Masks for finding edge values and angles	21
Figure 2.2. Calculating the angle and edge value.....	22
Figure 2.3. Results from running the edge tests.....	24
Figure 2.4. Edge thinning masks.....	26
Figure 2.5. Example of edge thinning	26
Figure 3.1. How certain transforms divide the time-frequency.....	31
Figure 3.2. Haar scaling function.....	34
Figure 3.3. Dilation equation for Haar scaling function.....	35
Figure 3.4. Haar wavelet function.....	35
Figure 3.5. Connection between approximations of a Gaussian curve at neighboring scales, using Haar functions	37
Figure 3.6. Three-level wavelet decomposition using the Haar functions	42
Figure 3.7. Absolute values of h, v, and d coefficients shown in Figure 3.6	43
Figure 3.8. Effect of smoothing and derivative operators on an edge.....	44
Figure 3.9. Mallat-Zhong choice of 1-D wavelet and scaling functions.....	47
Table 3.1. Impulse response filters used for wavelet transform implementation (as shown in [17], Table I).....	49
Figure 3.10. Decomposition of test image using Mallat-Zhong filters	51
Figure 3.11. Three levels of modulus images	52
Figure 3.12. Assignment of the edge direction based on the point angle.....	53
Figure 3.13. Three levels of local maxima images.....	54
Figure 3.14. Example of using grayscale variance for edge detection.....	55
Figure 3.15. Combination of Figure 3.13(c) and Figure 3.14	56
Figure 4.1. Example from [11].....	61
Figure 4.2. Masks for finding end points	62
Figure 4.3. Unit step directions	64
Figure 4.4. Example of finding search area from primary and secondary directions.....	65
Figure 4.5. Outline of edge linking process	67
Figure 4.6. Zones for edge linking	68
Figure 5.1. Original test images	72
Figure 5.2. Test image #1 after Nevatia-Babu edge detection (NBED).....	73

Figure 5.3. Test image #1 after NBED and linking.....	74
Figure 5.4. Test image #1 after wavelet edge detection (WED), five levels of decomposition.....	75
Figure 5.5. Test image #1 after WED and linking, five levels of decomposition	76
Figure 5.6. Before and after linking	77
Figure 5.7. Comparison of edge detection results for test image #1	79
Figure 5.8. Comparison of edge linking results for test image #1	80
Figure 5.9. Comparison of edge detection results for test image #.....	81
Figure 5.10. Comparison of edge linking results for test image #2	82
Figure 5.11. Comparison of edge detection results for test image #.....	83
Figure 5.12. Comparison of edge linking results for test image #3	84
Figure 6.1 Results of the edge pairing algorithm on some test images.....	89

Chapter 1.

INTRODUCTION

Today's world is filled with data of all sorts. One does not have to look far from their everyday routine to see data being generated, collected, manipulated and analyzed. Much of this data is grouped into coherent sets, either in the process of being generated, such as with a photograph or an audio signal, or by those who manipulate and analyze it. These groupings often make clear certain characteristics of the data which aid in the understanding of its meaning. For instance, the elements of a digital image are arranged spatially so that human eyes can detect physical objects in the photo. However, the organizational format may also hide other features of the data which are necessary for answering specific questions or solving certain problems. Feature detection is the process by which these useful but hidden characteristics of the data set are extracted. This is an admittedly broad definition, necessarily so because feature detection processes take so many different forms. The general idea of feature detection is narrowed somewhat to the specific problem addressed in this work by considering two major factors: the type of data being analyzed, and the tools being used to perform the feature detection. Then, by deciding on the feature to be detected and the precise method for doing the detection, the exact problem statement is reached.

1.1 Narrowing the feature detection problem

Although there are a very large number of kinds of data, they can be split into three main categories: one-dimensional, two-dimensional, and multi-dimensional (meaning more than 2-D).

An example of a set of 1-D data is a sonar signal. Multi-dimensional data could be points in a three-dimensional model. Outside the physical world, data also exists in dimensions greater than 3-D which may be used in feature detection procedures. The focus, however, of much of the research being done presently is two-dimensional data. This data usually takes the form of images, and indeed it is two-dimensional data that is analyzed using the procedure developed in this paper.

Within the category of two-dimensional data, there are several subcategories of image types which can be analyzed using feature detection. Facial images are one type which has received a lot of attention lately, with the popularity of face detection research for artificial intelligence applications. With the recent move to digitize fingerprint images, it has become necessary to develop methods for extracting the main features in a fingerprint for identification purposes. Images containing characters, written or typed, are now often processed by computers which attempt to “read” the letters by using feature detection. Medical diagnoses are being made with the help of anomaly detection in medical images such as mammograms and x-rays. Topographic features such as rainfall amounts or population densities can be analyzed for spatial patterns using feature detection methods. Overhead images of the world, such as satellite and aerial images, contain useful information about man-made and natural structures, as well as oceanic and atmospheric features.

The test images used in this work are taken from this last category. Although both satellite and aerial images give a view of the world from an overhead position, these two types of images differ with respect to certain image properties. This makes the feature detection process quite different depending on the values of these properties. The three most significantly varying properties are resolution, color and noise content. With respect to each of these, the majority of satellite images lie opposite the majority of aerial images. Therefore, the overhead image category can roughly be separated into two subcategories, aerial images and satellite images, based on these three image characteristics.

Resolution is a property that can greatly affect feature detection in aerial and satellite images. With the exception of newer satellites which can photograph the earth at higher resolutions, most satellite images are low resolution. This means that the details of larger features cannot be clearly seen, and some smaller features are not visible at all. For instance, in some satellite images roads appear as a single line, while small buildings are remain unseen. On the other hand, aerial images are taken from airplanes which can be quite close to the ground and are consequently often very high resolution images. It is usually possible to see both edges of a road as separate lines in an aerial images, and buildings are visible as more than just single points. If

the resolution is particularly high, the individual lanes and cars can be seen on the roads, and the buildings have definite shapes and textured roofs. Therefore, not only does the resolution of the image affect what kinds of features can be extracted from an image, but it also dictates to some degree what techniques will work best for feature detection. Methods have been developed for both low and high resolution overhead images, but for the purposes of this work, the test images are chosen to be higher resolution aerial images.

Two other significant differences between aerial and satellite images which effect feature detection are the presence of color and noise. Not all satellite images contain color information, and not all aerial images are grayscale, but often the satellites capture the multispectral data while the photographs taken from airplanes do not. Color may provide more information about the data, but its inclusion also adds complexity to any feature detection method which is to be used. In limiting the test images to grayscale aerial images, there is no need to take into account the color characteristics of an image, and consequently the method can remain simpler. Similarly, noise in an image makes the procedures more complicated by the necessity of eliminating or working around this noise. In general, satellite images contain more noise than aerial images. In particular, the aerial images used as test images here contain very little noise.

So, from all the possible types of data on which to perform feature detection, high resolution grayscale aerial images with low noise content are chosen for testing the system developed in this paper. This narrows the feature detection problem somewhat. However, it remains to be stated which specific techniques will be used on this data. For any type of data, there are many choices. Examples of methods used recently for a wide variety of feature detection procedures include mathematical morphology, template matching, neural networks and various other techniques based on probability and Bayes theory, the Hough transform and other forms of parametric modeling, and data fusion. Generally, these and the many other feature detection methods can be labeled as either a local or a global procedure. The method explored in this work is a local edge-based extraction method. The steps of the process will be introduced later in this chapter, and described in detail throughout the thesis.

It should be noted that the purpose of any feature detection tool is usually not to do something that humans cannot do manually. In many cases, such as face detection in images, humans can perform the feature detection better than a computer can. However, either the task takes very long for a person to do manually, or the results need to be recorded more accurately, or the human merely wants an aid in the process, which results in these processes being developed for computers to perform. Such is the case with feature detection in aerial images. With more and more of the world's information becoming digitized and with the popularity of geographical

information systems, the detection and recording of digital map data from aerial images has become increasingly important. In the past, analog information was recorded manually, requiring large amounts of time and effort to catalog the precise details. The human eye can pick out the physical features with ease, but recording the exact position of these features may be slow and less accurate. As well as speeding up this process, having a computer perform the task allows for rapid updating, displaying and transmission of the information. These are the main reasons why systems such as the one described in this paper are useful and necessary.

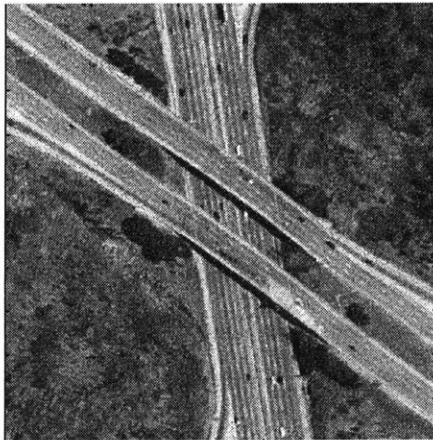
1.2 Automatic road detection in grayscale aerial images

The exact problem addressed in this project is the automatic detection of roads in grayscale aerial images. Roads are chosen as the feature to extract, as opposed to rivers, buildings, railroads, etc., because of their importance in most maps and because there is less variability in the characteristics of roads as compared to the characteristics of other features. Furthermore, an attempt is made to completely automate the process. In general, roads have several characteristics which help to automatically detect them: they tend to be long and continuous, with small curvature, sharp, parallel boundaries, and relatively constant grayscale and texture values between the boundaries. However, the problem is made difficult by recognizing that these characteristics do not always hold for individual roads: boundaries can be blurred, the background can be very close in grayscale level to the road itself, there can be trees or shadows breaking the continuity of the boundaries, or the road can be a short dead-end road or a driveway.

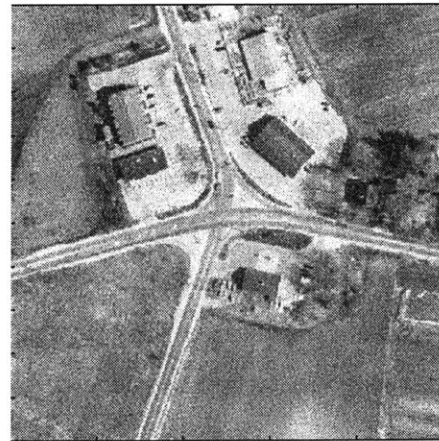
Figure 1.1 illustrates some examples of the variations found within aerial images.¹ Figure 1.1(a) shows a close-to-ideal image for road detection. The roads are wide, their boundaries are clear and continuous, the background is significantly darker than the roads. The only problem that this image demonstrates is the problem of cars on the roads, which break up the relatively smooth texture of the highway and disturb the edge detection by creating short ‘noisy’ edges. Figure 1.1(b) also has continuous roads with few occlusions, however here the grayscale level of the surrounding areas is closer to that of the roads. The road boundaries are sometimes broken by the presence of driveways. Figure 1.1(c) illustrates the difficulty of viewing roads through tree tops and their shadows. The edges of these roads are highly occluded, even making

¹ These images and all test images for this paper are taken from a digital orthophoto browser created by the Planning Support Systems Group in the M.I.T. Department of Urban Studies and Planning in conjunction with Massachusetts Geographic Information Systems (MassGIS) and are found at <http://ortho.mit.edu>.

it hard for a human observer to make out the exact position of the edges. Figure 1.1(d) shows a more densely populated area in which the building edges are often more sharp than those of the roads. Shadows, this time from houses, also complicate the analysis of such an image. Furthermore, although all four of these images are approximately the same resolution, the road widths and lengths vary.



(a)



(b)



(c)



(d)

Figure 1.1. Examples of test images

The disparity in characteristics from image to image is the primary source of difficulty in the road detection process. Many systems have been developed using a variety of methods for automatic road detection, and each tries to handle the varying image characteristics in different ways. As with feature detection procedures in general, these systems can be classified as being either local or global in nature. Global procedures attempt to develop accurate models for the

roads, incorporating the many characteristics into as few parameters as possible.[2] Within the category of local methods, many techniques exist. Edge detectors and linkers are often used to find road seeds, which are then grown into the full road network.[23][25] The textural and spectral properties of roads are employed to group together road pixels. Similarly, the difference between the grayscale levels of roads and the background is exploited in order to determine the boundaries of the roads. In addition, methods which attempt cooperation between local and global methods also exist.[1] (Many of these references give further lists of road detection methods.) What is consistent throughout most of these methods is that they become quite complicated in their attempt to deal with all the variations in images. It is the overall goal of this paper to develop a simple method for road detection and see how well it performs. The system focuses on road finding, which is often followed by road tracing in most other road detection procedures. Few assumptions are made, and they are that roads are usually long and are either straight or changing direction very slowly. It is anticipated that these assumptions are enough to allow the procedure to find as many of the road edges as possible, while ignoring the non-road edges.

It is clear then that the system to be developed will be a local, edge-based method. It has two main steps: edge detection, followed by edge linking. Within these steps are two, more basic goals of the system. One is to experiment with wavelets for edge detection. Wavelets and the wavelet transform are a relatively new tool for image analysis. The results from using wavelets for the edge detection step are compared to results from using a more well-established linear edge detection method. The second goal is to develop a new edge linking procedure that is designed to specifically link road edges. Two previous efforts describing local methods for road detection and one paper detailing the use of wavelets for edge detection provide models for the three separate steps as well as the system as a whole.

- [18] The linear edge extraction method developed by Nevatia and Babu uses a set of masks to associate an angle and edge value with each image point. Then, through a series of local tests, they determine the edges. After filling in some of the edge gaps with a linking procedure, they approximate the edges with linear segments. They use the fact that a road should have two parallel edges for its boundaries to detect the roads. Our system follows these same basic steps: edge detection, and edge linking. The edge detection procedure of Nevatia and Babu works quite well and provides useful information for later steps. Subsequently, it is used directly as the first edge detector in

our system, with small changes. Because of its proven success, it is used as a comparison with the wavelet edge detector.

- [17] In this paper by Mallat and Zhong, the wavelet transform is applied to the edge detection problem. Although much of the work focuses on determining edge characteristics based on the output of the wavelet transform and rebuilding signals from edge information, they also develop a fast implementation of the transform which is used as the second edge detector in our system. The basic theory of wavelets as well as Mallat and Zhong's implementation are described later in this thesis.
- [11] A semi-automatic road detection system developed by Heipke, Englisch, Speer, Stier and Kutka also follows the same pattern of steps as Nevatia-Babu. However, they use a Sobel filter for edge detection. Before linking, they include a step for thinning the edges which is used in our system. Their linking scheme is based on searches in small areas, predefined by the direction of an edge. It only deals with small gaps in the edges. Our linking procedure comes from expanding the idea of searches based on edge directions to enable it to fill both large and small gaps and to perform well on extensive edge networks.

Although not all elements of the steps in our method are new, their combination is. The system which results from combining ideas from the three above works leads to a new road detection procedure for use on high resolution aerial images.

1.3 Organization of the remaining chapters

Chapters 2 and 3 are dedicated to the detailed explanation of the first main system step, edge detection. Chapter 2 describes the edge detector developed by Nevatia and Babu, which itself consists of three steps. Chapter 3 ventures into the world of wavelets. It begins by introducing the basics of wavelet theory before detailing the wavelet transform implementation of Mallat and Zhong. It finishes by explaining how to use the output from the wavelet transform to extract the image edge points.

Chapter 4 continues on to the second step of the system, edge linking. It is here that the new edge linking procedure for road linking is specified. The basic idea behind the method is to

continue a broken road edge in the direction it is heading. Although this idea is simple, there are many details involved in the actual algorithm. Because this is a new method, all these details are explained.

Chapter 5 illustrates the results of applying the edge detection and linking steps to several test aerial images. Results are shown after each of the two stages and results from the two edge detectors are compared. Conclusions are drawn as to how well the goals of the project are met and the overall success of the method.

Chapter 6 concludes the work by first discussing some possible extensions and improvements to the system. One extension, the edge pairing step, has been implemented and an example is shown of its results. Also addressed in this chapter are the possibility of extending the system to detect features other than roads, as well as the improvements which are gained from making the system semi-automatic instead of automatic. This chapter also contains the concluding summary remarks. Specifically, it addresses what has been learned about the nature of the road detection problem.

Chapter 2.

NEVATIA-BABU METHOD FOR EDGE DETECTION

2.1 Introduction

Assuming that edges represent sharp changes in grayscale level, and that roads have a significantly different grayscale level than the surrounding background, edge detection is a natural first step in extracting roads from aerial images. The problem of detecting edges in images is one that has been extensively studied. Several well-established methods continue to be amongst the most used edge detection techniques. The simplest of these consist of a single filter which takes the derivative of an image. Filters based on taking the first derivative detect edges by looking for local extrema, while those based on the second derivative associate edges with zero crossings. Multiscale methods, such as the Canny edge detector (based on the first derivative) and the Marr-Hildreth edge detector (based on the second derivative) are very useful for detecting edges at various scales. Other more complex methods which are built on these simple techniques also exist, and much research continues to be done on edge detection techniques for images with many new detectors being developed for specific applications.

As was mentioned in the first chapter, two methods are plugged into the edge detection module of this system and are compared to see which performs better for road detection. The first method, which is the topic of this chapter, is a simple but effective local edge detection method. It was developed by Ramakant Nevatia and Ramesh Babu for use in their linear feature extraction system. They have good success with extracting runways from an aerial image of an airport using their technique. Roads characteristics are very similar to those of runways, and tend

to be linear or close to linear. It therefore seems to be an appropriate technique to use as the first step in a road detection system. Further confidence in the technique is achieved by recognizing the several other researchers who have noted the usefulness of applying the Nevatia-Babu edge detection method to the problem of road detection, including [23] and [25].

There are two main steps to this method. The first step involves determining an edge value and an edge angle for each image pixel. A series of masks are convolved with the image in order to determine how strongly a pixel is part of an edge and in what direction that edge is pointing. The second step consists of choosing the edge points based on the pixel characteristics established in the first step. The specifics of these two steps are detailed in the next two sections. A third step is added for our implementation. A thinning procedure, developed by Heipke, Englisch, Speer, Stier and Kutka in [11] is applied to the edges to whittle them down to a one-pixel width. This procedure is described in Section 2.4.

2.2 Determination of edge angles and edge values

As was mentioned above, the first step in the Nevatia-Babu method involves labeling each pixel of an image with an edge value (magnitude) and an edge direction (angle). The edge value is meant to give an indication of how strongly a point is part of an edge. The direction denotes the angle of the edge the point is most likely a part of. In order to find these two parameters for each pixel, the neighborhood of each point in the image is compared to several masks, which represent ideal edges at various orientations. Each mask is basically a matrix filled with the values corresponding to the digital values of an ideal edge oriented in a certain direction. The neighborhood is chosen to be the same size as the masks with the target image point at its center. The goal of each comparison between a neighborhood and a mask is to determine how strongly the image points in the neighborhood correspond to the angle represented by the mask. The angle of the mask which corresponds the strongest to a given neighborhood becomes the angle associated with the center pixel of the neighborhood. The value outputted from the comparison of that mask and the neighborhood is the edge value associated with the center pixel. This process is illustrated using the actual masks from our system at the end of this section.

First, the masks must be determined. Two factors must be considered when choosing the ideal edge masks: how many masks to use and how large they should be. The number of masks corresponds to the number of orientations of the ideal edge that will be compared to image points. Because there is no front or back to the edges, only angles from 0 to 180 need be considered, with

0 and 180 being the same orientation. If the program were to use four angle values, 0, 45, 90, and 135, four masks would be needed. In our case (as with Nevatia-Babu), six masks were found to work well, with angle values 0, 30, 60, 90, 120 and 150. The size of the masks is a detail considered closely by Nevatia and Babu. As they mention, small masks may have problems discriminating between noise and true edges, while large masks may miss the finer details of the edges. In concurrence with their findings, a set of 5x5 masks worked well for the test images used in the present work, especially considering the high resolution and low noise content of these images. The masks are shown in Figure 2.1.

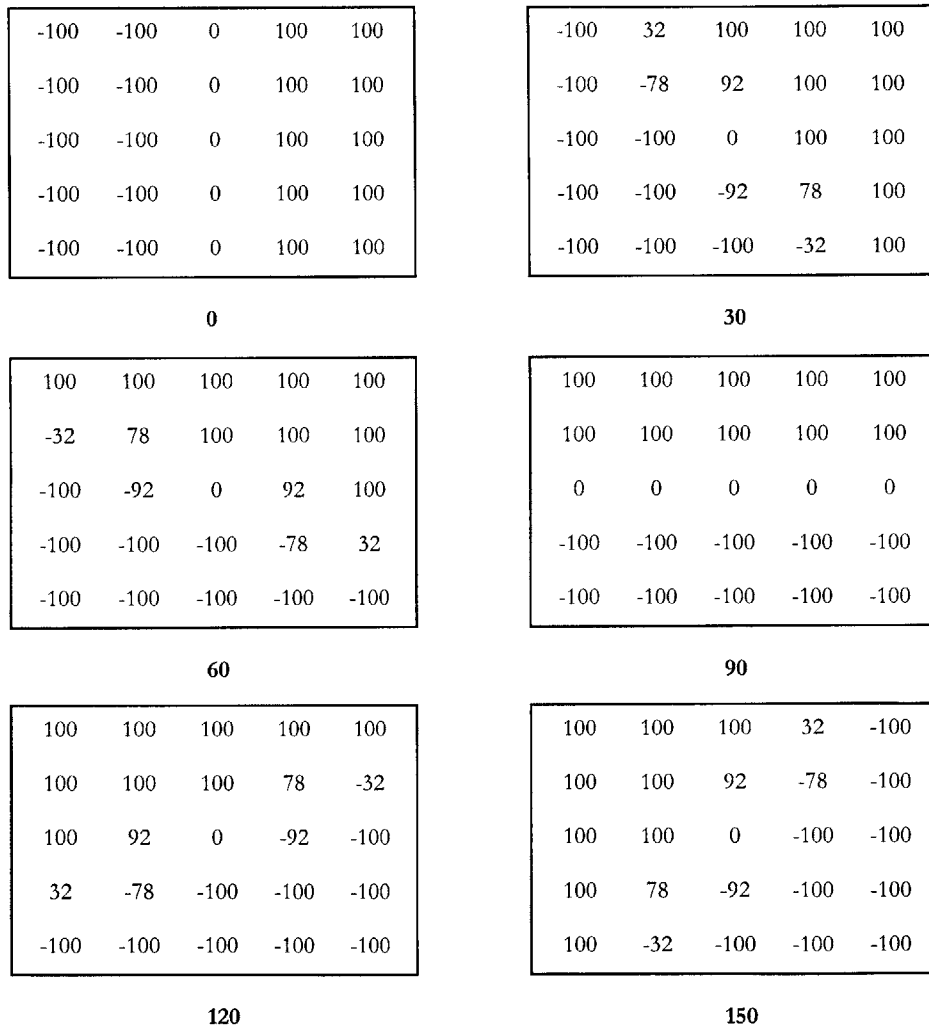
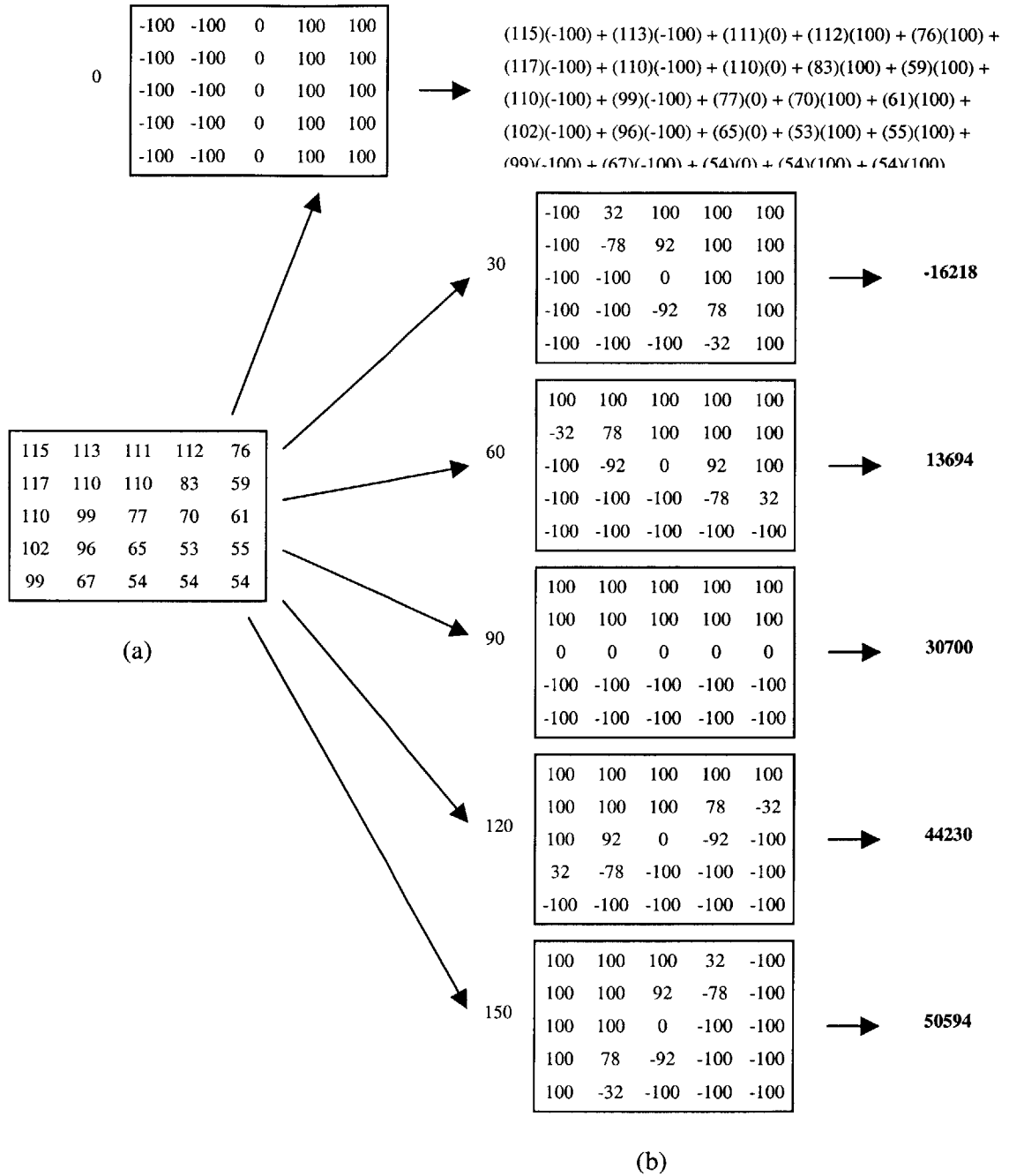


Figure 2.1. Masks for finding edge values and angles



edge value = $\max\{\text{abs}(-35100), \text{abs}(-16218), \text{abs}(13694), \text{abs}(30700), \text{abs}(44230), \text{abs}(50594)\}$

edge angle = **150**

(c)

Figure 2.2. Calculating the angle and edge value

Using these six masks, the edge value and angle are determined for a given image point in the following way:

- (i) The neighborhood is determined with the image point at its center.
- (ii) For each mask, the products of corresponding points in the neighborhood and the mask are summed.
- (iii) The largest in absolute value of the six sums becomes the edge value.
- (iv) The angle of the mask which produces this largest sum becomes the edge angle.

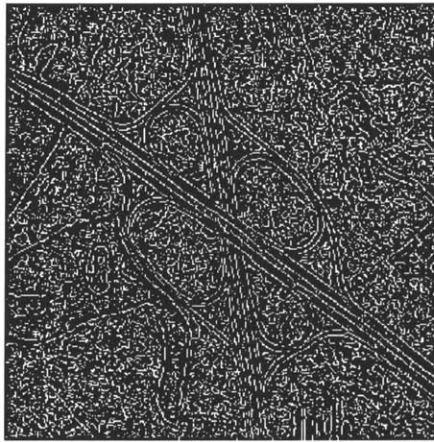
Figure 2.2 on the previous page shows this process for an example neighborhood.

2.3 Selection of edge points

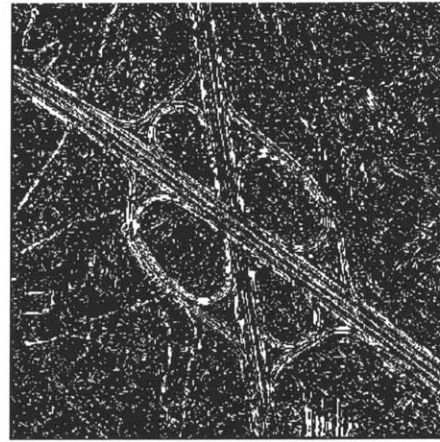
The direction and edge value information is used to determine which image pixels are truly part of an edge. The points with the highest edge value are the strongest edge point candidates. The angles help determine the points lying along a certain edge. Nevatia-Babu use three criteria involving the edge value and direction parameters to test for an edge pixel: (i) a local maximum test; (ii) an angle test; and (iii) a thresholding test. Each test is applied to each image pixel in turn. Test (i) checks that the pixel is a local maximum. It uses the edge angle at this point to find the two immediate neighboring pixels in the direction *normal* to that of the edge. It then checks that the center point has a larger edge value than both these neighbors. The check can be extended to include any number of neighbors along this normal direction. Including more pixels in the check implies that the center pixel is a local maximum over a larger area. For the purposes of this system, the center point is a local maximum if it is greater than its *two* neighbors on either side in the direction normal to the edge. Test (ii) checks that the angle value of the center point is close to that of the neighboring points *along* the edge angle. To pass this test, the difference between the angle of the center pixel and its two immediate neighbors in the edge direction must be below some threshold. Test (iii) is a simple thresholding of edge values. If the edge value at a point is larger than some preset threshold, it passes this third test. The preset threshold is determined by the user. The results of running these three tests individually and in combinations are shown below.



(a)



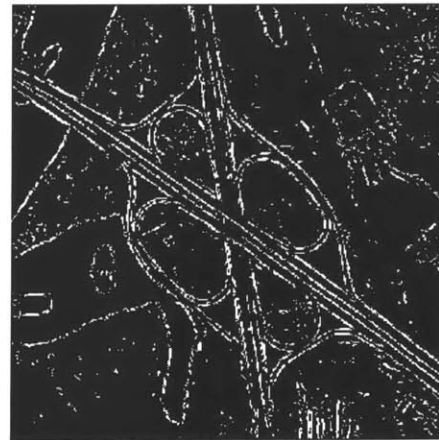
(b)



(c)



(d)



(e)

Figure 2.3. Results from running the edge tests: (a) original test image; (b) edges chosen by test 1; (c) edges chosen by test 2; (d) edges chosen by test 3; (e) edges chosen by combining all three tests.

Nevatia and Babu combine all three tests to find edge points. The result of choosing edge points based on tests 1, 2, and 3 is shown in Figure 2.3 (e). (If a point is chosen to be an edge point, its value is set to 1; otherwise the value is 0.) This is not a bad edge image for our purposes. Many of the road edges are prominent and unbroken, and they are not too thick. Also, there are several small gaps in these edges, but the larger, more troublesome gaps (in later stages) are fairly sparse. This combination of edge tests results in a better edge image than any of the other combinations of tests (all of which use only two out of three tests). However, the edges resulting from using only the third test, the threshold test, are also quite good, perhaps even better than using all three tests. As seen in Figure 2.3 (d), the road edges are quite complete and more of the smaller gaps are filled. This image has an increase in continuous, unbroken edges and is also a good candidate for further road detection processing. Because it requires less calculations, the edge image resulting from using only test 3 is used in the final system.

2.4 Edge thinning

One further step is necessary for implementation of the Nevatia-Babu edge detector for use in our road detection system. Once the edge points are selected, any single points are deleted from the edge image in order to eliminate some of the noise. The remaining edges vary in thickness. Before proceeding to the linking stage of the system, it is helpful to thin the edges to a one pixel thickness using a borrowed method from [11]. The goal is to remove edge pixels one by one until only the “skeleton” remains. A pixel in the skeleton can have only two other edge pixels in the 3x3 neighborhood surrounding it, unless it is the end of an edge or at a crossing point. The 3x3 neighborhood of each edge pixel is inspected in order to determine if the center pixel is part of the skeleton or not. There are many possible patterns which indicate that the center pixel is not part of the skeleton and can thus be removed. However, by taking into account certain conditions on properly thinning edges, the number of patterns which need to be considered is reduced dramatically. These conditions guarantee that (i) thick edges are not split into parallel thin edges, but are thinned as a single edge; (ii) that no holes are created in edges, and; (iii) edges are not shortened at all, but remain the same length. The additional consideration of symmetry determines seventeen 3x3 masks, shown in Figure 2.4, which need to be applied to each edge pixel in order to thin the edges. One pass of the thinning algorithm compares the neighborhoods of each edge pixel with the original, transpose and rotations of each of the seventeen masks. Also, they must be compared starting from the left, right, top and bottom in

order to take full advantage of symmetry. Figure 2.5 shows an example of finding an edge pixel which does not belong in the skeleton. In dealing with the images used in this work, one pass through all the checks was enough to satisfactorily thin the edges. However, if edges are thick enough, the thinning may have to be repeated several times.

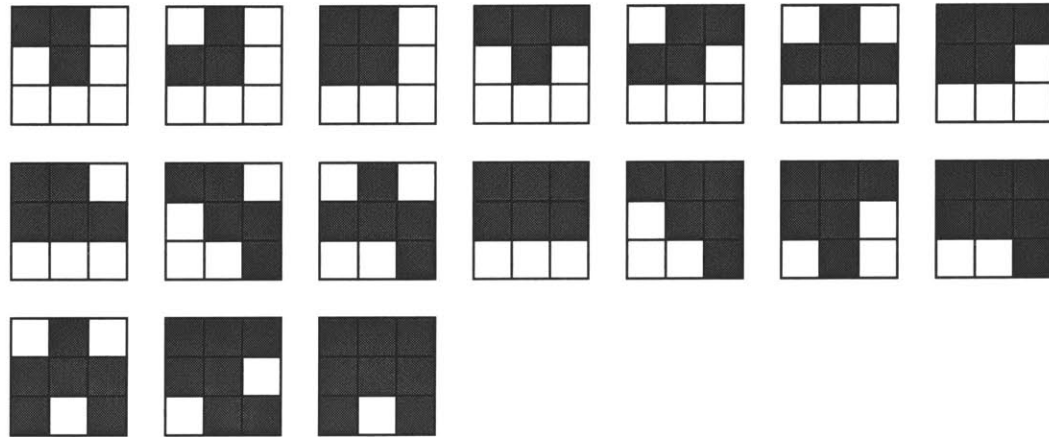


Figure 2.4. Edge thinning masks

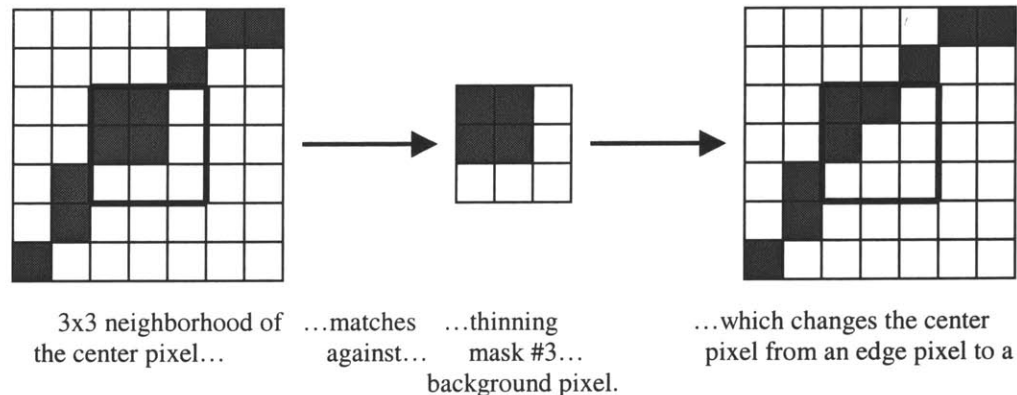


Figure 2.5. Example of edge thinning

2.5 Conclusion

The Nevatia-Babu edge detector is chosen as the first method for extracting road edges from the test images. Not only does it have a good track record with other road detection systems, but from the discussion above it can be seen to have several advantages.

- It is a simple procedure, to understand and to implement. Determining the edge values and angles for each pixel requires the equivalent of a convolution of the image with each of the masks shown in Figure 2.1, then a simple search for the maximum resulting value at each point. Choosing the edge points requires performing one basic test for each pixel. The test does not involve a lot of computation, concentrating on a very small neighborhood around the pixel in question, so this step is also quick and easy to implement.
- The second advantage revolves around the fact that this method was devised to extract *linear* features, and roads are often linear. The masks which represent the ideal edges at various orientations model linear edges, not curved edges. Therefore, the largest edge values will correspond to pixels that are a part of linear edge pieces. When thresholding occurs, as in test 3 described above, these largest values will be the ones to pass the test and be labeled as edge points. Therefore, this method should favor linear edges over edges with a large curvature, which should in turn help to identify the road edges more easily.

For these two reasons, the Nevatia-Babu edge detector is used as one option for the first step in our road detection system.

Chapter 3.

WAVELETS FOR EDGE DETECTION

3.1 Introduction to wavelets

Wavelets are a complex mathematical tool that has found applications in a wide variety of fields, including signal and image processing. This section is by no means a complete guide to wavelets or their many uses. It is a very brief introduction to where they came from, what they are, and how they are particularly useful for edge detection in the context of a road extraction system. Section 3.1.1 describes the similarities and differences between the wavelet transform and the Fourier transform. Section 3.1.2 defines wavelets and outlines the idea of multiresolution, which is one of the underlying properties of the wavelet transform. Section 3.1.3 discusses how the multiresolutional approach works for decomposing images and what advantages wavelets have for edge detection in images. The material is written for those who have no previous experience with wavelets or the wavelet transform. More comprehensive introductions to wavelets are asterisked in the bibliography.

3.1.1 Background

The theory of wavelets and the wavelet transform has become well-defined only within the last fifteen years. The underlying idea was used prior to this, under a variety of different names, in a variety of different fields from harmonic analysis, to signal processing, to computer vision. Due to the lack of communication between the fields using the wavelets idea, there was no realization that the methods being used were related until the mid-eighties. It was at this time

that several individuals working on the theory of wavelets joined forces and united these seemingly separate ideas under the heading of “wavelets”.

The idea of wavelets arose out of the need for a better tool for analyzing signals. Transforms are often used for this task. In general, transforms take a signal and represent it in a different form, usually because the new form makes explicit some characteristic of the signal that is not obvious from the original representation. Probably the most well-known and most often-used transform is the Fourier transform. The Fourier transform, abbreviated FT, takes a one-dimensional signal that depends on time or a two-dimensional signal that depends on space and transforms it into a signal that depends on frequency. It does this by breaking the signal into sine and cosine components. The frequency content of the signal becomes obvious after a signal is transformed by the FT. However, the time or space information which was readily apparent in the original signal is now hidden deep in the numbers and is not obvious at all. The signal must be transformed back using the Inverse FT in order to be able to see the time or space information again. Figure 3.1 (based on [22], p. 8) (a) and (b) illustrate the trade-off between time and frequency information for the 1-D case, where the horizontal axis is time and the vertical axis is frequency. Any point in a signal must lie in the time-frequency plane. These diagrams show how precise the time and frequency information is at all points in the plane. (a) represents the original signal, where time is completely localized (imagine the vertical strips are infinitesimally thin), but at any time, it is impossible to separate out the frequency information. (b) shows the time-frequency plane of the FT. Here, frequency is completely localized, but for a given frequency there is no time information.

By only transforming a small piece of the signal at a time, some time or space information can be kept. This technique is the windowed FT, and it divides the time-frequency plane as shown in Figure 3.1 (c). By using a fixed-size window, there is some time localization *and* some frequency localization for each point. Despite this advantage, there are trade-offs between how much time/space and frequency information can be had at the same time. A small window gives good time localization, but the lower frequency information will be lost because the widest sine and cosine curves (which represent the low frequencies) will not fit inside the window. Making the window larger incorporates more of the lower frequencies but also makes the time information less precise.

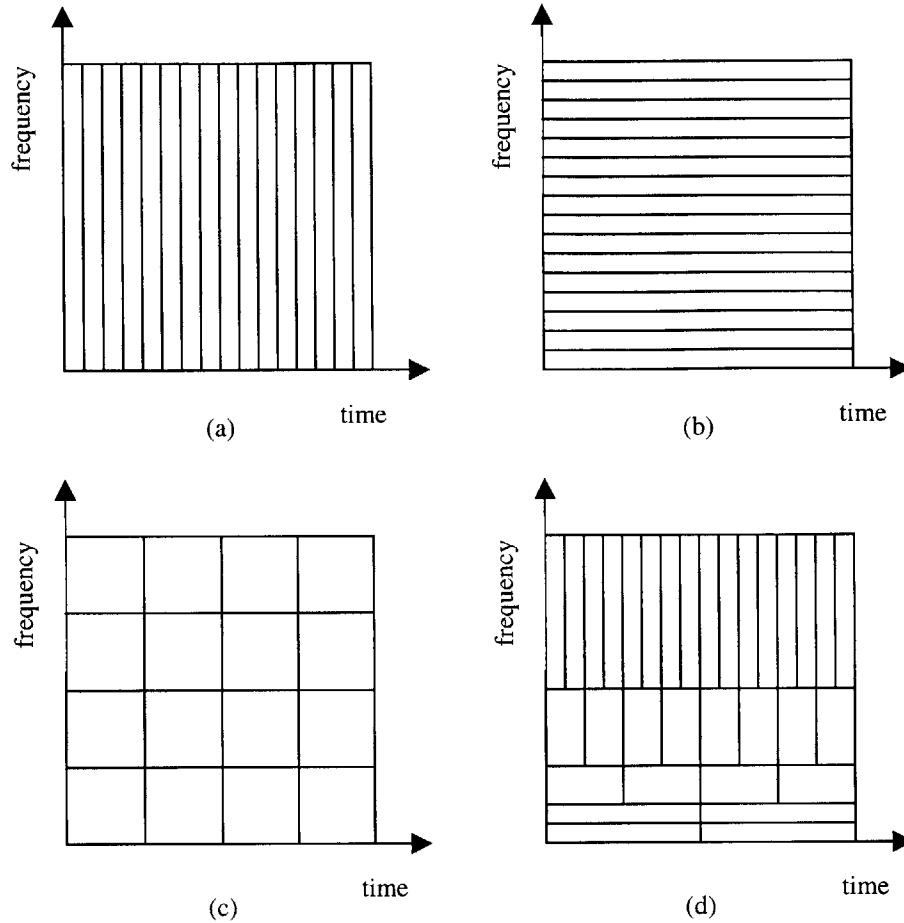


Figure 3.1. How certain transforms divide the time-frequency plane: (a) original signal; (b) FT; (c) windowed FT; (d) wavelet transform

Thus it can be seen that the major problem with the windowed FT is that the window size is fixed, no matter what frequency is being examined. Ideally, the window size would change so that it would be wider at lower frequencies and thinner at higher frequencies. This would allow all frequencies to fit inside their window, but it would still give the best possible time-localization for each frequency. The wavelet transform splits the time-frequency plane in just this manner, as shown in Figure 3.1 (d). The lower the frequency, the wider a time-window it provides. It was this property of the wavelet transform that initially made it attractive to so many people who were studying signals. However, there is much more to wavelets and the wavelet transform than time-frequency localization, as the next section will show.

3.1.2 Wavelet basics

To understand the wavelet transform at its simplest level requires only a small alteration in the FT. Instead of decomposing a signal into multiples of sines and cosines, the wavelet transform decomposes the signal into multiples of wavelets. Here the wavelets are a family of functions consisting of all dilations and translations of a parent wavelet, $\psi(t)$ ²:

$$\psi_{j,k}(t) = 2^{\frac{j}{2}} \psi(2^j t - k) \quad (3.1)$$

The parent wavelet $\psi(t)$ must be a function that integrates to zero. The $2^{j/2}$ constant is for normalization purposes. As j increases, the wavelets get taller and thinner. When the wavelet transform is applied to a signal, the signal is broken up into the components corresponding to each of these varying-height and width wavelet functions, just as the signal is broken into sine and cosine components with the FT.

If j and k are allowed to vary continuously, then there will be a very large number of these wavelet functions, of all heights and widths. In this case, the decomposition will contain redundant information and will take very long to complete. The transform with continuous j and k is called the *continuous wavelet transform* (CWT). Although this is suitable for some applications, most need a faster implementation and do not want the redundant information. By limiting j and k to being integers, the *discrete wavelet transform* (DWT) is defined. Typically, the DWT employs dilations of the parent wavelet function by a factor of a multiple of 2 and translations by integer shifts. Using the DWT, a signal $f(t)$ in L^2 can be expressed as the sum of the weighted wavelet function,

$$f(t) = \sum_{j,k} b_j[k] \psi_{j,k}(t) \quad j, k \text{ are integers} \quad (3.2)$$

where the b coefficients are called the wavelet coefficients. These coefficients can be found quickly and easily if the wavelets are orthogonal. That is, if all the wavelet functions $\psi_{j,k}$ are orthogonal to each other (all dilations and translations of the parent wavelet with j, k integers are

² Although wavelets are being introduced in order to use them in the analysis of a two-dimensional system, most of the following discussion of wavelet basics focuses on the one-dimensional case. This is for simplicity and because one-dimensional wavelets are also used in our system.

orthogonal to one another), then the wavelet coefficients can be calculated by a single integral involving the signal and the wavelet functions:

$$b_j[k] = \int_{-\infty}^{\infty} f(t) \psi_{j,k}(t) dt \quad (3.3)$$

The uniqueness of the wavelet transform comes from introducing another family of functions, called the scaling functions. These functions are closely related to the wavelet functions and are defined in much the same way, based on a parent scaling function $\phi(t)$, which must integrate to one:

$$\phi_{j,k}(t) = 2^{\frac{j}{2}} \phi(2^j t - k) \quad (3.4)$$

Instead of expanding the signal with respect to all dilations and translations of the scaling function, this second set of functions is primarily used in the decomposition to approximate the signal at a certain scale. The variable j controls the dilation of the functions and also indicates scale level, allowing for the following types of signal approximations:

$$f_j(t) = \sum_k a_j[k] \phi_{j,k}(t) \quad (3.5)$$

The a_j coefficients are the scaling coefficients at scale j .

If the wavelet and scaling functions together form an orthogonal system, the DWT has especially nice and useful properties. Three things must be true of the functions in order to classify the system as being orthogonal: (1) All wavelets $\psi_{j,k}$ are orthogonal to each other (all dilations and translations of the parent wavelet with j, k integers are orthogonal to one another), (2) the parent scaling function ϕ is orthogonal to all its integer translations, and (3) the parent wavelet is orthogonal to all the integer translations of the parent scaling function. When these three conditions are met, the system is said to be orthogonal. Note that in an orthogonal system, the wavelet and scaling functions are intimately related. The two must be chosen together to satisfy these conditions.

Orthogonal systems of wavelet and scaling functions have the advantageous property of *multiresolution*. Multiresolution refers to the ability to decompose the signal at multiple resolutions or scales. When the system functions are orthogonal and also fulfill some other basic equations, not only can they be used to approximate the signal at these various resolutions, but the information at each scale is contained in and can be determined from the information at the scale above. This leads to very fast and efficient decompositions of signals.

Multiresolution begins by choosing the scaling and wavelet functions carefully. They are found by solving the following two equations:

$$\phi(t) = 2 \sum_{k=0}^N h_0[k] \phi(2t - k) \quad \psi(t) = 2 \sum_{k=0}^N h_1[k] \psi(2t - k) \quad (3.6)$$

The first equation is called the dilation equation, the second is called the wavelet equation. There are several things to note about these equations. First, both the wavelet and the scaling function are defined based on the scaling function. Although the transform is called the *wavelet* transform, the scaling function plays a major role when the multiresolution approach is being used. Next, note that these equations involve functions at two scales (because of the $2t$ factor), and therefore are not always easy to solve. Sometimes there may be no solution. If there is a solution, it is most likely not a closed formula solution, and it almost certainly is not a smooth solution. However, it does have compact support, that is, it is zero outside $[0, N]$. Finally, for the purposes of signal and image processing it is interesting to see that the wavelet and scaling functions, which are continuous functions, can be described by a set of discrete filters, $h_0[n]$ and $h_1[n]$. (Section 3.3 details the steps taken to find the two filters used in our edge detection implementation.)

Here a simple example is introduced to help understand the concepts being discussed. The Haar wavelet system is the earliest and simplest example. The scaling function $\phi(t)$ is the box function, from 0 to 1:

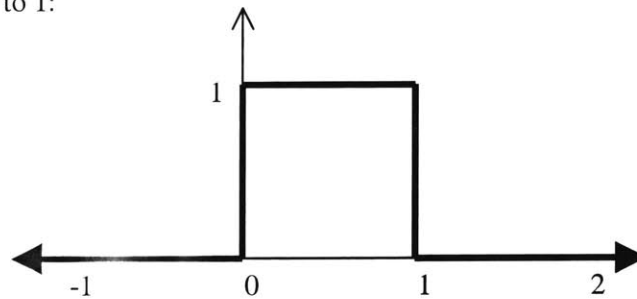


Figure 3.2. Haar scaling function

The dilation equation for this example has two terms. The two versions of the Haar scaling function on the right side of the equation shown in Figure 3.3 both contribute evenly to build $\phi(t)$. This implies that $h_0[0] = h_0[1] = \frac{1}{2}$.

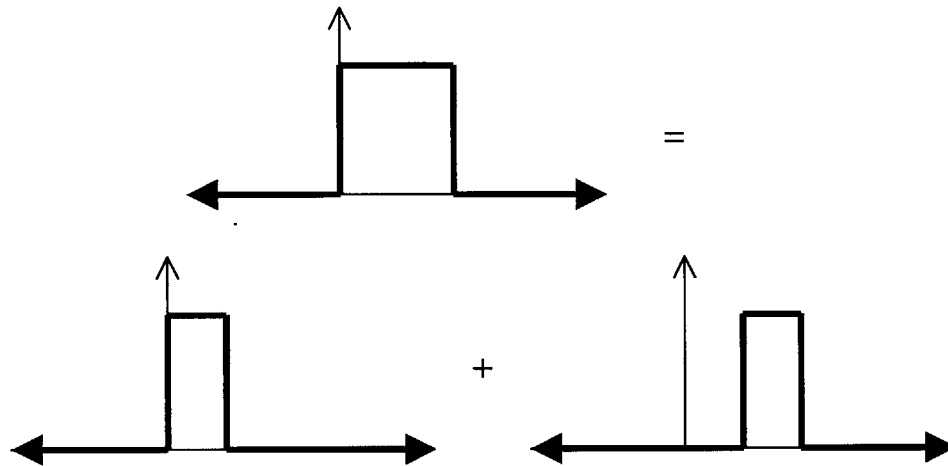


Figure 3.3. Dilation equation for Haar scaling function

Then the Haar wavelet is formed by combining the coefficients $h_1[0] = \frac{1}{2}$ $h_1[1] = -\frac{1}{2}$ with the same two components that were used to form the scaling function, as shown in Figure 3.4, corresponding to the wavelet equation in Equation 3.6.

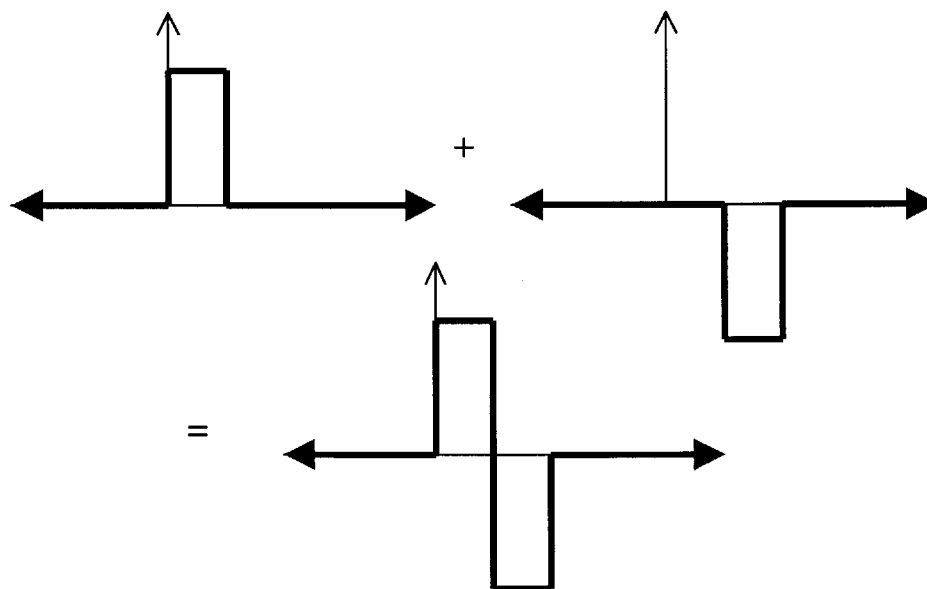


Figure 3.4. Haar wavelet function

The Haar functions display another interesting property that is common with many other scaling and wavelet functions. The scaling is an averaging procedure, while the wavelet is a differencing procedure, as can be seen by the coefficients of the h_0 and h_1 filters. In general, the scaling function will be a smoothing function, with the corresponding filter being a lowpass filter. The wavelet function usually has a corresponding filter that is a highpass filter.

If the parent scaling and wavelet functions are found by solving the dilation and wavelet equations, it can be shown that the family of functions $\phi_{j,k}(t)$ form an orthonormal basis for the space of all finite-energy continuous functions. Similarly, $\psi_{j,k}(t)$ can also be used as an orthonormal basis for this space of functions. Focusing on the scaling functions, this implies that any continuous function or signal $f(t)$ can be described exactly by using the scaling functions as building blocks. As mentioned above, the signal can also be approximated at scale j by using only the scaling functions from this scale:

$$f_j(t) = \sum_k a_j[k] \phi_{j,k}(t) \quad (3.7)$$

Similarly, $f(t)$ can be approximated at a finer scale by using thinner scaling functions, which is equivalent to increasing j :

$$f_{j+1}(t) = \sum_k a_{j+1}[k] \phi_{j+1,k}(t) \quad (3.8)$$

Because the approximation of Equation 3.8 uses thinner functions, it will be able to more accurately represent the function. The key idea of multiresolution comes from examining the difference between the approximations at the two neighboring scales, j and $j+1$. It turns out that this difference is captured exactly by the wavelet approximation at the coarser scale, j :

$$f_{j+1}(t) - f_j(t) = \sum_k b_j[k] \psi_{j,k}(t) \quad (3.9)$$

This implies that the same approximation can be made by using either $\{\phi_{j+1,k}(t)\}$ or $\{\phi_{j,k}(t), \psi_{j,k}(t)\}$.

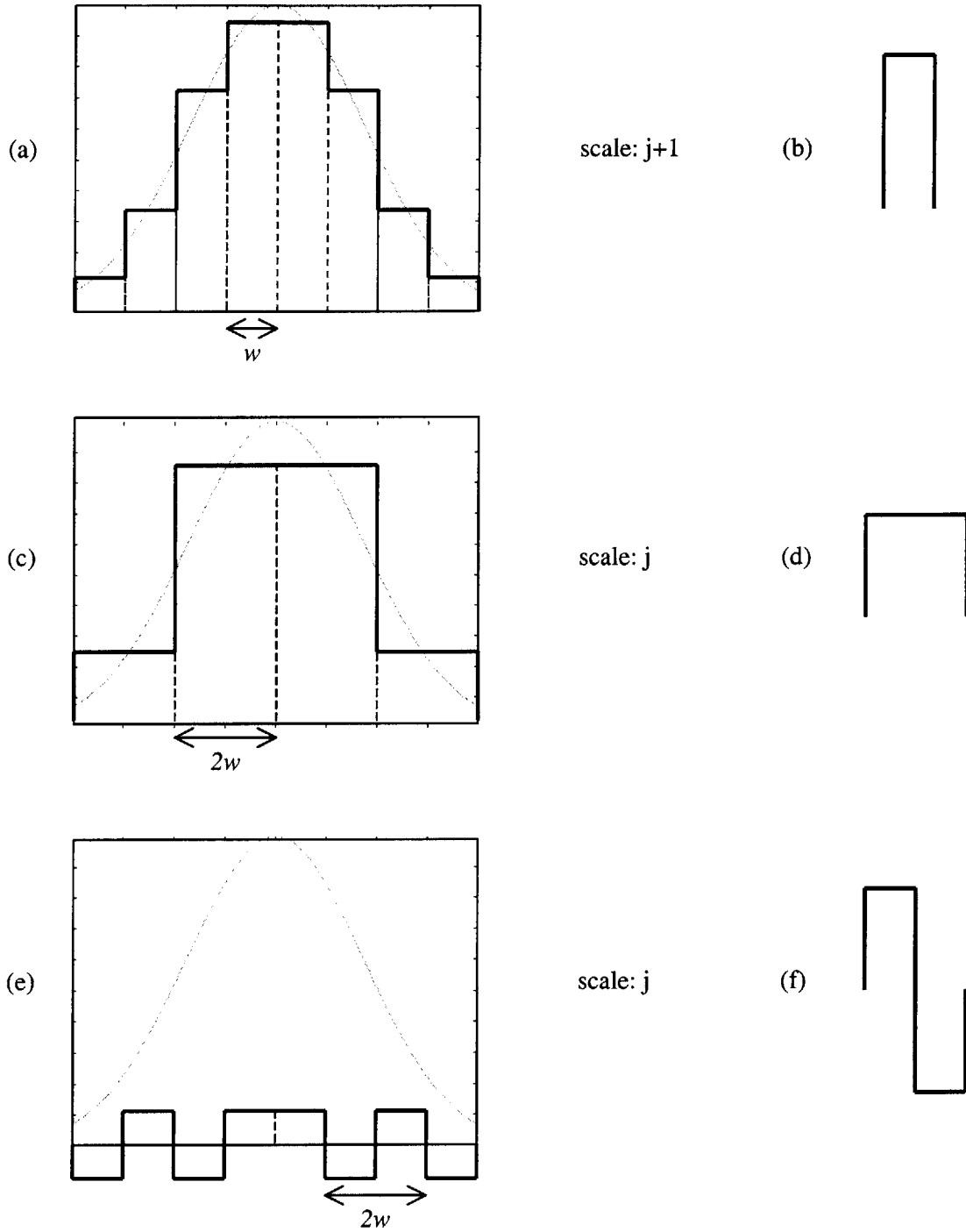


Figure 3.5. Connection between approximations of a Gaussian curve at neighboring scales, using Haar functions: (a) scaling function approximation at scale $j+1$; (b) unit scaling function at scale $j+1$; (c) scaling function approximation at scale j ; (d) unit scaling function at scale j ; (e) wavelet approximation at scale j , also difference between (a) and (c); (f) unit wavelet function at scale j

Figure 3.5 ³ on the previous page illustrates this idea using the Haar example. The first diagram shows an approximation of a Gaussian using scaling functions of width w . This finer scale is the $j+1$'s scale. Figure 3.5 (b) shows the unit scaling function being used for this approximation. The second approximation in Figure 3.5 (c) is made at the coarser scale, j , using a unit scaling function which has a width of $2w$, shown in Figure 3.5 (d). Obviously the approximation is not as good at this level. The third diagram shows the wavelet approximation at scale j , which also corresponds to the exact difference between the approximations in (a) and (c). The unit wavelet function is shown in Figure 3.5 (f). It also has a width of $2w$. It can be seen from this example that the approximation made at level $j+1$ is equivalent to the sum of the two approximations made at level j .

This concept becomes better defined by considering certain function spaces. Say V_j is the space of all functions which can be described exactly by weighted combinations of $\phi_{j,k}(t)$, where the weights are the $a_j[k]$ coefficients. W_j is the corresponding space of all functions which can be described exactly by weighted combinations of $\psi_{j,k}(t)$, with the $b_j[k]$ coefficients now becoming the weights. Then it follows from the discussion in the previous paragraph that

- $V_j \subset V_{j+1}$: If a function can be described exactly by the scaling functions at a coarser level j , the finer scaling functions at level $j+1$ can also describe it exactly.
- $W_j \subset V_{j+1}$: If a function can be described exactly by the wavelet functions at a coarser level j , the finer scaling functions at level $j+1$ can also describe it exactly.
- $V_j \perp W_j$: The two spaces are orthogonal, which follows from the orthogonality of the functions.
- $V_{j+1} = V_j \oplus W_j$: Every function in V_{j+1} is the sum of functions in V_j and W_j .

This fourth point assumes the first three, and using it the fast and easy decomposition of a signal can take place. The decomposition starts in the V_N space, where the original signal $f(t)$ can be described exactly by the scaling functions $\phi_{N,k}(t)$. The first level of decomposition calculates the $a_{N-1}[k]$ coefficients corresponding to the approximation $f_{N-1}(t)$ using the functions $\phi_{N-1,k}(t)$. The wavelet coefficients $b_{N-1}[k]$ are calculated from the approximation using the functions $\psi_{N,k}(t)$ and represent the details lost in going from the $f(t)$ to $f_{N-1}(t)$. Together, the $a_{N-1}[k]$ and the $b_{N-1}[k]$ coefficients contain the same information as the original signal. The second level of decomposition calculates the $a_{N-2}[k]$ coefficients corresponding to the approximation $f_{N-2}(t)$ using the functions $\phi_{N-2,k}(t)$, as well as the $b_{N-2}[k]$ coefficients from the approximation using $\psi_{N-2,k}(t)$.

³ This diagram is meant to illustrate the idea discussed in the previous paragraph and is not numerically exact. The heights of the box functions in this figure may not be exactly accurate.

Together the $a_{N-2}[k]$ and the $b_{N-2}[k]$ coefficients contain the same information as the $a_{N-1}[k]$ coefficients. The decomposition continues in this manner, at each level approximating the signal using wider functions and recording the difference between levels in the wavelet coefficients.

The decomposition is made even faster by directly connecting the a and b coefficients from neighboring scales. By closely examining the relationships between scales, it can be shown that

$$a_j[n] = \sqrt{2} \sum_k h_0[k - 2n] a_{j+1}[k] \quad \text{and} \quad b_j[n] = \sqrt{2} \sum_k h_1[k - 2n] a_{j+1}[k]. \quad (3.10)$$

That is, the scaling and wavelet coefficients at scale j can be directly calculated from the scaling coefficients at scale $j+1$. The only necessary tools are the two filters $h_0[n]$ and $h_1[n]$. Instead of talking about approximating continuous functions using other functions, now the decomposition involves applying a digital filter to the functions. This also means that the signals need not actually be continuous. The signal can now be discrete, as in the case of the digital images used in this project. In the case of discrete signals, finding the scaling and wavelet coefficients involves performing convolutions between the discrete values and the filter coefficients.

Thus, it can be seen again how important these filters are: not only do they describe the parent scaling and wavelet functions, but they also allow for fast determination of the coefficients at all scales of both continuous and discrete signals. In fact, there are two equations corresponding to those in Equation 3.10 for reconstructing the signal from its coefficients, and these equations also involve $h_0[n]$ and $h_1[n]$. The filters are indeed central to the implementation of the DWT, for both the 1-D and 2-D cases.

3.1.3 Discrete wavelet transform for 2-D signals

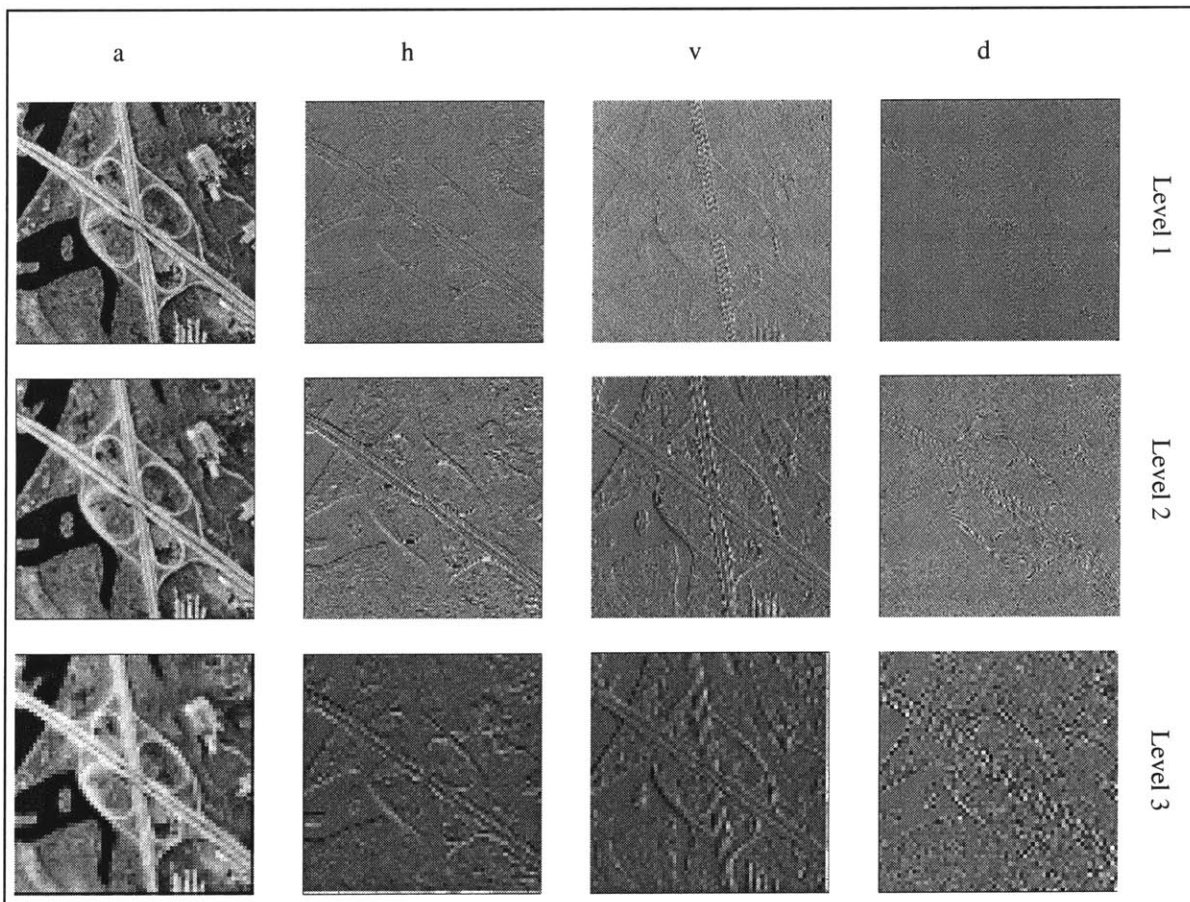
Images are 2-D discrete signals. As with the decomposition of 1-D discrete signals described at the end of the previous section, images can also be decomposed using two one-dimensional filters. The filter which corresponds to the scaling function is the lowpass filter, while the highpass filter is the wavelet filter. Both filters are applied in both the horizontal and vertical directions. This results in four sets of coefficients instead of two: $a_j[n, m]$ are the coefficients at scale j which result from applying the lowpass filter first along the rows, then along the columns; $h_j[n, m]$ are the coefficients at scale j which result from first applying the lowpass filter along the rows, then applying the highpass filter along the columns; $v_j[n, m]$ are the

coefficients at scale j which result from first applying the highpass filter along the rows, then applying the lowpass filter along the columns; and $d_j[n, m]$ are the coefficients at scale j which result from applying the highpass filter first along the rows, then along the columns. As with the one-dimensional signal, the $a_j[n, m]$ coefficients approximate the image at scale j and the $h_j[n, m]$, $v_j[n, m]$ and $d_j[n, m]$ coefficients record the differences between the approximations at scale $j+1$ and scale j . That is, in the case of images, the information contained in the four sets of coefficients at scale j is equivalent to the information contained in the $a_{j+1}[n, m]$ coefficients. Therefore, at each scale of the decomposition, both filters are applied to the last approximation to get the four sets of coefficients at the next level. The iteration is on the a coefficients.

A typical image decomposition using the Haar wavelet and scaling functions is shown in Figure 3.6. Figure 3.6 (a) shows the original image, a 500x500 image of a highway intersection. This is the same test image as used in Chapter 2. Figure 3.6 (b) shows the four sets of coefficients at all three levels of decomposition. The coefficients are shown as grayscale values in order to demonstrate what each set represents. The a coefficients represent smoothed versions of the image, where the image becomes more smoothed as the level increases. These are the lower frequency components of the image. The h , v , and d coefficients represent the horizontal, vertical and diagonal detail components respectively, the high frequency components of the image. Although these images are all shown as being the same size, each level has only a quarter of the coefficients as the level above it. Finally, Figure 3.6 (c) shows a common way of displaying the decomposition information. Despite the fact that the details of each image are not as clear as in (b), the hierarchical format emphasizes the multiresolution property that is discussed throughout this section. It also more accurately represents the actual mechanics of an image decomposition. In this representation, only the lowest level of a coefficients are displayed because only these coefficients are kept when implementing the decomposition. However, all the detail components are kept at each level, which is also true in the implementation. The advantage of arranging the sets of coefficients in this manner is that it can easily be seen that after the decomposition is completed, there are exactly the same number of coefficients as there are in the original image. Furthermore, because of multiresolution, these sets of coefficients contain exactly the same information as the original image. The information is simply in a new form, one which may emphasize different characteristics of the image. This is exactly the purpose of any transform, including the wavelet transform.

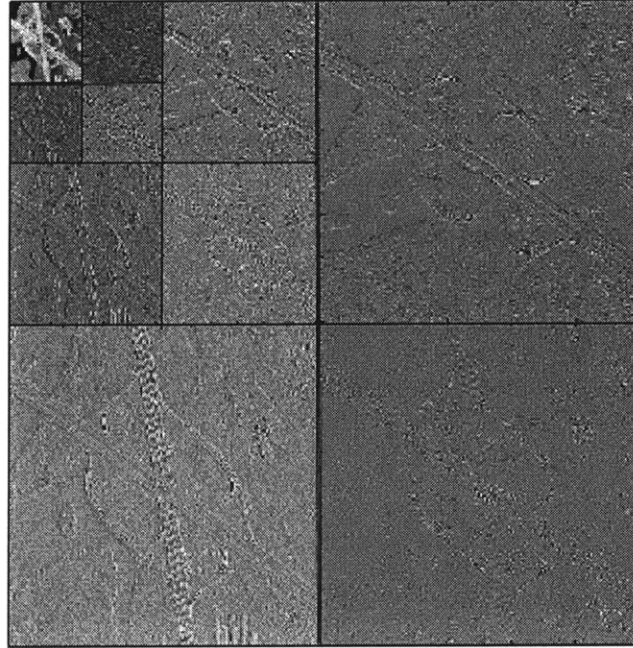


3.6 (a)



3.6 (b)

(Figure 3.6 continued on next page...)



*Figure 3.6. Three-level wavelet decomposition using the Haar functions:
hierarchical representation of the wavelet and scaling coefficients*

For the purposes of edge detection, the wavelet decomposition works very well. Consider again that the detail coefficients represent the high frequency components of the image. Edges are nothing *but* high frequency details. Therefore, the largest high frequency coefficients from the wavelet decomposition should give a good indication of the edge locations within the image. This may not be clear from the figure above. Figure 3.7 on the following page shows the absolute values of the three levels of h , v , and d coefficients from Figure 3.6. This makes it obvious that the detail components are closely related to the edges and should be able to be manipulated to yield the image edges. This is the main reason that the wavelet transform is used as an edge detection procedure in this system. Other possible advantages of using wavelets for edge detection which are not experimented with in this project are mentioned in the conclusion to this chapter.

3.2 Relation of wavelet transform to Canny edge detector

The brief introduction to wavelets in Section 3.1 mentioned the multiscale and multiresolution properties of the wavelet transform as being particularly useful characteristics to

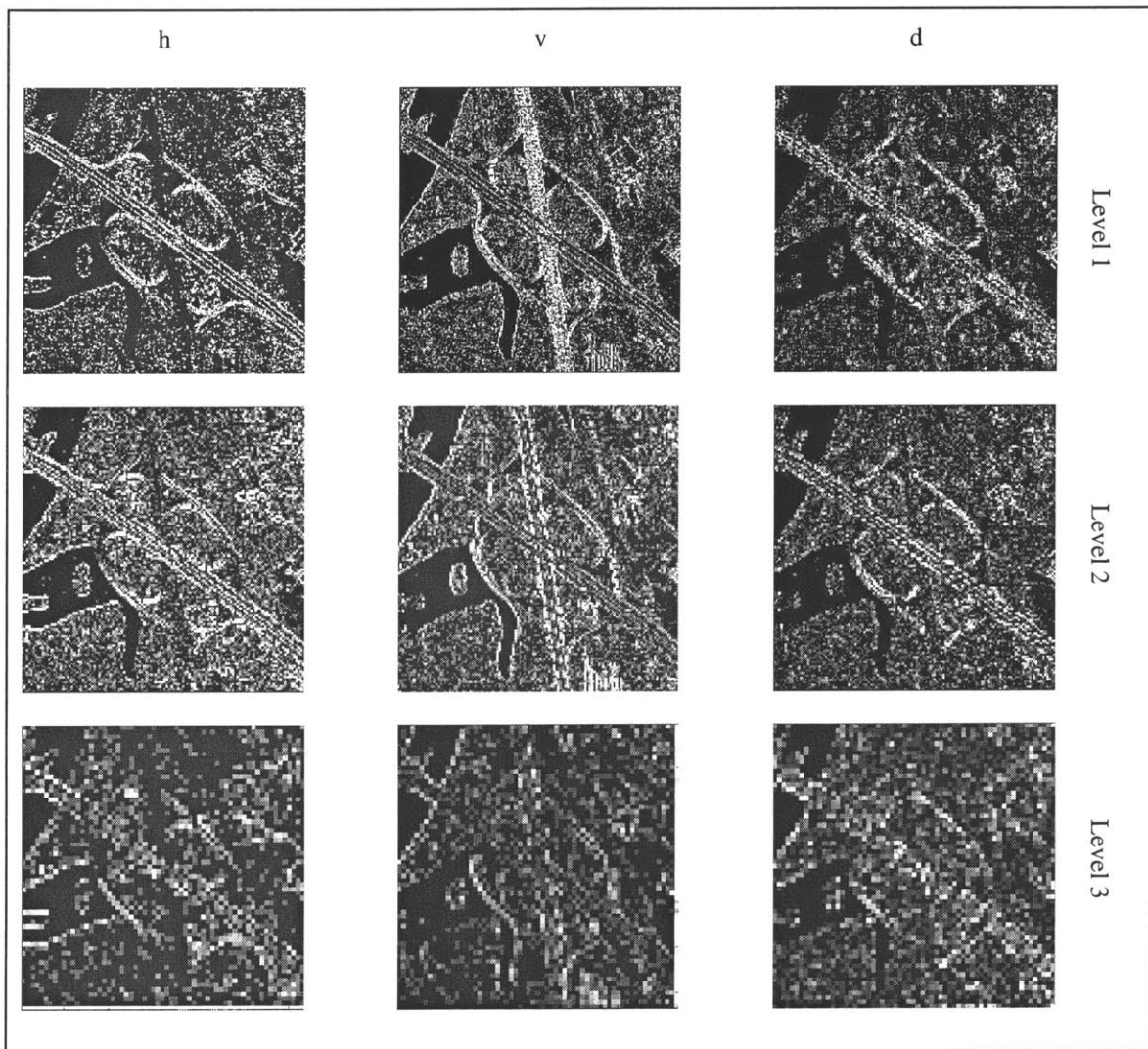


Figure 3.7. Absolute values of h , v , and d coefficients shown in Figure 3.6

exploit when using this tool for edge detection. Even though the multiresolution property is unique to wavelets, using *multiple scales* for the detection and analysis of edges is not a concept which developed with wavelet theory. Other multiscale edge detection methods designed prior to the invention of wavelets have met with notable success and are among the most relied upon procedures for extracting edges from images. Many of these methods are based on the same basic idea: first smooth the image at various scales using dilations of a smoothing function⁴, then examine the first or second derivative of this smoothed function for points representing sharp variations. The local extrema of the first derivative and the zero crossings of the second

derivative should indicate where the edges are located on the original image. Multiscale methods which follow this procedure differ by the smoothing function they use. An obvious choice for the smoothing function is a Gaussian. The multiscale method which smoothes the image using a Gaussian and proceeds to examine the second derivative of the smoothed image for zero-crossings is called the Marr-Hildreth method. Similarly, the multiscale method which examines the first derivative of a Gaussian-smoothed image for local extrema is the Canny edge detector.[3] These methods are well-established and well-liked due to the simplicity and effectiveness of their basic concept. In a paper entitled “Characterization of signals from multiscale edges”, Stephane Mallat and Sifen Zhong relate the process of using wavelets for edge detection to these popular methods, especially the Canny method.[17] By understanding the relationship between multiscale edge detection methods and the use of the wavelet transform for extracting edges, a clear picture of the latter emerges. This relationship, as explained by Mallat and Zhong, is outlined in the remaining paragraphs of this section.

The relationship is best understood if it is first explained in one dimension and then extended to 2-D for use with images. Figure 3.8 below outlines the basic process of multiscale edge detectors at a specific scale. An edge is modeled as a step, which is a generalization but still a good model. The edge has two distinct and significantly different grayscale values on either side and the actual edge point is modeled by the sharp transition between these values. Regardless of the smoothing function used, smoothing the signal has the effect of softening the edge so that the change from one grayscale value to the other is more gradual. The inflection point of this smoothed curve becomes the maximum of the first derivative curve and the zero-crossing of the second derivative. If the smoothed edge were decreasing from left to right, the inflection point would be a minimum after taking the first derivative. The arrows indicate the actual edge point. It is clear then that by finding the extrema of the first derivative of the smoothed image the edges should be found.

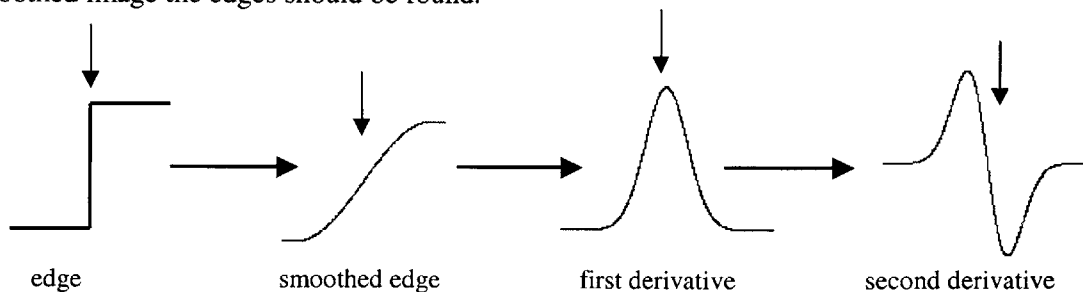


Figure 3.8. *Effect of smoothing and derivative operators on an edge*

⁴ A smoothing function is any function whose integral is equal to 1 and which converges to 0 at infinity and -infinity.

Therefore, there are two operators involved in treating the image before the extrema are found: the smoothing operator and the first derivative operator. In order to achieve equivalency between the wavelet transform and these multiscale methods, the wavelets should be designed using these two operators. As Mallat and Zhong suggest, it is possible to choose the wavelet to be the first derivative of a smoothing operator, $\theta(x)$. Then applying the wavelet to the signal f is the same as applying the derivative of θ . By changing the order of the two operators, it is finally seen that the application of the wavelet to the signal is equivalent to first smoothing the function with θ , then taking the derivative, which is the procedure used in multiscale edge detection methods as described above. These equivalencies are shown in Equation 3.11.

$$\psi(x) = \frac{\partial \theta(x)}{\partial x} \longrightarrow f * \psi(x) = f * \left(\frac{\partial \theta(x)}{\partial x} \right) = \frac{\partial}{\partial x} (f * \theta)(x) \quad (3.11)$$

So, in choosing the wavelet to be the derivative of a smoothing function, the connection is made between multiscale edge detection procedures and the use of wavelets for edge detection. This idea also extends to the two-dimensional case. Two 2-D wavelets ψ_1 and ψ_2 are chosen to be the partial derivatives of a single two dimensional smoothing function $\theta(x,y)$. When applying these two wavelets individually to the image f , the order of the derivative and smoothing operators can again be switched. The two components resulting from the application of the two wavelets to the image are put into a single matrix. They are labeled $V(x,y)$ and $H(x,y)$ here because they tend to accent the vertical and horizontal edges, as will be seen later. Because of the order rearrangement this matrix is equivalent to the gradient of the smoothed image, as shown in Equation 3.12⁵. This is identical to how 2-D multiscale edge detection methods such as Canny work.

$$\left. \begin{aligned} \psi_1(x, y) &= \frac{\partial \theta(x, y)}{\partial x} \\ \psi_2(x, y) &= \frac{\partial \theta(x, y)}{\partial y} \end{aligned} \right\} \longrightarrow \begin{pmatrix} V(x, y) \\ H(x, y) \end{pmatrix} = \begin{pmatrix} f * \psi_1(x, y) \\ f * \psi_2(x, y) \end{pmatrix} = \begin{pmatrix} \frac{\partial}{\partial x} (f * \theta)(x, y) \\ \frac{\partial}{\partial y} (f * \theta)(x, y) \end{pmatrix} \quad (3.12)$$

$$= \nabla (f * \theta)(x, y)$$

⁵ The matrix is actually equivalent to a constant multiple of the gradient, but the constant, which may be 1, is not shown in this equation for simplicity's sake.

As well as explaining this connection between the Canny edge detector and the wavelet transform, Mallat and Zhong develop a specific set of wavelet filters which satisfy the above properties and which lead to a fast implementation of the edge detection. The steps to building these filters are described in the next section. Once the edge detection is performed, the local extrema must be found in order to identify the edge points. The two components of the wavelet transform shown as $V(x,y)$ and $H(x,y)$ in Equation 3.12 are combined to form a modulus image. Then the local maxima of this image are chosen as the edge points. Finally, to improve the quality of the edge image, local grayscale variance is combined with the local modulus maxima to clear up the edges. These last three steps which actually select the edge points are detailed in Section 3.4.

3.3 Mallat-Zhong filters for edge detection

Mallat and Zhong develop a set of filters that result in the fast implementation of the 2-D wavelet transform described in the last section. Two wavelets are built which, when applied to the image, determine the two components labeled $V(x,y)$ and $H(x,y)$. The two-dimensional wavelet functions are built to be separable. This separability property is what allows for the fast implementation. The wavelets are built very carefully however, and still maintain the important property of being first partial derivatives of a single smoothing function. The combination of the two properties leads to a set of filters which are used here to extract the image edges.

In order to ensure that the two wavelets are separable, they are built as the combinations of two 1-D functions. These 1-D functions are also chosen to have very specific properties. They are a scaling function and a wavelet function, with the wavelet function itself being the first derivative of a smoothing function. The assumption is made that the scaling function ϕ has a frequency response G_0 , while the wavelet function ψ has a frequency response G_1 . In order to specify these frequency responses, consideration is made as to what properties the wavelet function should have. Sticking with the main idea behind this method, the wavelet should be the first derivative of a smoothing function θ . Mallat and Zhong also decide that the wavelet should be antisymmetrical with respect to 0 and have small compact support. Both these properties facilitate and speed up the wavelet transform implementation because the associated impulse response is finite and also antisymmetrical around 0. It is also deemed important that the scaling function be symmetrical with respect to 0, which results in a corresponding short, symmetrical

impulse response. By requiring the two functions to have these properties, the possible choices for G_0 and G_1 narrow considerably. Mallat and Zhong choose

$$G_0(\omega) = e^{\frac{i\omega}{2}} \left(\cos\left(\frac{\omega}{2}\right)\right)^{2n+1} \quad G_1(\omega) = 4ie^{\frac{i\omega}{2}} \sin\left(\frac{\omega}{2}\right) \quad (3.13)$$

, which in turn implies the function equations:

$$\hat{\phi}(\omega) = \left(\frac{\sin(\frac{\omega}{2})}{\frac{\omega}{2}}\right)^{2n+1} \quad \hat{\psi}(\omega) = i\omega \left(\frac{\sin(\frac{\omega}{4})}{\frac{\omega}{4}}\right)^{2n+2} \quad (3.14)$$

Here the wavelet is the first derivative of the function

$$\hat{\theta}(\omega) = \left(\frac{\sin(\frac{\omega}{4})}{\frac{\omega}{4}}\right)^{2n+2} \quad (3.15)$$

. $\hat{\phi}(\omega)$, $\hat{\psi}(\omega)$ and $\hat{\theta}(\omega)$ are plotted in Figure 3.9. These graphs show the antisymmetrical and symmetrical properties of the wavelet and scaling functions, as well as the fact that both have small support areas.

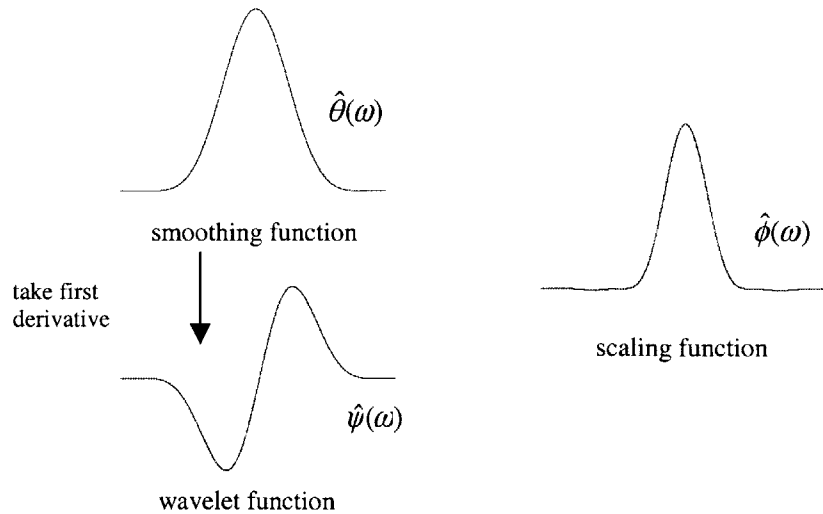


Figure 3.9. Mallat-Zhong choice of 1-D wavelet and scaling functions

Recall that the purpose of forming these two 1-D functions is to use them to build two separable 2-D wavelets. These two wavelets, ψ_1 and ψ_2 , are formed according to the following equations:

$$\psi_1(x, y) = \psi(x)2\phi(2y) \quad \psi_2(x, y) = 2\phi(2x)\psi(y) \quad (3.16)$$

These combinations of ψ and ϕ are chosen with fast implementation in mind. First, remember that to fulfill the underlying idea behind this method, ψ_1 and ψ_2 should be the partial derivatives of a single smoothing function. This is shown to be true in the numerical sense by first looking at the two smoothing functions $\theta_1(x, y)$ and $\theta_2(x, y)$:

$$\theta_1(x, y) = \theta(x)2\phi(2y) \quad \theta_2(x, y) = 2\phi(2x)\theta(y) \quad (3.17)$$

θ in this case is the 1-D smoothing function shown above in Figure 3.9. By taking the partial derivatives of θ_1 and θ_2 with respect to x and y respectively, and by recalling that the 1-D wavelet function is the first derivative of θ , it can be seen that these partial derivatives of the 2-D smoothing functions are equivalent to the 2-D wavelets from Equation 3.16:

$$\frac{\partial \theta_1(x, y)}{\partial x} = \frac{\partial}{\partial x} (\theta(x)2\phi(2y)) = \frac{d\theta(x)}{dx} 2\phi(2y) = \psi(x)2\phi(2y) = \psi_1(x, y) \quad (3.18)$$

$$\frac{\partial \theta_2(x, y)}{\partial y} = \frac{\partial}{\partial y} (2\phi(2x)\theta(y)) = 2\phi(2x) \frac{d\theta(y)}{dy} = 2\phi(2x)\psi(y) = \psi_2(x, y) \quad (3.19)$$

Next it is noted that θ_1 and θ_2 are numerically close enough that they can be considered to be the same function $\theta(x, y)$, which implies that both wavelets are the derivatives of a single smoothing function.

The purpose of describing the formation process of the 2-D wavelets is to verify this very property, as well as to emphasize the separability of the wavelets. However, the purpose of building the wavelets in this manner to begin with is to capitalize on the simplicity of the impulse responses G_0 and G_1 by basing the wavelet transform implementation on their use. That is, due to the wavelets ψ_1 and ψ_2 being separable into the component functions ϕ and ψ , a very simple implementation arises involving the impulse responses $g_0[n]$ and $g_1[n]$ associated with the

component functions. As was mentioned earlier, G_0 and G_1 are chosen so that ϕ and ψ have short support and are symmetrical and antisymmetrical respectively. This results in short impulse response filters g_0 and g_1 ⁶ with the same symmetric properties, shown in Table 3.1.

n	$g_0[n]$	$g_1[n]$
-1	0.125	
0	0.375	-2.0
1	0.375	2.0
2	0.125	

Table 3.1. Impulse response filters used for wavelet transform implementation (as shown in [17], Table I)

At each level of decomposition k , these filters are applied to generate three component images, two which highlight the horizontal and vertical details, H_k and V_k , and one which is the smoothed image, S_k . Here, k is not the same as the dilation variable j from previous sections in the chapter. k is a counter for the levels of decomposition, and can be thought as being equal to $N-j$. Note that whereas j decreases with each level of decomposition, k increases. Although introducing a new notation may be slightly more confusing, it follows the conventions of Mallat and Zhong's paper and is therefore kept. Returning to the decomposition, the iteration is on the smoothed image, that is, H_k , V_k , and S_k are generated by decomposing the smoothed image at level $k-1$, S_{k-1} . The g_1 filter does the differencing, so it is used to compute the detail components: H_k is found by convolving the columns of S_{k-1} with g_1 , while $V_k(x,y)$ is found by convolving the rows of S_{k-1} with g_1 . The g_0 filter does the smoothing, so S_k is found by convolving both the rows and columns of S_{k-1} with g_0 . This algorithm is summarized as follows, for N levels of decomposition:

```

k = 1;
S0 = original image;
while k <= N
    Hk(x,y) = Sk-1*(D, g1);
    Vk(x,y) = Sk-1*(g1, D);

```

⁶ These filters correspond to the h_0 and h_1 filters from Section 3.1. They are labeled using g 's instead of h 's to prevent confusion with the horizontal coefficients, H_k , introduced below.

```


$$S_k(x,y) = S_{k-1} * (g_0, g_0);$$


$$k = k + 1;$$

end

```

where D is the dirac function (1 at 0 and 0 everywhere else), and $A*(B, C)$ represents a 2-D separable convolution where the 1-D filters B and C are convolved in one dimension with the rows and columns of A , respectively. Figure 3.10 on the next page shows the output from the first three levels of decomposition of the test image shown in Figure 3.6.

3.4 Selection of edge points

The wavelet transform is applied to a test image in the hopes that the edges will be more obvious in the output images than in the original. Specifically, the antisymmetrical shape of the wavelet filter should highlight the edges in the V_k and H_k output components. The smoothed component S_k is useful during the implementation of the transform for iteration purposes, but the next step focuses solely on isolating the edge information from the V_k and H_k components. Recall that the method to extract edges revolves around first applying a wavelet that is the first derivative of a smoothing function, which is done in the previous section, and then finding the local extrema of the output signal. There are two output signals from this implementation. To perform the local extrema search, V_k and H_k are first combined to form a modulus image. Then a search is made for local maxima within this image. (Minima are ignored because the modulus image is made up of absolute values). After the local maxima are found, an additional step is added to those outlined by Mallat and Zhong in order to clean up the edge image. The local maxima information is combined with local gray scale variance information in an attempt to discard many of the useless, “noisy” edges that are not wanted for the remaining steps of the road detection process.

3.4.1 Calculating the modulus image

The modulus image is created using the vertical and horizontal edge components outputted from the wavelet transform. Because there is a V_k and H_k component at all

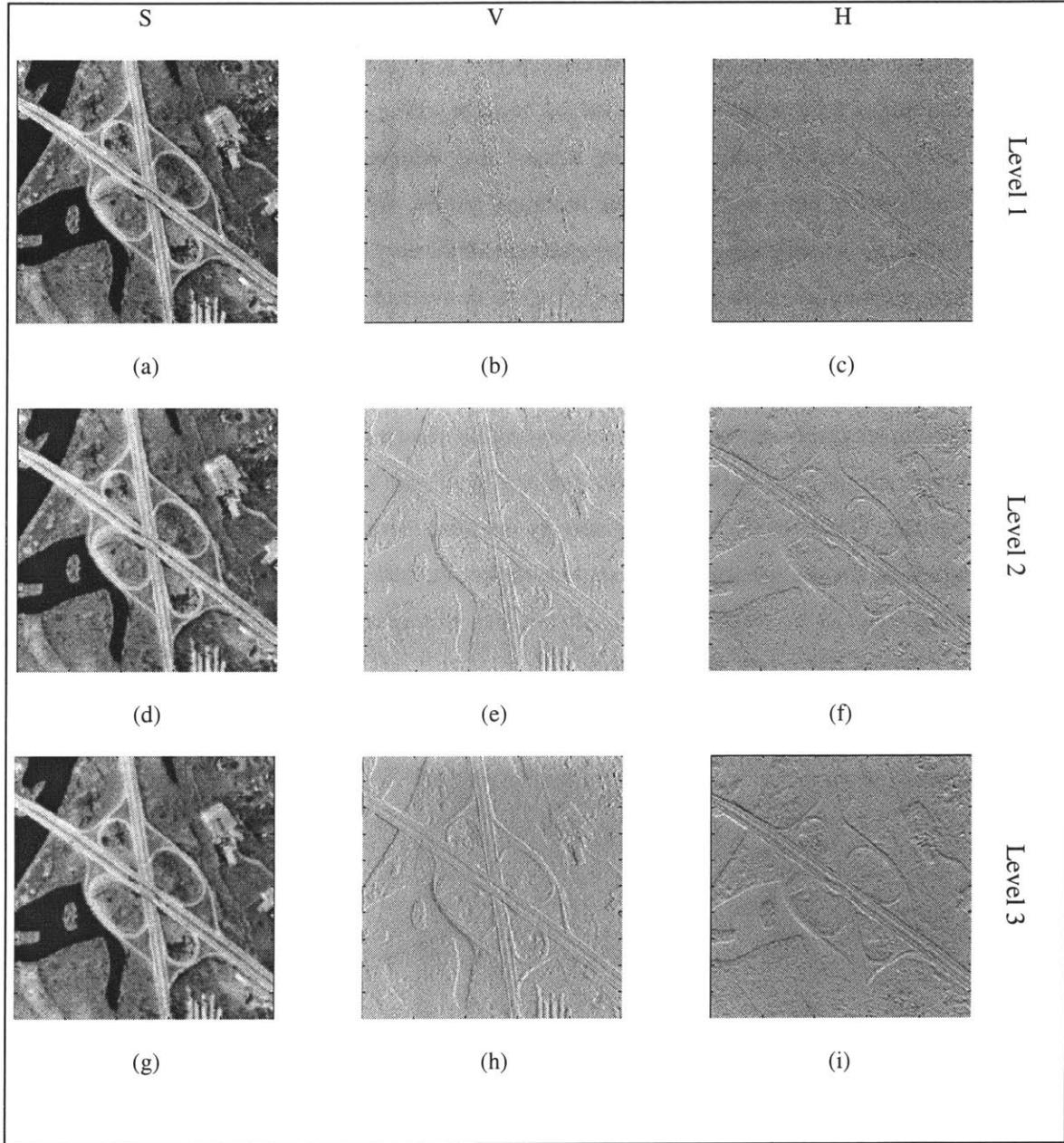


Figure 3.10. Decomposition of test image using Mallat-Zhong filters: (a) $S_1(x, y)$; (b) $V_1(x, y)$; (c) $H_1(x, y)$; (d) $S_2(x, y)$; (e) $V_2(x, y)$; (f) $H_2(x, y)$; (g) $S_3(x, y)$; (h) $V_3(x, y)$; (i) $H_3(x, y)$

decomposition levels, there is also a modulus image for each level. The calculation of the modulus values is simple:

$$M_k(x, y) = \sqrt{|V_k(x, y)|^2 + |H_k(x, y)|^2} \quad (3.20)$$

By combining $V_k(x,y)$ and $H_k(x,y)$ in this manner, all orientations of edges are accounted for, not just those in the vertical or horizontal directions. It is clear that points lying along vertical or horizontal edges have large modulus values because either $V_k(x,y)$ or $H_k(x,y)$ has a large absolute value. Certainly points such as corners and intersections which lie along both a horizontal and vertical edge have very large modulus values. But in addition to these cases, any point which lies on a sharp edge of *any* orientation should also have a relatively large modulus value. Along such edges, both $V_k(x,y)$ or $H_k(x,y)$ have average or larger than average absolute values, which when squared and combined, lead to a large modulus value. Therefore, it is expected that most edge points should be emphasized in the modulus image. Figure 3.11 shows the modulus images for the three levels of decomposition shown in Figure 3.10. At all levels, the strongest edges are the road edges and other major edges, such as those of the buildings and the banks of the lake. These edges become clearer by the third level. However, other edges such as those representing the foliage are also present as weaker but still obvious edges. It is hoped that the remaining steps in the edge detection process can eliminate some of the noisy, unwanted edges while keeping the strong, desired edges.

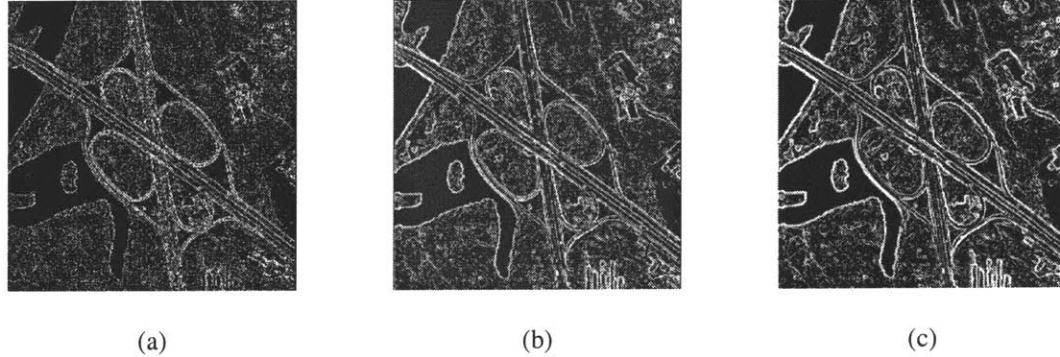


Figure 3.11. Three levels of modulus images: (a) $M_1(x,y)$; (b) $M_2(x,y)$; (c) $M_3(x,y)$

3.4.2 Finding the local maxima

The largest modulus values clearly represent the strongest edges in the original images. However, to simplify further processing of the edges, it is necessary to create a binary image based on the modulus image in which the edge points have a value of 1 and all other points are 0. One possibility which may seem attractive is to simply threshold the modulus values. The problem with this solution is that choosing the threshold is not trivial. Instead, the final step

illustrated in Figure 3.8 is taken. That is, a point from the modulus image is chosen to be an edge point in the binary image if it is a local maximum.

With a 2-D signal such as the test images, it is important to decide in which direction the point must be a local maximum for it to become an edge point. It makes sense that it need not be a local maximum *along* the direction of the edge it is a part of, but rather in the direction *normal* to the edge direction. Thus, it is necessary to know the direction or angle of the edge of which a point is a part. Luckily, this information can be easily calculated from the value of V_k and H_k . The edge angle at a point (x,y) is given by the following equation:

$$A_k(x, y) = \text{argument}(V_k(x, y) + iH_k(x, y)) \quad (3.21)$$

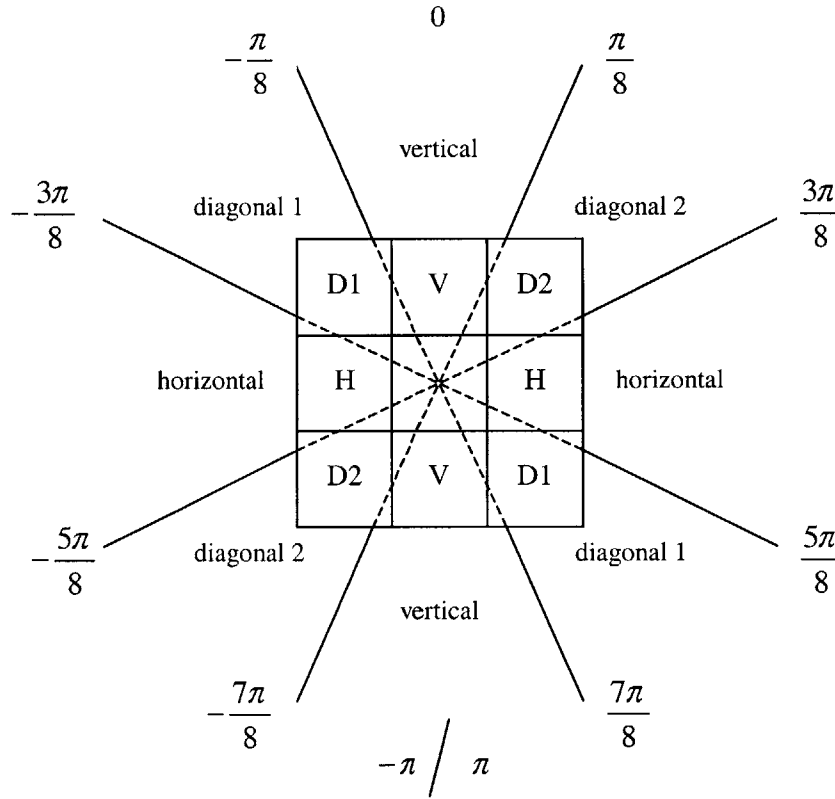


Figure 3.12. Assignment of the edge direction based on the point angle

In order to determine if a point is a local maximum, $A_k(x, y)$ is found and then the edge at that point is classified as one of four directions, horizontal, vertical, diagonal 1 or diagonal 2, based on the angle value. Figure 3.12 illustrates how these directions are assigned. The direction normal to the edge direction is determined. Then the modulus value of the central point is

compared to the modulus values of the two neighbors on either side in this normal direction. If it is greater than these two points, it is deemed a local maximum and therefore becomes an edge point. The local maximums found in this manner for the modulus images of Figure 3.11 are shown below in Figure 3.13. Although these are not the clearest images, it should be apparent that as the level number increases, that is, as the scale becomes coarser, the desired road edges and other major edges become more complete and there is slightly less clutter from noisy edges.

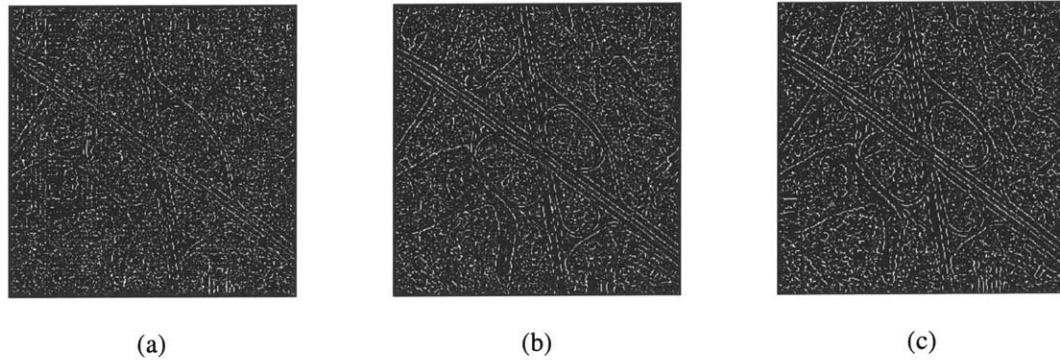


Figure 3.13. Three levels of local maxima images: (a) level 1; (b) level 2; (c) level 3

Finally then, the process outlined in Section 3.2 has been completed. Although the process is not as simple as that depicted in Figure 3.8, in essence the steps are those of smoothing and taking the derivative, by means of the wavelet transform, and finding the local extrema.

3.4.3 Introducing local grayscale variance

This final step is a means of cleaning up the edge image. Choosing local maxima from the modulus image is a very local calculation. This is obvious in the images from Figure 3.13 which are filled with edges, both wanted and unwanted. Certainly some of these edges are stronger than others in the original image. Stronger edges have a larger difference in the grayscale values on either side of the edge points. So it should be possible to eliminate some of the weaker edge points which are present in the local maxima image by also considering the local gray scale variance around the points.

The local grayscale variance is used in the following way: For each image pixel, a small neighborhood of points around the pixel is taken into account. The mean grayscale value of the points in the neighborhood is calculated. Then the variance is calculated as the squared sum of the differences from this mean to the values of all neighborhood points. This variance is assigned

to the original center pixel. If a point is part of or is located close to an edge, then it is expected that that edge will run through the considered neighborhood. If this is so, there should be a larger grayscale variance within the neighborhood. Therefore, once all image pixels have a sum of variances associated with them, the points with the larger variances are labeled as edge points, and those with smaller variances are labeled as background. The cutoff point between the two is the mean of all the sums of variances. Figure 3.14 shows an example of the image which results when this process is run on the test image from Figure 3.6, considering 7x7 neighborhoods. The black points are those which have a local variance higher than average, and are thus designated edge points.

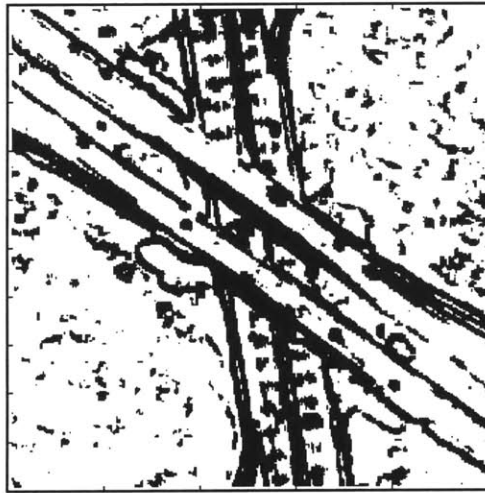


Figure 3.14. Example of using grayscale variance for edge detection

Although this grayscale variance calculation is also very localized, the combination of the information it produces with the local maxima information leads to the removal of several unwanted edges from the edge image. It can be seen from the figure above that if a large enough neighborhood is used, many of the smaller, noisy edges do not appear in the output. Therefore, eliminating edges from the local maxima image which do not have corresponding areas of black points in the grayscale variance image produces a cleaner edge image with fewer undesirable edges. For example, Figure 3.15 shows the combination of the local maxima image from Figure 3.13 (c) and the local grayscale variance image from Figure 3.14. It is an image such as this that is passed on to the next stage of the road detection system.



Figure 3.15. Combination of Figure 3.13(c) and Figure 3.14

3.5 Conclusion

This chapter serves as an introduction to one of the newer tools being used for edge and feature extraction, that is, the wavelet transform. Specifically, the focus is on the work done by Mallat and Zhong, the filters they built to perform a fast implementation of the wavelet transform and the multiscale/multiresolution edge detection concepts behind the filters. This is not the only way to perform edge detection using wavelets, but it is a method that works well and gives insight into the characteristics of wavelets through the explanation of the details.

Obviously the technique described in this chapter is quite different and certainly more theoretically complex than the Nevatia-Babu method described in Chapter 2. Although in this case there is no explicit tailoring of the technique to extract linear or road features, the wavelet method still has some unique advantages.

- As was mentioned at the end of Section 3.1.3, three of the four sets of coefficients generated at each level of the wavelet decomposition contain high frequency details of the image. These details correspond very closely to the strongest edges of the image and can be exploited for edge detection procedure.
- The size of this chapter as compared to Chapter 2 is an indication that the theory behind the wavelet method is more complicated than for some other edge detection techniques. In fact, to truly understand wavelets and their properties requires much

more than simply reading this chapter. However, although there is more knowledge required to understand where the algorithms and filters come from, the implementation of the DWT for edge detection is very fast and easy. This is evident from the simplicity of the algorithm stated in Section 3.3. The edge detection procedure involving wavelets is as fast or faster than the Nevatia-Babu method, and usually gives better results.

- The separation of details by scale means that not all the detail information need be considered. It is easy to see from the output of the DWT which scales provide the best edge information. This is largely based on the size/width of the edges being detected. It is possible to focus on the best scale and ignore a lot of other information that may decreased the overall quality of the results. For the broader problem of feature detection where the features may be buildings or cars, etc., it may also be possible to predetermine which scale will produce the best results based on the size of the specific feature. This could potentially reduce the amount of computation which needs to be done. This was not attempted in this project.
- Also not undertaken in the building of this system was the tailoring of the parent wavelet and scaling functions to a specific task. There are a huge number of functions which satisfy the conditions for being a wavelet or scaling function. In fact, there are a great deal more conditions to consider when building one's own wavelet than are mentioned in this chapter. It is possible to tailor them quite specifically, and it is conceivable that the parent functions could be built to increase the success in detecting a specific feature. This of course would require a deeper knowledge of both the properties of wavelets and of the feature to be detected.

For these reasons, it seems worthwhile to experiment with wavelets as an edge detection tool in this road detection system.

Chapter 4.

EDGE LINKING

4.1 Introduction

No edge detection method is perfect in that it finds all the edge points and only edge points. Usually, unwanted noise is present in the form of short, erratic edges, and many edge pieces remain disconnected from other pieces on the same edge by gaps. Both these problems are addressed in the edge linking process. In general, the goal of edge linking is to fill the gaps in edges by somehow deciding which pieces should be connected and by joining them. Numerous methods have been devised to perform this task. As with the problem of edge detection, specifically road detection, linking methods are either local or global-based systems.

Local methods deal with a neighborhood around the pixel to be linked, analyzing the characteristics of these neighboring pixels to determine where the links should be made. The simplest of local schemes focus on a small neighborhood and use information from earlier edge detection in order to find pixels with similar characteristics. For example, if edge angle and magnitude values are found for each pixel, as in the Nevatia-Babu method described in Chapter 2, linking between two points may take place if the difference in angles and/or the difference in magnitudes is below some fixed threshold.[8] As the size of the neighborhoods gets larger, the searches for points to link become more complicated and a variety of path metrics are introduced to help the process. A sequential search of points to link may use a maximum likelihood metric which calculates the most likely path between certain points.[5] Fuzzy reasoning has also been used to deduce which pixels in the search area should be linked together. In this case, a variety of

local properties between points, such as alignment, proximity and interim edgeness, are analyzed together to find the best link.[15]

Global methods look at the overall pattern of edges and try to describe the features using a few variables. In one such method, the entire edge image is modeled as a potential function, where each edge point exerts a force on all other edge points, and links are made between points with the strongest attractions.[24] Another common global method is to use the Hough transform to describe an edge as a curve of specified shape and estimate the missing edge points from the known formula of the curve.[8] Other tools used for global edge linking are Markov Random Fields [7], and Least-Square-Error curve fitting [19].

4.2 Overview of edge linking technique

The linking procedure described in this work is a local method and specifically focuses on linking edges which represent roads. The method revolves around the idea of using the edge direction to outline a search area in which points to be linked will be found. The same idea underlies a simple edge following process developed by Heipke, Englisch, Speer, Stier and Kutka.[11] After thinning the edges, they use the angle associated with a specific edge point to determine a small search area. The points within this small area are numbered and checked in ascending order for possible edge points to link. A sample of one of their search matrices is shown below in Figure 4.1. This matrix corresponds to an angle of 45 degrees at the point p . From the choice of placement of the numbers within the matrix, it is apparent that both the proximity of a point to p and how close the point is to the edge direction of p are important considerations in choosing a point to link with p . These considerations carry over to the edge linking method described in this chapter. Once our search area is determined, it is searched in a manner such that points closer to the root point are search first, as are points which lie along the principal search direction. However, the Heipke method will obviously only fill small gaps, of five pixels or less. Also, their search area is based on a single edge angle calculated at one point. Our edge linking process can not only fill gaps of arbitrary largeness, but a good part of the edge linking focuses on accurately finding the direction of an edge so as to have a better search area.

So, although the general idea of search areas for edge linking has been used before, the details of the searches performed by this method are new. As mentioned above, steps are taken to determine which edge points are actually a part of the true edge leading away from an end point. Once this is accomplished, an estimation is made as to the direction in which the edge is heading.

20	18	16	14	13
11	9	7	6	15
4	2	1	8	17
		3	10	19
p		5	12	21

Figure 4.1. Example from [11]

The primary and secondary directions, found by looking at the last few step directions within an edge, are assigned to each end point as candidates for the true edge direction. We assume roads are relatively straight, so we expect the road edges to continue in the direction they are heading. Therefore, searching is done away from the end point in the primary and secondary directions. In this manner, links are made which fill the gaps in road edges.

4.3 Details of edge linking technique

4.3.1 Determining end points

In our system, as in many instances of edge linking, the links are made between the end points of edge segments. This ignores some possible cases of edge joins. For instance, a road may contain a fork, which appears in the shape of a *Y*. If the base of the *Y* and one upper branch are clearly visible as a complete edge after edge detection, but a gap disconnects the second branch from the base, there could be a problem. In this case, the end of the disconnected edge should be linked to a point somewhere in the middle of the larger piece. By limiting the linking to occur between end points, gaps such as this one may be missed. However, not many gaps are missed by considering only end point to end point links, and it simplifies the process somewhat.

Therefore, the first step in the linking process is to establish which edge points are end points. End points are defined by Zhu in [24] as edge points which satisfy one of two conditions. An edge point is also an end point if and only if (i) it has only one other edge pixel in its 3x3

neighborhood, or (ii) it has two edge pixels in its neighborhood that are directly adjacent.⁷ However, if both these conditions are used to determine the end points, there are many points designated as end points which are really just left over glitches from the thinning process. The second part of Zhu's definition leads to these unwanted end points, which tend to be single points sticking out of the sides of longer edges. Thus, part (ii) of the definition is ignored and end points are determined from only part (i). This means that there are eight possible arrangements of pixels in the surrounding 3x3 neighborhood that result in the center pixel being labeled as an end point, shown in Figure 4.2. The neighborhood of each known edge point is compared to all eight masks in order to determine the end points.

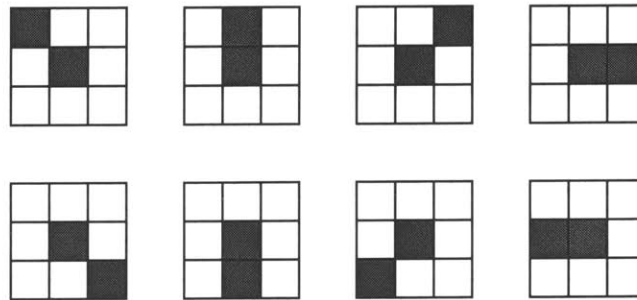


Figure 4.2. Masks for finding end points

4.3.2 Tracing the tree

The next step involves forming a tree to represent each edge. Unfortunately, the edge detection very rarely returns a clean set of edges, each with an obvious beginning and end and a single edge path in between. There may be some such edges in the image passed to the linking process, but there are also many messy edges which do not have a clear beginning or ending. Most have multiple branches, with some branches not being a part of the desired edges. Therefore, each connected group of edge branches is represented in the program as a tree, with endpoints forming the root and the tips of branches. The tracing of all branches begins at the root, with all edge pixels being recorded in one and only one tree.

If the edge is a simple, clean edge such as described above, the tree will only have one branch and the tracing is simple. Beginning at the root point, a search is made in the 3x3

⁷ Two pixels are directly adjacent if they have a difference of one in only one of their coordinates, while the other coordinate is the same. For example, in a 3x3 block of pixels, the upper left corner and the upper center pixels are directly adjacent, while the upper left corner and the upper right corner pixels are not.

neighborhood of the point for other edge points. If an edge point which is not already in the branch is found in this neighborhood, it is recorded as the next pixel in the tree and the search is repeated for this new tree point. This continues along the length of the branch. Once a point is reached which has no other new (i.e. not already in the tree) edge points in its immediate neighborhood, the branch ends, as does the tree.

With multiple-branched trees, the tracing is slightly more complex. The initial branch leading away from the root is traced in exactly the same way as the single-branched tree. However, at some point, there will be more than one new point in the 3x3 surrounding neighborhood. This point is labeled as a branch point and signifies the end of one branch and the beginning of another. Actually, each edge point in the branch point's neighborhood is recorded as being the beginning of new branch, and each of these branches is traced out in the manner detailed above, perhaps ending in an end point, perhaps itself branching into more edge segments. A stack is used to keep track of which branches have already been traced, ensuring no branches are missed. Also, each branch has a number, and branches which are connected are kept track of by listing the pairs of numbers. This allows for the entire network of branches to be recorded as a single tree, containing all the edge pixels in that edge group. In addition, all end points contained in the tree are noted so that a single tree with multiple end points is only traced once.

In this way, each edge is catalogued as a series of branches. The representation of the edges as trees has the advantage of providing the needed edge information in a simple format. If a certain branch or even a certain edge point needs to be accessed, it can be quickly located within the list of branches. This not only helps with the remaining steps of the linking scheme, but with later steps of the road detection process as well. In addition, the branch lengths are used to remove some of the edge noise by deleting trees (edges) with a small total length.

4.3.3 Finding primary and secondary directions

As was mentioned above, linking occurs between end points. Each end point now represents the base of a tree of connected edge segments. In order to effectively link this base end point with another end point, an area must be determined in which to search for possible points to link with. It is expected that links with the base end point will be made away from the edge, in approximately the direction that that edge is heading when it reaches the end point. Therefore, the search area must be based on this edge direction. This makes it critical to accurately establish in what direction the edge is heading when it reaches the end point. To do this, both a primary and secondary direction are associated with each end point. These two

directions have eight possible values, representing the eight unit step directions, as shown in Figure 4.3. The specifics of how to get these directions are detailed below, but basically the edge direction is broken into unit steps and the primary and secondary directions are the first and second most prevalent step directions. Finding these directions requires two steps: (i) deciding which branches in the edge should be included in the direction calculation, and (ii) actually calculating the edge direction and determining the primary and secondary directions.

1	2	3
4		5
6	7	8

Figure 4.3. Unit step directions

For the simple single-branched tree, the first step is very easy: the one branch is used to make the direction calculation. With a multiple-branched tree, the choice is not immediately clear. All combinations of branches which start at the root point are considered. Assuming that road edges do not change direction very much or very quickly, the best combination of branches for making the direction calculation will be the combination that produces the straightest and most continuous edge. To measure the straightness of each branch combination, an overall direction is determined for each of the combinations by calculating the angle between the first point in the combination (the root point) and the last point. The individual directions of each branch in the combination are also found as the angle between the beginning and end points of the branch. These directions can range anywhere between -90 and 90 degrees. To determine how straight the combination is, the absolute differences between each branch direction and the overall combination direction are added. The combination with the smallest sum of absolute differences is determined to be the best combination. Previously, the best combination was taken to be the longest. The choice of the straightest leads to slightly better results with no real increase in complexity. There is no guarantee that either choice is the true edge, but in most cases it is the sufficient for the final steps of edge linking.

The best combination is the edge used to calculate the directions associated with the endpoint and its search area. Given the combination of branches, this calculation is quite easy. In order to calculate the primary and secondary directions, only the last twenty steps of the edge leading up to the end point are used. These last twenty unit steps are polled, with the first and second most prevalent directions becoming the primary and secondary directions associated with

the end point. As Figure 4.3 shows, the step directions, and hence the primary and secondary directions, are limited to eight possible values. The number of times the primary and secondary directions occur in the last twenty steps are called the primary and secondary confidences, indicating roughly how strongly the edge is moving in each direction.

4.3.4 Searching for links & edge linking

Finally, once all end points have their primary and secondary directions, the search begins for possible links. Assuming road edges do not change direction suddenly, we expect to be able to continue a broken edge in the direction it is heading in order to find appropriate edge pieces to link it with. Therefore, as mentioned above, search areas for a particular end point are based on that point's primary and secondary directions.

For a given end point, the search area is determined as follows: Say x is the primary confidence and y is the secondary confidence. A basic step pattern is determined consisting of $x + y$ steps, with x of those steps being in the primary direction and y in the secondary direction, and with the y steps in the secondary direction being evenly spaced throughout the $x + y$ steps. For example, if the primary step direction was down and to the right, labeled as 8, and had a confidence of 13, and the secondary direction was to the right, labeled as 5, with a confidence of 6, the basic step pattern would consist of $13 + 6 = 19$ steps and would look like “8 8 5 8 8 5 8 8 5 8 8 5 8 8 5 8 8 5 8”, shown in Figure 4.4(a). Starting at the given end point, the step pattern is

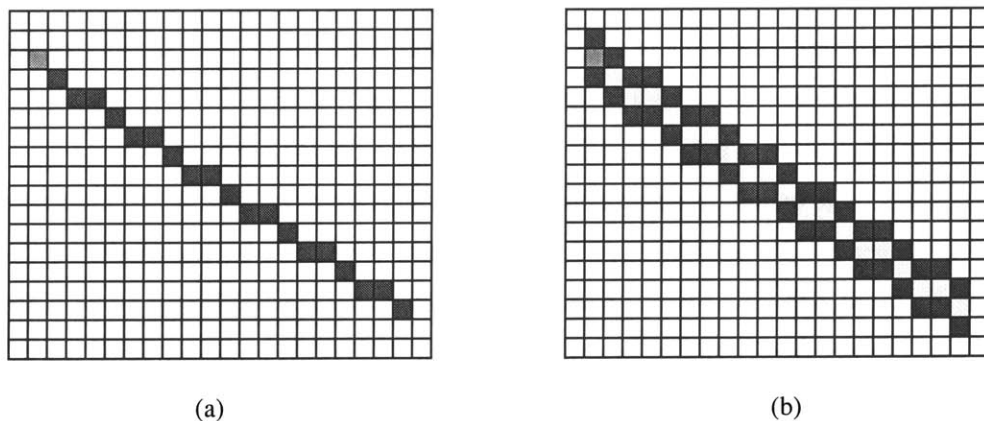


Figure 4.4. Example of finding search area from primary and secondary directions: (a) first nineteen steps of center path; (b) paths followed after taking one step to the “left” and one to the “right”

repeated until a certain pre-stated number of steps have been taken away from the end point. Then the search returns to the starting end point, takes one step to the left and one to the right, and uses these two points as new starting points for the step pattern. The left and right step directions are determined by the primary and secondary directions in order to ensure that no pixels within the bounds of the search area will be left unchecked and no pixel is checked twice. For instance, continuing the example from above, the “left” step would be taken upwards and the “right” step downwards, as in Figure 4.4(b). Ten steps are taken away from the end point both to the left and the right, with the step pattern search beginning at each new point and continuing outward. The search area then is an irregular box/parallelogram with the end point in the middle of one of the shorter ends.

Because the links are made between end points, all end points in the search area are recorded. We call the end point which is the base of the search the root point, and any end point found within the search area is labeled as a candidate end point. Once all candidate end points are located, the series of questions outlined in the chart shown in Figure 4.5 on the next page are used to determine which points should be linked to the root.

The questions revolve around two measures: (i) the distance between the root and the candidate end point, and (ii) the number of steps off the center path that the candidate point lies, which is equivalent to how far off the determined edge direction the candidate point lies. The values of these two measures determine four zones within the search area where linking is possible. The first zone is the center path leading away from the root point (steps off center = 0). If a candidate point is in this zone, linking takes place. If no points lie in zone one, the candidate end point with the shortest distance to the root is found and becomes the target point for linking. Zone two consists of all points 1, 2 or 3 steps off center. If the target point lies in this zone, the link is made. Zone three contains all points between 4 and 6 steps off center, inclusive. Zone four consists of all remaining points in the search area (7+ steps off center). If the target point is in either of zones three or four, a check is made on the alignment of the root and the target. The alignment test succeeds if the difference in their primary directions is less than or equal to 90 degrees. If the point is in zone three and the alignment test succeeds, linking occurs. If the point is in zone four and the alignment test succeeds, linking is done if the candidate point has at least a distance of 20 from the root. This final distance check prevents linking with points that are in the bottom corners of the search area, which tend to be bad links. To summarize, candidate points in zone one are always linked; a point in zone two is linked if it has the shortest distance to the root; a point in zone three is linked if it has the shortest distance to the root and is aligned with the root;

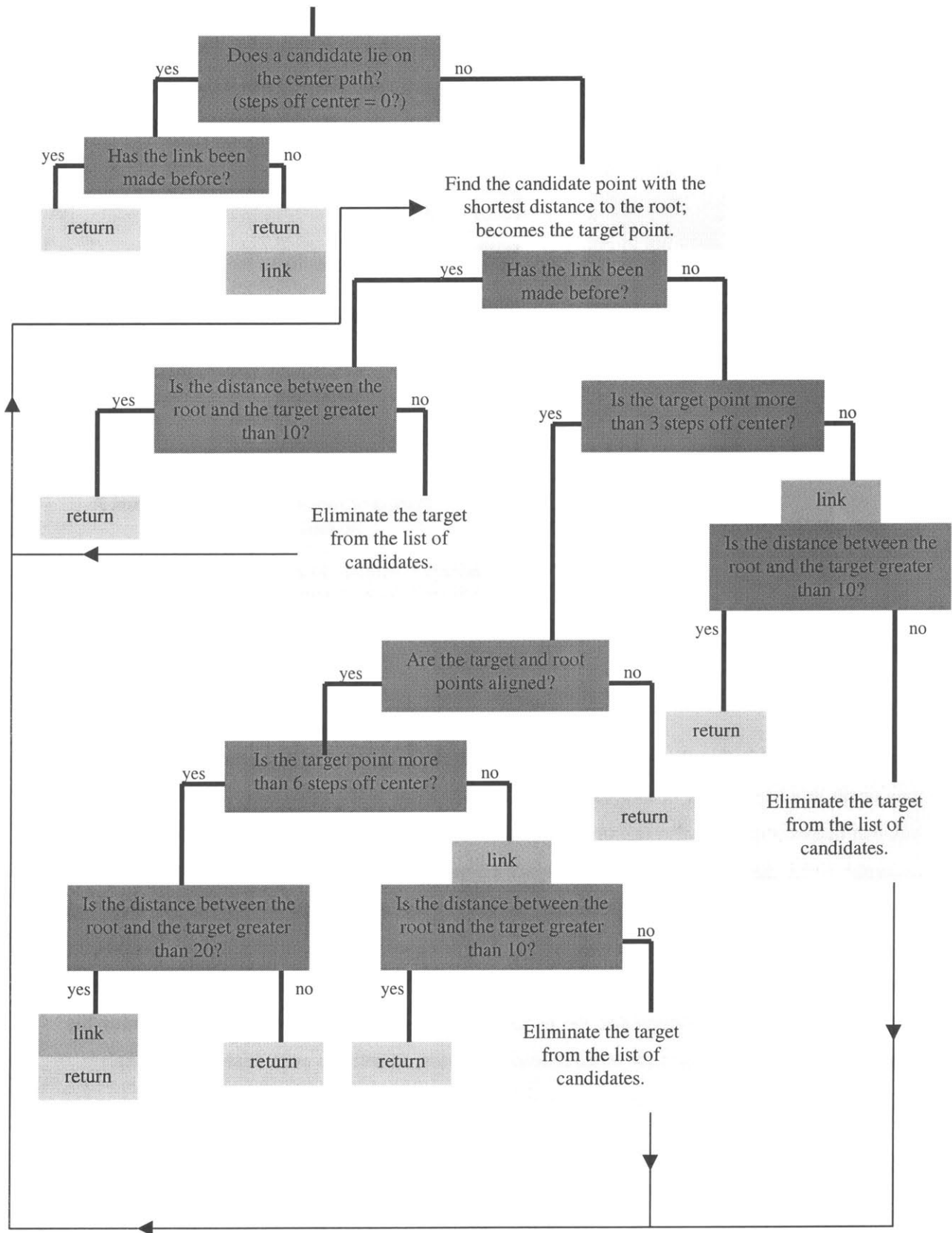


Figure 4.5. Outline of edge linking process

a point in zone four is linked if it has the shortest distance to the root, is aligned with the root and is more than 20 pixel units from the root. These linking rules are illustrated in Figure 4.6.

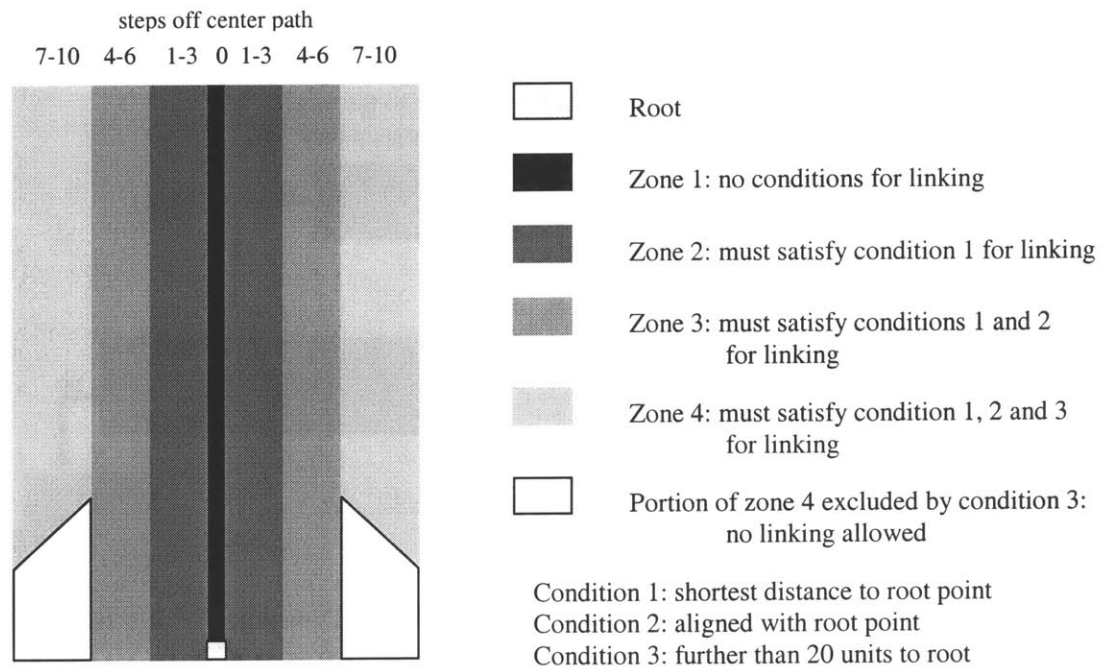


Figure 4.6. Zones for edge linking

It is possible that the root point should be linked to more than one candidate point. This may occur at a triple point or intersection. Often when this is the case, there is only one longer link that makes sense, but also one or more shorter links. In order for the program to make all the necessary links, the process described in Figure 4.5 must be repeated for the same root point. The program tends to find the shortest links first, because it chooses the closest point outside zone 1 as the target point. If a link is made that is less than or equal to a length of 10 units, the process repeats to find other short links and perhaps a longer link. However, once a link longer than 10 units is made, the process stops and the program moves to the next end point. Continuing the linking after a longer link is made tends to lead to useless, sometimes even misleading, links.

4.4 Conclusion

This new linking method is tailored for filling gaps in road edges. It takes advantage of the assumption that an edge is a part of a road and is not changing direction very quickly. It is a

fairly simple method that does a good job of estimating the edge directions. This in turn leads to the determination of a suitable search area. The maximum length and width of the area are fixed by the user for a specific image, so the system can potentially handle any edge gap size. In addition, multiple links with a single end point are possible, which certainly improves the performance of the method.

Due to the simplicity of the linking procedure, it cannot take into account every case of a link between two edge points. As was mentioned at the beginning of Section 3.2.1, requiring that a link to be between two end points ignores the fact that other edge points may provide a better starting point for linking. The example stated in that section explains how a branch in the road may be missed or linked incorrectly. However, it was also stated that these cases are rare, and in many cases the links that *are* made represent the branch structure well enough for the later steps of the system. In addition, although it is possible to figure out which edge points, other than the end points, should be beginnings of links, it requires a large increase in complexity and steps which do not work coherently with the rest of this linking procedure.

Another simplification which may limit the accuracy of the linking is that links are made with straight lines. This is done because it usually produces satisfactory results. If the link is short, there is little difference between a curved linking line and a straight one. Even if the link is long, in most cases the link should be either straight or only slightly curved, so the straight line suffices. It is only in the case of a longer link where the road is curving significantly that the straight line is not acceptable. As opposed to the issue raised in the previous paragraph, this can be addressed more easily. Although it has not been implemented yet, it should be possible to use the directions of the edges at both ends of the link to decide on a better path for the linking.

Overall, some edge gaps may be missed and some false links may be made. However, edge linking is not the final step in this system and a complete edge image is not the final goal. In this case, linking is another step in the process which helps improve the quality of the edge image for further processing. Provided that enough edges are kept in the edge detection stage, this linking process will fill most of the gaps which should be filled.

Chapter 5.

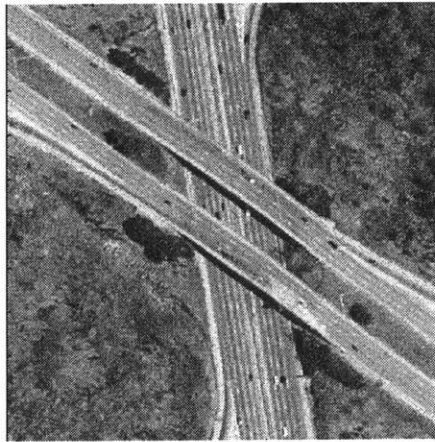
RESULTS AND DISCUSSION

5.1 Introduction

In the preceding chapters, two edge detection methods and one edge linking procedure are described in detail. Each of the edge detectors is followed by edge linking to produce a version of the edge-based road detection system. This chapter tests these two versions of the system and discusses and compares the results. Section 5.2 examines the results from the system that uses the Nevatia-Babu edge detector (NBED). Section 5.3 looks at the corresponding results from using the version that uses the wavelet edge detector (WED). These two sections are meant to illustrate the types of outputs that each version of the system produces. Section 5.4 focuses on comparing and contrasting the results of the two systems. Section 5.5 ends the chapter by discussing the overall conclusions on the performance of the systems based on the preliminary goals of the paper stated in the introductory chapter.

Three images are used for testing the system. They are shown in Figure 5.1. These images, all 500x500 pixels, are three of the four shown in Figure 1.1. As explained in the introductory chapter, each image presents some typical image characteristics which may prove to be difficult to handle in the road detection process. Figure 5.1 is a segment of the ‘interchange’ image, shown in Figure 2.3 (a) and used throughout this work as a test image. Figure 5.1 (a) is referred to as test image #1. Possible difficulties with this image include the lane markings and cars on the roads, as well as the texture of the surrounding foliage. Despite these possible trouble characteristics, this image should produce good results because of its sharp, relatively straight

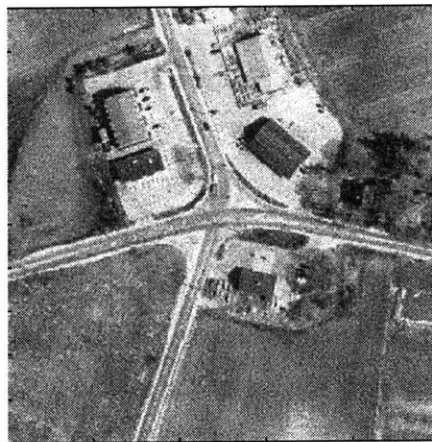
road edges and the close-to-uniform gray scale of the roads. Because of these positive properties, this image is used extensively for testing. Sections 5.2 and 5.3 use this image exclusively to detail the results of the two systems.



(a)



(b)



(c)

Figure 5.1. Original test images: (a) test image #1; (b) test image #2; (c) test image #3

Test images #2 and #3 are shown in Figures 5.1 (b) and (c) respectively. Test image #2 contains a grid-like residential area. Although the road edges are very straight and quite complete, the building edges are also strong and may cause some problems with road linking. The third test image shows a more rural crossroad. Here, the dirt along the road sides and in the parking lots is very close in gray scale to the roads and could potentially cause problems. Both images are used in Section 5.4, along with test image #1, to compare the results of the two systems.

5.2 Results from using Nevatia-Babu edge detector followed by linking

The results from running our implementation of the NBED on test image #1 are shown in Figure 5.2. Recall from Chapter 2 that this edge detection method is based on thresholding the edge values of the image pixels, where the edge value of a pixel represents how strongly it belongs to an edge. The images shown in Figure 5.2 are binary images which result from setting the pixels with the highest edge values to one (white), and all other to zero (black). Figure 5.2 (a) shows the complete resulting image, which is 500x500 pixels. Figure 5.2 (b) displays a 100x100 block of the full result so as to more clearly illustrate the nature of the edges which are found.

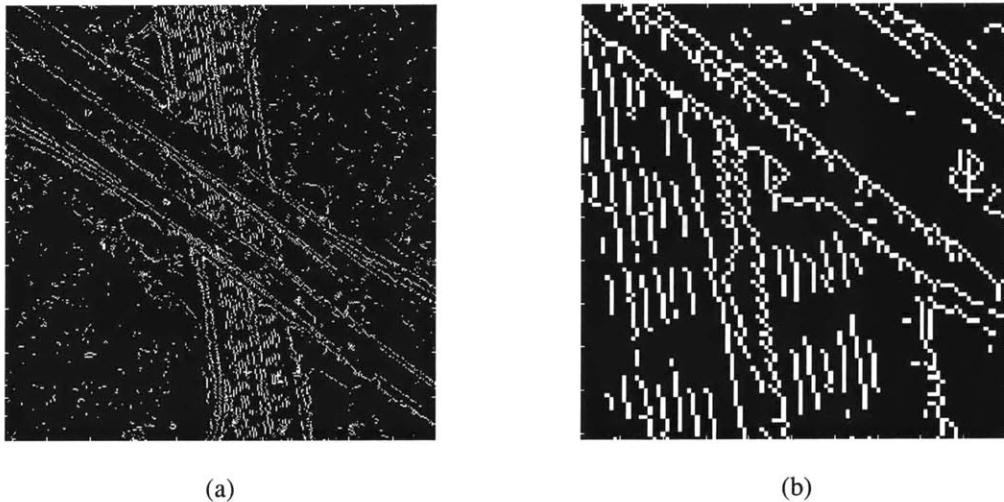


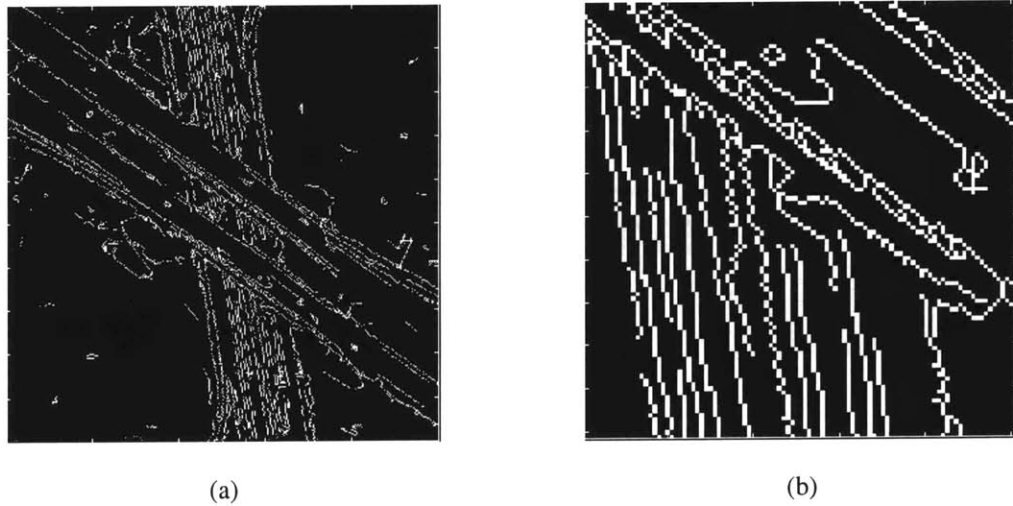
Figure 5.2. Test image #1 after Nevatia-Babu edge detection (NBED):
(a) complete image; (b) zoom of 100x100 block of image

Although not all the precise details of the edges are apparent in the full image (a), the overall trend is quite obvious. A human eye can clearly see the outline of the roads in this edge image. This is a direct result of the sharp road edges. However, this image also contains quite a bit of noise, that is undesirable, non-road edges. These are a direct result of the lane lines, the cars and the trees, as mentioned above. The zoomed image (b) shows parts of both the vertical and horizontal roads. Many of the lanes edges have been found in the vertical road. This image makes clear that this implementation of the NBED extracts a large number of edges, some which are unwanted.

The algorithm could be adjusted to find less edges by thresholding the edge values at a higher level. Tests were run which varied the value of the threshold. It was found that a higher threshold indeed decreased the number of edges, but many of the edges which were eliminated

were important road pieces. The threshold used in these tests, which was based on energy considerations, proved to provide the best balance of road and non-road edges.

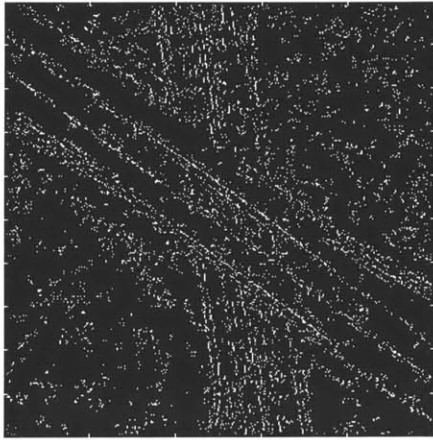
Figure 5.3 illustrates the results after linking the edges from Figure 5.2. Again, both the full result and a 100x100 block of the result are shown. The most obvious change evident from the complete image in Figure 5.3 (a) is the deletion of many of the non-road, noisy edges which are present after only edge detection. Recall that the edge linking procedure includes a step which eliminates short edges. Thus, the post-linking image is much cleaner than pre-linking. The zoomed image in Figure 5.3 (b) also indicates that several gaps are filled by the linking process. Although some gaps remain, many of the breaks in road edges evident after edge detection have been connected, which also contributes to the cleaner look of the result. In addition, it should be noted that few unreasonable links are made overall.



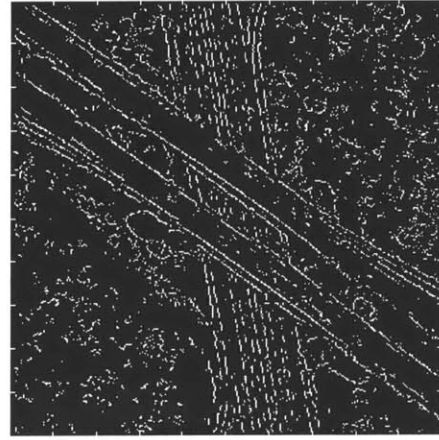
*Figure 5.3. Test image #1 after NBED and linking:
(a) complete image; (b) zoom of 100x100 block of image*

5.3 Results using wavelet edge detector followed by linking

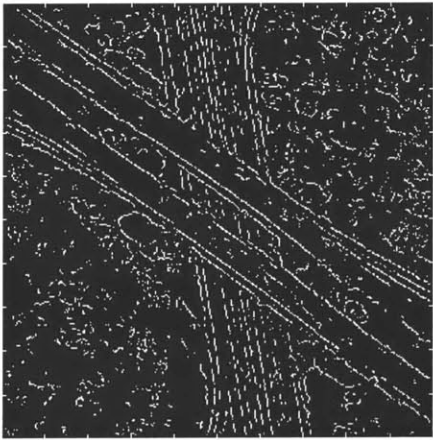
When the NBED is applied to a test image, a single edge image results. In contrast, when the WED is used, there is an edge image for each level of decomposition. Figure 5.4 shows the results of running the WED on test image #1 for five levels of decomposition. All five images



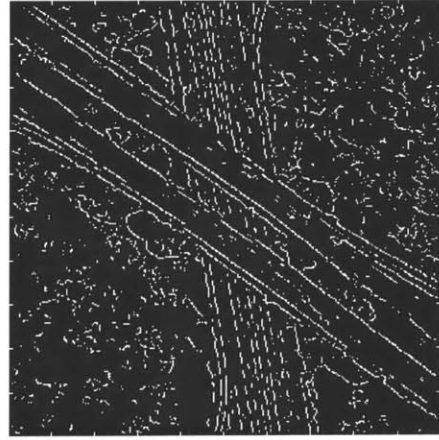
(a)



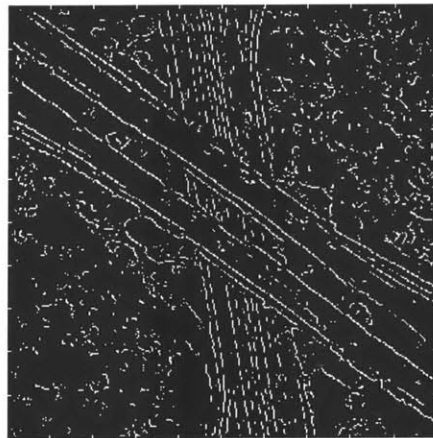
(b)



(c)

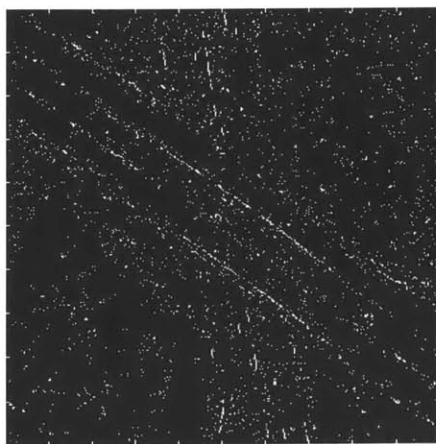


(d)

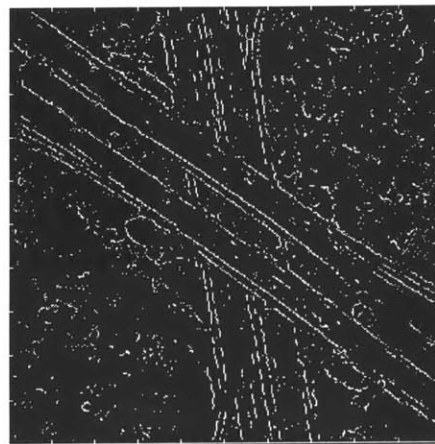


(e)

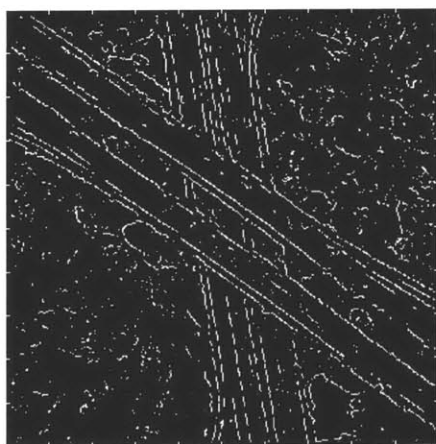
*Figure 5.4. Test image #1 after wavelet edge detection (WED), five levels of decomposition:
(a) level 1; (b) level 2; (c) level 3; (d) level 4; (e) level 5*



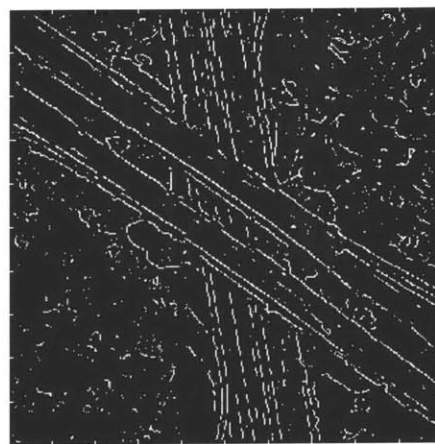
(a)



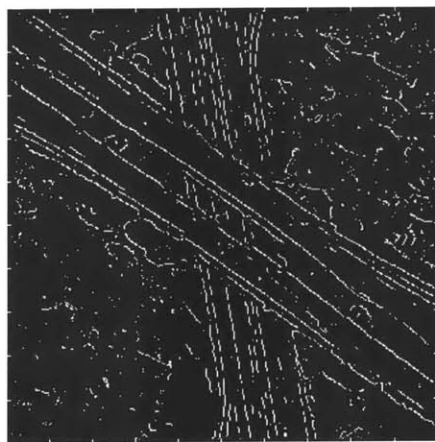
(b)



(c)



(d)



(e)

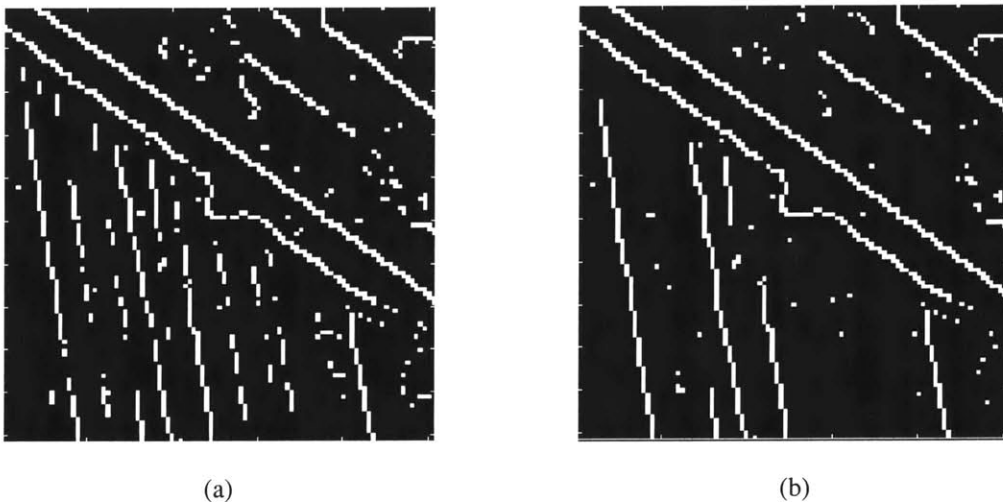
*Figure 5.5. Test image #1 after WED and linking, five levels of decomposition:
(a) level 1; (b) level 2; (c) level 3; (d) level 4; (e) level 5*

are 500 x 500. They are also binary images, formed by thresholding the wavelet coefficients at each scale. These edge images are not drastically different than that produced by the NBED. This figure is meant to show the trend across levels rather than the edge details.

Recall that the wavelets used to do the decomposition double in width at every level. Therefore, certain levels should have better results than others. At these levels, the width of the wavelet will correspond closely to the width of the features. From Figure 5.4, it can be seen that the finest level, level 1, does not work well for detecting the roads in this image. Figure 5.4 (a) is visibly inferior to the other results. The other four levels, although slightly different, are approximately the same quality. That is, they have about the same amounts of noise contents, and they have nearly the same numbers of completed road edges. The number of edges, both road and non-road edges, decreases slightly as the level increases, but there is very little variation among Figures 5.4 (b), (c), (d) and (e).

A similar pattern can be seen after the edge linking procedure is performed. Figure 5.5 indicates that while the outputs at levels 2-5 have improved slightly as a result of linking, the output at level 1 has deteriorated. It is very hard to discern any roads at all in Figure 5.5 (a). The other four images, Figures 5.5 (b), (c), (d) and (e) remain very similar. The five images are shown more to relay the fact that each wavelet decomposition level produces an output image rather than to illustrate drastic differences between the levels.

Overall, it appears as if the linking method in this case does a worse job of eliminating the noisy edges. Actually, much of this noise is isolated single points, which should be eliminated in the edge detection stage, not the linking stage. The WED does not include a step to



*Figure 5.6. Before and after linking: (a) zoom of Figure 5.4 (c), before linking;
(b) zoom of Figure 5.5 (c), after linking*

delete single points, whereas the NBED does. In reality, the linking algorithm applied to this image does about as well after the WED as after the NBED. This is more clearly seen by returning to the zoomed, 100x100 blocks of the above images. Figure 5.6 (a) and (b) show close-ups of Figure 5.4 (c) and Figure 5.5 (c) respectively. That is, these images show the before and after linking results at level 3. As with the linking after the NBED, linking here manages to fill several gaps in road edges, without making any false links. The next section compares results for both systems, and it then becomes more evident as to which method performs better.

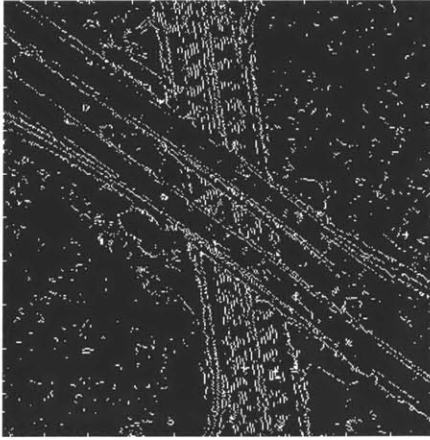
A note here about the multiple level output of the WED: With some images, the variability in the quality of the edge images outputted at each level is larger than it is above. That is, sometimes a few levels clearly give superior results, and each of these levels gives different information about the edges. In this case, it may be that a combination of edge images from several levels would yield the best result. Combining information across decomposition levels is not trivial. There are various ways to do it, and many have been tried as part of this work. None were found to be satisfactory, but this does not mean it cannot be done successfully. Although the levels are seen as being separate results in this project, it should be kept in mind when using wavelets for edge detection that it may be beneficial to combine the edges across levels.

5.4 Comparison of results

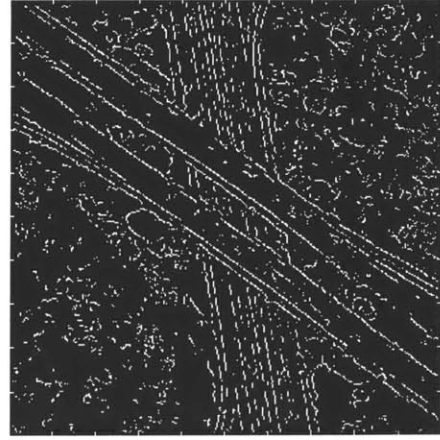
Now that the types of results which are produced by each method have been displayed, the focus changes to comparing and contrasting these results for the three test images from Figure 5.1. The resulting edge images are given after edge detection, and again after edge linking. For all tests using the WED, the level three results are shown. Although the edge linking procedure is the same following both the NBED and the WED, the combination of edge detection and linking can yield different results for each method.

Figure 5.7 shows the results after edge detection for test image #1. These images are all displayed previously in this chapter, but are exhibited here together to emphasize the differences between the outputs of either method. Both methods do a fair job of detecting the road edges. However, the NBED seems to detect more road edges, such as the lane lines in the vertical road, and less noisy edges, while the WED detects more noisy edges and less road edges. This is also evident from the zoomed images (c) and (d).

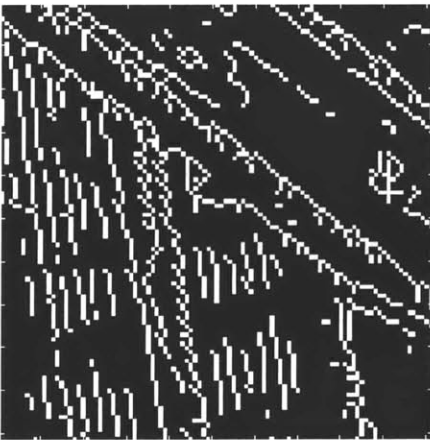
After edge linking, these differences become more obvious. Figure 5.8 shows a noticeable contrast in both the number of noisy, unwanted edges and the completed road edges.



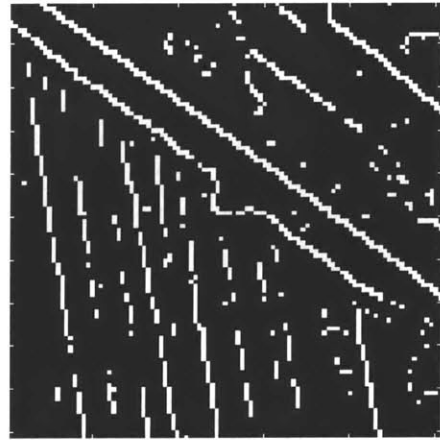
(a)



(b)



(c)



(d)

Figure 5.7. Comparison of edge detection results for test image #1: (a) using NBED, complete image; (b) using WED, complete image; (c) using NBED, zoom of 100x100 block of image; (d) using WED, zoom of 100x100 block of image

More short, non-road edges are eliminated and more links are made in Figures 5.8 (a) and (c) than in (b) and (d). This seems to favor the NBED method. However, concentrating on the vertical road, the WED gives a less complicated set of edges. Figure 5.8 (c) has many vertical edges very close together. This may be harder to sort out if further processing were to occur. Another difference which favors the WED is the “straightness” or “smoothness” of the edges themselves. The thinning algorithm that is a part of the NBED sometimes leads to thicker edges being thinned to straight edges which wobble back and forth every few pixels. The WED does not include this thinning step, so the edges tend to be smoother. This difference is more evident in the results from test images #2 and #3.

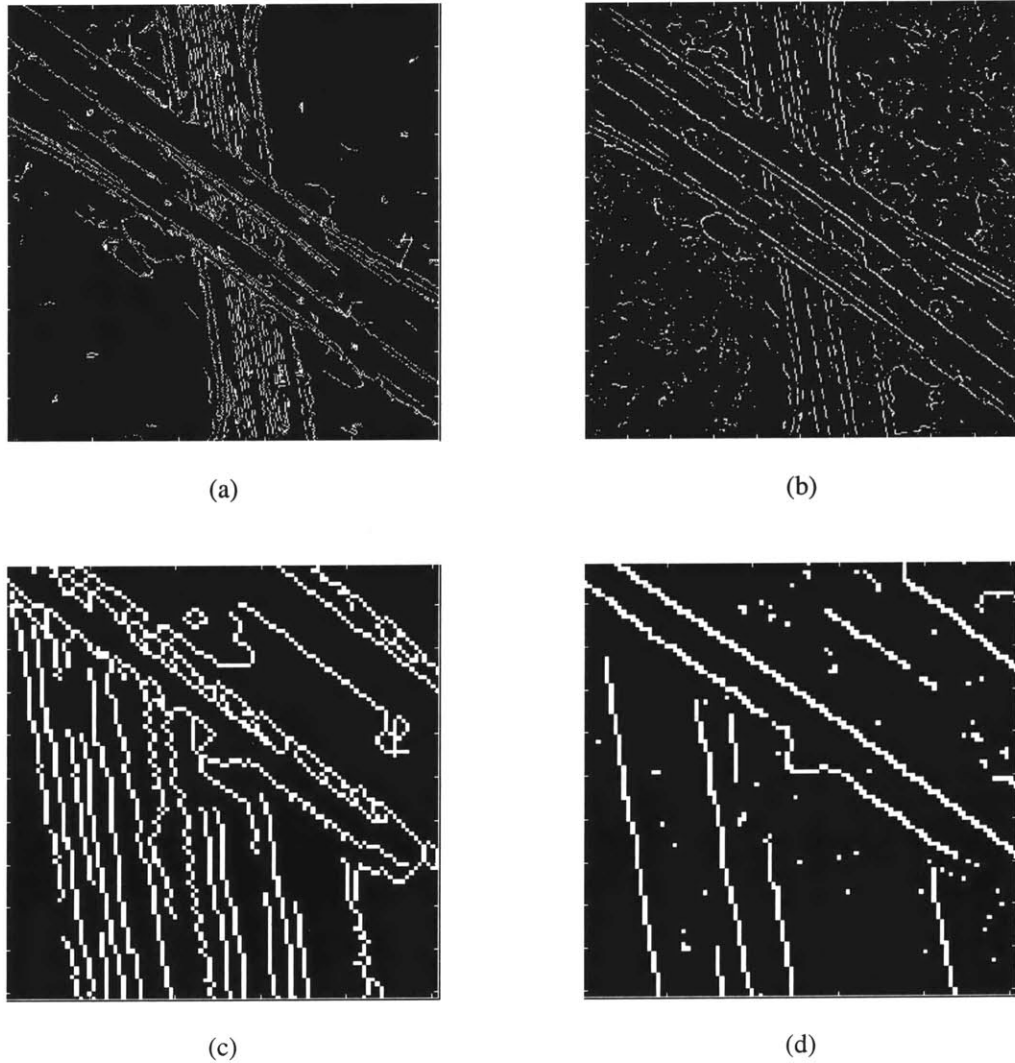
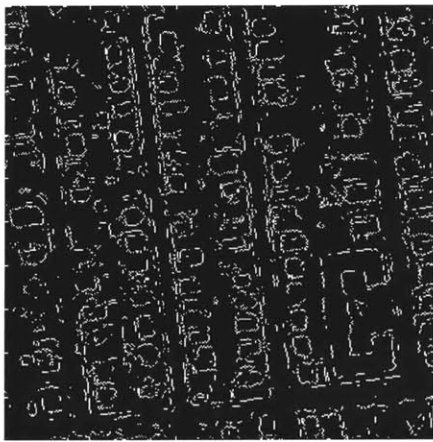
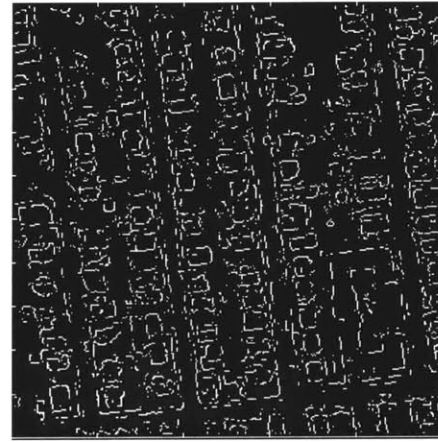


Figure 5.8. Comparison of edge linking results for test image #1: (a) using linking after NBED, complete image; (b) using linking after WED, complete image; (c) using linking after NBED, zoom of 100x100 block of image; (d) using linking after WED, zoom of 100x100 block of image

Test image #2 is the grid-like, urban street pattern. This image is full of houses and roads, with no foliage, so the potential for noise is much lower. Nonetheless, the buildings can still cause problems. The results after edge detection are shown in Figure 5.9. The two full 500x500 images look remarkably similar. Again, both methods do a fair job of finding road edges, however it is clear that the building edges are more prominent. The zoomed images in Figure 5.9 (c) and (d) better illustrate the differences in the two methods with respect to test image #2. The NBED again finds more road edges, and eliminates more single points and short edges. The WED again displays smoother edges in contrast to the more jagged edges in Figure 5.9 (c).



(a)



(b)



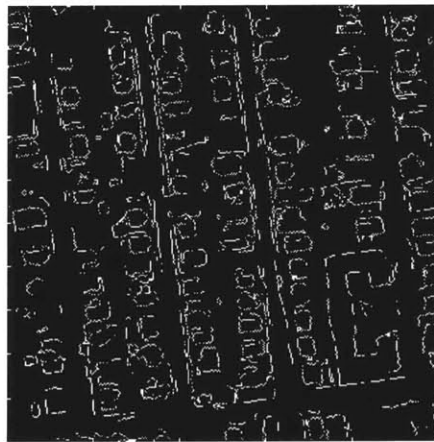
(c)



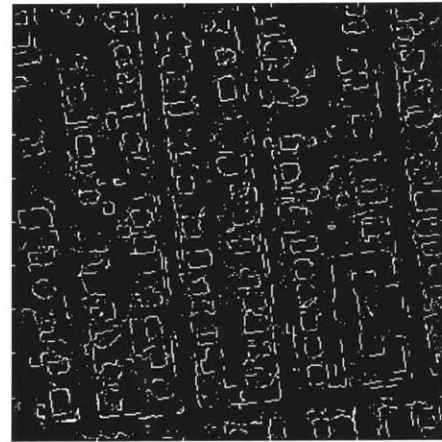
(d)

Figure 5.9. Comparison of edge detection results for test image #2: (a) using NBED, complete image; (b) using WED, complete image, (c) using NBED, zoom of 100x100 block of image; (d) using WED, zoom of 100x100 block of image

Figure 5.10 shows the results after edge linking. Most of the linking occurs in the building edges. This is because of the lack of large road edge pieces provided by the edge detection methods. The roads appear to have strong edges in the original test image, but it seems these edges are not sharp enough to be detected very well by either the NBED or the WED. The edges on the upper parts of the two middle vertical roads show are the most completed edges in the output images. This area is highlighted in the zoomed images (c) and (d). Here the main difference between methods is that the NBED and linking managed to complete the inner road edges on both sides of the road, while the WED did not have enough edge pieces to link these segments and deleted them instead. Both methods perform about equally well, with neither of them doing a good job at road detection in this case.



(a)



(b)



(c)



(d)

Figure 5.10. Comparison of edge linking results for test image #2: (a) using linking after NBED, complete image; (b) using linking after WED, complete image; (c) using linking after NBED, zoom of 100x100 block of image; (d) using linking after WED, zoom of 100x100 block of image

The final tests are run on test image #3. The results after edge detection and linking are shown in Figures 5.11 and 5.12 respectively. The differences in the two methods visible from these images are identical to those discussed above. To summarize these differences:

- The NBED detects more road edges (as well as building edges and other major feature edges) than the WED.
- The WED detects more noisy, non-road edges than the NBED.
- The edges found by the WED are smoother than the correspond edges found by the NBED, which tend to be more jagged.
- The linking method tends to do perform a bit better with the NBED rather than the WED.



(a)



(b)



(c)



(d)

Figure 5.11. Comparison of edge detection results for test image #3: (a) using NBED, complete image; (b) using WED, complete image, (c) using NBED, zoom of 100x100 block of image; (d) using WED, zoom of 100x100 block of image

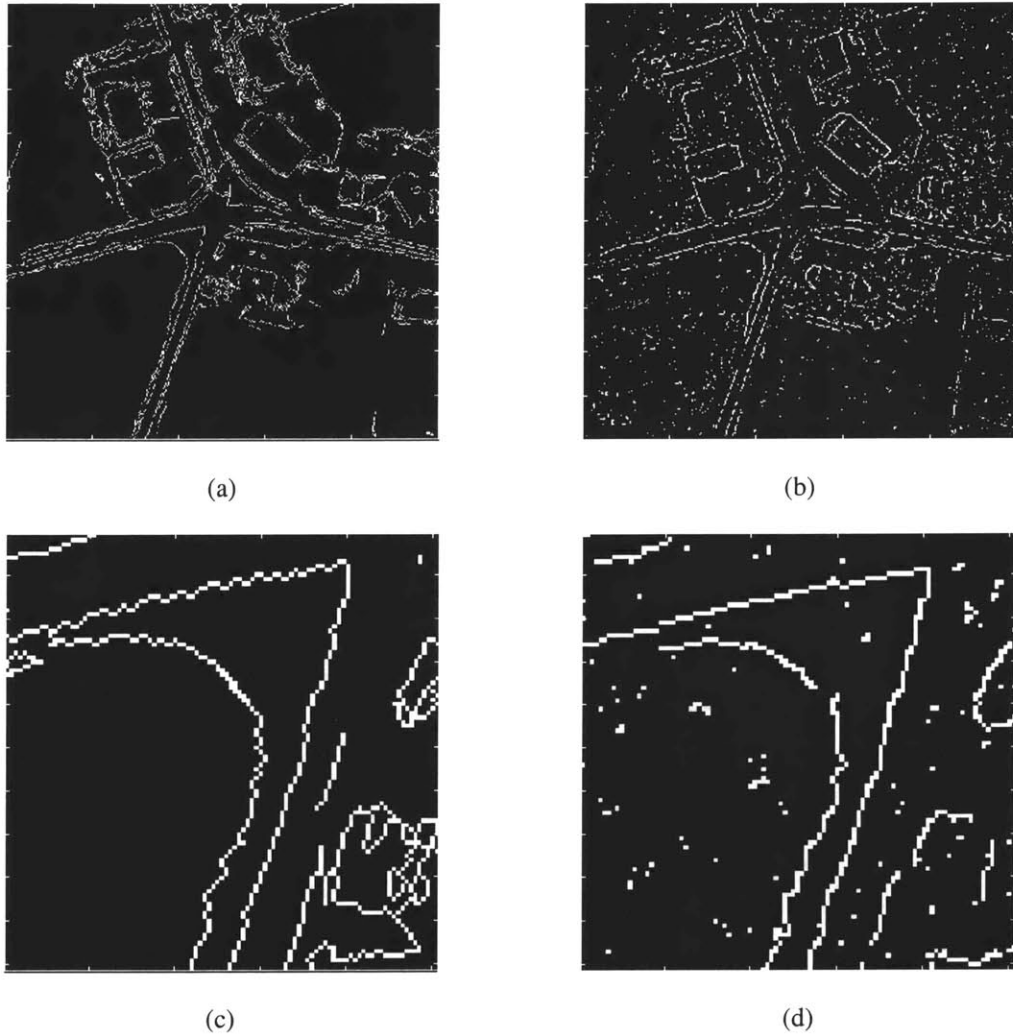


Figure 5.12. Comparison of edge linking results for test image #3: (a) using linking after NBED, complete image; (b) using linking after WED, complete image; (c) using linking after NBED, zoom of 100x100 block of image; (d) using linking after WED, zoom of 100x100 block of image

5.5 Conclusions

Overall, both systems do a satisfactory job of detecting road edges. The first goal of this work is to develop a simple system for road detection in aerial images, which is done, and to test how well it works. The NBDE followed by edge linking performed slightly better than the WED with edge linking. This is logical because the edge linking algorithm was originally developed using the NBDE as the preceding edge detection step. However, despite the fact that when applied to test images #1, #2 and #3 the NBDE method with linking led to fairly clean results, the output would not suffice for most applications. Yet recall that this system is meant to perform

edge *finding*, which is normally followed by edge *tracing*. In this context, the results produced by the NBDE with edge linking may suffice. That is, in many test runs enough road edges are found and linked, and enough non-road edges are eliminated, for the next road tracing stage to have a good set of road seeds to work with. Of course the sufficiency of the results depends on the road tracing algorithm and the final goal of the complete system. These topics are beyond the scope of this project. It is stated only that in many cases the output images obtained from our system are satisfactory road finding results.

One further consideration in how well the system performs is the quality of the aerial images and the characteristics of their content. All test images used in this project are of high quality and they have properties which lend themselves to good road detection results. The images are all taken from the same source, are high resolution and have little noise content. In addition, they generally have sharp boundaries between road and background, and few occlusions blocking road edges from view. In most real road detection applications, the quality of images will be much lower. If the task is to automatically map a certain area, the program cannot skip parts of the area with trees blocking the road edges or with blurred or undefined boundaries. From a few tests with lower quality images, it is found that the results of the system are much worse when the quality of the aerial image is lower. Our road detection program performs well for images with sharp, continuous edges, but once discontinuities in the edges are introduced the quality of the results declines. This suggests that in some cases, further processing would need to be performed in order to extract meaningful information to pass to a road tracing algorithm. Some possible extensions and improvements to the system are discussed in Chapter 6.

The second goal of this work is to experiment with wavelets for edge detection. The results of applying the WED to the test images are not exactly as anticipated. Although many road edges are found by this detection method, it was expected that the edge images at the different levels would vary more than they do, and specifically that one or two levels would have very good results. In our case, most of the levels give similar, but good, results. The wavelets produce images comparable to the NBED. Of course, improvements can be made. Further work with this experiment could include testing various other wavelets. In addition, more work could be done to try to combine edge information across levels, as mentioned above.

Designing and implementing a new linking method is the third goal of this paper. Judging from the results in this chapter, the zone-based algorithm which is developed in Chapter 4 leads to fairly successful results, for a relatively non-complex procedure. In tests using the images from this chapter and other test images not displayed in this paper, the linking method is found to often fill many of the small to medium gaps in the road edges, and to rarely make false

links. Further work in this area may include the designation of a good measure by which to precisely determine how well this linking method performs as compared to others.

Chapter 6.

FUTURE WORK AND CONCLUSION

6.1 Possible extensions to the system

From the results shown in the last chapter, it is clear that edge detection and linking can only go so far in extracting the road edges from the test images. These two steps certainly help to locate the road edges, but the results are by no means ideal. There remain gaps in the road edges, and non-road edges are also present in the results. Thus, there is room for improvement. This chapter discusses two extensions to the current system which may improve the success of the procedure in locating the roads. One is an edge pairing step which follows the edge detection and linking steps. This algorithm has been partially implemented for a few test cases and results are shown and discussed below. The second extension is a hypothetical one, involving the change of the automatic system to a semi-automatic one. In addition, this chapter discusses the possibility of using these steps to extract features other than roads, such as buildings.

6.1.1 Edge pairing

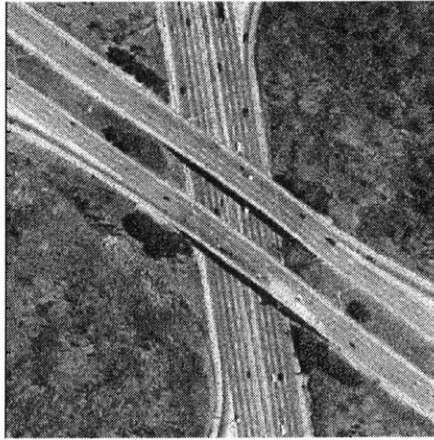
As mentioned above, there are two major problems with the results which make them not completely satisfactory: large gaps and remaining unwanted edges. The edge pairing step addresses the second problem. The edge detection and linking methods work to find road edges and join them together. However they do little to eliminate edges which are found by the edge detection method and but do not correspond to roads. The goal of the edge pairing step is to

focus on which of the remaining edges *do* correspond to roads and more precisely locate where those roads are within the image.

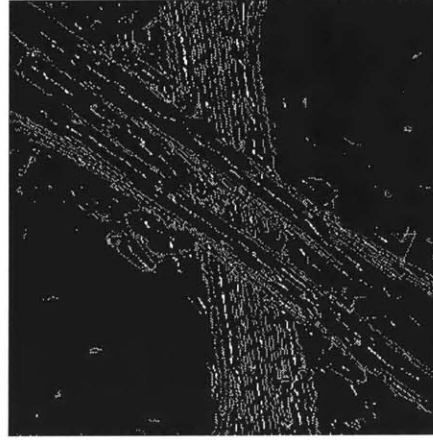
One characteristic that applies to all roads is that they are bounded by two parallel or very close to parallel sides. Even when occlusions, shadows or other details break the continuity of the road boundaries, enough pieces usually remain after edge detection and linking to be able to pair together edges which represent the two sides of a road. Many of the local methods for road detection include a step in which pairing of parallel lines occurs. After the edges are extracted in one way or another, they are usually approximated by linear segments. The straight lines which represent the edges also have orientations, determined somewhere during the preceding steps of the system. Sophisticated algorithms have been developed to find the best sets of pairings between anti-parallel lines. For example, Scher, Shneier and Rosenfeld use an iterative method which assigns scores to pairings based on lengths and overlaps of edges.[20] Zhu and Yeh expand on this method, including tests for the intensity and width between two edges.[23]

In the present work, a simplified version of these methods is implemented. It closely resembles a step of the RoadF algorithm developed by Zlotnick and Carnine in [25]. They pair roughly parallel edges together and then plot the midpoint between these two edges, later using sets of these midpoints as seeds for growing the entire road network. Our simple pairing method borrows from this idea by plotting the points halfway between paired edges. The user inputs the minimum and maximum widths of the roads to be found. At each edge point, the local direction of the edge is found. Then a search is done along the line of pixels normal to this local direction for other edge points that are between the minimum and maximum distances away. If another edge point is found, the local direction of that edge is also found and compared to the local direction of the original edge point. If they differ by less than a small threshold, then the two edges are locally parallel or close to parallel, and they are paired. In this case, a point is drawn half way between the two edges and a record is made of the pairing. The output from performing this procedure on all edge points is an image showing the linked edges with points located half way between paired edges.

This pairing algorithm is not run on all the test images, but is applied to a few images to get some idea of the results it provides. Figure 6.1 shows a sampling of those results, using the NBED before the pairing algorithm. The road centers are seen as the bright white points. Clearly, the success of the pairing step depends highly on the success of the previous detection and linking steps. In areas of the images where the road edges are more complete, the pairing algorithm does quite well. These areas include most of the horizontal and vertical roads in (a)



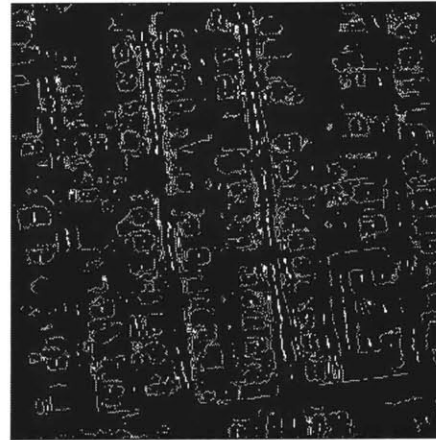
(a)



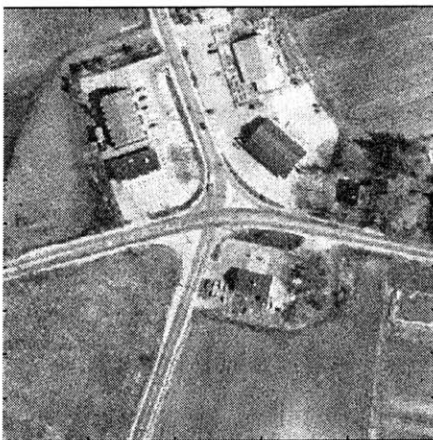
(b)



(c)



(d)



(e)



(f)

Figure 6.1 Results of the edge pairing algorithm on some test images

and (b), as well as the third vertical road in (c) and (d), and the top part of the vertical road in (e) and (f). Obviously, there will be no pairing where there are road gaps. The shortfall of this method is that it does not give any new information about the road networks.

However, this method does do a good job of eliminating non-road edges, which was the goal of using this procedure. Most of the center points which are found belong to roads. Therefore, considering that the next step in a road detection system would be doing road tracing, it may be possible to get better road detection results from doing further processing on only the center points and disregarding the edge points. That is, it may be possible to perform some type of linking or tracing on the plotted midpoints in a way similar to the seed growing in [25]. It is this elimination of non-road edges that may make the edge pairing a useful part of a complete road detection system.

6.1.2 Automatic vs. semi-automatic systems

Many papers written about road extraction methods discuss the pros and cons of both the automatic and semi-automatic approaches. Automatic methods have no human component. As is envisioned for the system described in previous chapters, the computer determines all input values and parameters from analyzing the original image. In contrast, semi-automatic procedures involve human interaction with the computer. For example, a common way to incorporate the human into the process is to have them enter a starting point and direction for a given road edge. The computer uses the human's input to initialize its own tracing of the edges. Human's can also be used to stop the process if it is headed off-track, or to help it if it gets stuck somewhere during the tracing.

There are advantages and disadvantages to both approaches, depending on the goal of the system. If the goal is to eliminate the human operator altogether, in order to be able to run the process quickly and without supervision, then only an automatic method will do. However, if the goal is to get very precise, accurate results and to speed up the process somewhat, a semi-automatic system may be better. Humans have the ability to very easily find the roads in a wide variety of aerial images, despite occlusions, shadows or other inconsistencies. If a computer is to find the roads as accurately as a human observer, extremely complex algorithms are required to model all these varying characteristics. Therefore, by combining the precision of the human eye with the calculation speed of a computer, it may be possible to get more accurate results and still improve the procedure time over a completely manual system.

The edge detection and linking steps described in the previous chapters, and even the edge pairing step from the last section, can all be performed automatically with no human interaction. Nevertheless, it may improve the results if the method were to become semi-automatic by including some human input. For instance, perhaps the operator could use the mouse to click on a point somewhere inside a road. From this, the computer could determine the texture and/or gray scale of that particular road. Then other road points could be found by comparing texture and/or gray scale information. As another possible way to include a human, the computer could show doubtful edges or links to the human to check if they should be included.

By extending the current system in ways such as these and making the method semi-automatic, results could almost certainly be improved. Having a human operator to help guide the system is an effective way of handling the difficulties of the road detection problem which arise from the variations in image characteristics. Furthermore, these changes can be implemented on top of the current system, so the steps taken by the computer remain simple and fast.

6.1.3 Detection of features other than roads

As has been demonstrated throughout this paper, the individual steps of this system have been designed, where possible, to specifically extract road edges. The Nevatia-Babu edge detector and the edge linking procedure especially focus on finding road features. This tailoring to a specific feature type is done by introducing certain feature characteristics. In the case of road features, it is assumed that roads are long, with low curvature, and with parallel or close to parallel boundaries. More subtly, it is also assumed that edges represent a sharp boundary between two largely different gray scales. This last assumption, unlike the first three, is not dependent on the feature being extracted. Of the three which are dependent on the feature, the first two are built into the edge linking method and the third is used in the pairing algorithm described earlier in this chapter. Although the combination of all three does a good job of describing the main characteristics of roads, they are not unique to the road feature. For instance, railroads are also long, straight and bounded by two parallel edges. Rivers tend to be long and bounded by two edges. Buildings have straight edges and in general are bounded by two pairs of parallel sides. Therefore, it is reasonable to consider applying the system for the extraction of features other than roads.

It is expected that only small changes would need to be made to accommodate the detection of other features. Returning to the example of building extraction, most buildings have a basic rectangular shape. From this assumption, it is known that a building will have corners at the meeting points of these sides, and that the sides should be in pairs of parallel edges. Also, in general these edges will be much shorter than those considered in the road case. So the linking algorithm could be changed to favor shorter edges, with a component added to take into account the importance of the corners. This could be followed by a pairing algorithm that looks for two or more pairs of parallel lines with shared corners. Although these changes require more than one or two lines of code, they could easily fit within this system.

Similar types of changes could conceivably be made to take into account multiple other features, such as the aforementioned rivers and railroads. Although roads and buildings tend to be the features which are most often extracted from aerial images, other features are important for certain applications. For instance, an automatic map generation application would need to find roads, rivers, lakes, railroads, etc. It seems plausible that an extraction system for any of these feature types could be based on the edge-based steps described in this paper. Thus, the system is not only quite simple, but appears to be extendible to other types of feature detection as well.

6.2 Conclusion

Throughout this work, a simple edge-based road extraction method is developed. The system has edge detection as its first step, using either the Nevatia-Babu edge detector or the wavelet edge detection. The second step of the method involves a new edge linking procedure centered around linking edge pieces by continuing them in the direction they are headed. The details of each of these steps are outlined in the preceding chapters. The final section in Chapter 5 addresses the specifics of how well the system generally works and how well the goals of this paper are met.

This work is concluded by returning to the largest difficulty in the road detection process: the variation in image characteristics from image to image. Although in general roads do have certain characteristics in common, such as length, curvature and parallel boundaries, individual road properties vary quite widely across all aerial images. This project attempts to focus on the common road characteristics and to see if the variations can be generalized or even ignored. The results of running the system on many aerial images, including several not shown in this work, lead to the conclusion that the variations must be accounted for somewhere in the process in order

to achieve excellent results. Of course this adds a significant amount of complexity to the procedures. It is unclear whether this complexity should be built directly into the road finding algorithm developed here, or into a road tracing procedure which would be expected to follow road finding. However, it is clear that the variation in aerial image characteristics is the largest obstacle to developing simple road detection techniques. It seems that simple steps can form the basis for a road detection system, but that more complex steps must be incorporated in order to deal with the variety of aerial images that need to be processed by most applications.

BIBLIOGRAPHY

- [1] S. Airault, R. Ruskone, O. Jamet, "Road detection from aerial images: a cooperation between local and global methods", *Proc. SPIE Image and Signal Processing for Remote Sensing*, SPIE Vol. 2315, pp. 508-518, 1995.
- [2] M. Barzohar, D. B. Cooper, "Automatic finding of main roads in aerial images by using geometric-stochastic models and estimation", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. 18, No. 7, pp. 707-721, July 1996.
- [3] J. Canny, "A computational approach to edge detection", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. 8, No. 6, pp. 679-698, Nov. 1986.
- [4] *I. Daubechies, *Ten Lectures on Wavelets*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1992.
- [5] A. Farag, E. J. Delp, "Edge linking by sequential search", *Pattern Recognition*, Vol. 28, No. 5, pp. 611-633, May 1995.
- [6] M. Fischler, J. Tenenbaum, H. Wolf, "Detection of roads and linear structures in low-resolution aerial imagery using a multisource knowledge integration technique", *Computer Graphics and Image Processing*, Vol. 15, No. 3, pp. 201-223, March 1981.
- [7] D. Geman, S. Geman, C. Graffigne, P. Dong, "Boundary detection by constrained optimization", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. 12, No. 7, pp. 609-628, July 1990.
- [8] R. C. Gonzalez, *Digital Image Processing*, 2nd Edition. Reading, MA: Addison-Wesley Publishing Co., 1987.
- [9] *Automatic Extraction of Man-Made Objects from Aerial and Space Images*. Edited by A. Gruen, O. Kuebler, P. Agouris. Basel, Boston: Birkhauser Verlag, 1995.
- [10] H. Hajj, T. Nguyen, R. Chun, "A 2-D multirate Bayesian framework for multiscale feature detection", *SPIE Conference on Wavelet Applications in Signal and Image Processing IV*, SPIE Vol. 2825, pp. 330-341, 1996.
- [11] C. Heipke, A. Englisch, T. Speer, S. Stier, R. Kutka, "Semi-automatic extraction of roads from aerial images", *Proc. SPIE ISPRS Commission III Symposium*, SPIE Vol. 2357, pp. 353-360, 1994.
- [12] *B. Hubbard, *The World According to Wavelets: the Story of a Mathematical Technique in the Making*, 2nd Edition. Wellesley, MA: A. K. Peters, 1998.
- [13] Johnson II, C.-C. Li, "Fuzzy thresholding and linking for wavelet-based edge detection in images", *Proc. SPIE Conference on Applications of Soft Computing*, SPIE Vol 3165, pp319-329, 1997.
- [14] *G. Kaiser, *A Friendly Guide to Wavelets*. Boston: Birkhauser, 1994.

- [15] T. Law, H. Itoh, H. Seki, "Image filtering, edge detection, and edge tracing using fuzzy reasoning", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. 18, No. 5, pp. 481-491, May 1996.
- [16] S. Mallat, "A theory for multiresolution signal decomposition: a wavelet representation", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. 11, No. 7, pp. 674-692, July 1989.
- [17] S. Mallat, S. Zhong, "Characterization of signals from multiscale edges", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. 14, No. 7, pp. 710-732, July 1992.
- [18] R. Nevatia, K. R. Babu, "Linear feature extraction and description", *Computer Graphics and Image Processing*, Vol. 13, No. 3, pp. 257-269, July 1980.
- [19] J. Porrill, "Fitting ellipses and predicting confidence envelopes using a bias corrected Kalman filter", *Image and Vision Computing*, Vol. 8, No. 1, pp. 37-41, Feb. 1990.
- [20] A. Scher, M. Shneier, A. Rosenfeld, "A method for finding pairs of antiparallel straight lines", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. 4, No. 3, pp. 316-323, May 1982.
- [21] *G. Strang, T. Nguyen, *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.
- [22] *M. Vetterli, J. Kovacevic, *Wavelets and Subband Coding*. Englewood, NJ: Prentice Hall, 1995.
- [23] M.-L. Zhu, P.-S. Yeh, "Automatic road network detection on aerial photographs", *Proc. CVPR'86 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 34-40, 1986.
- [24] Q. Zhu, M. Payne, V. Riordan, "Edge linking by a directional potential function (DPF)", *Image and Vision Computing*, Vol. 14, No. 1, pp. 59-70, Feb. 1996.
- [25] A. Zlotnick, P. D. Carnine Jr., "Finding road seeds in aerial images", *CVGIP: Image Understanding*, Vol. 57, No. 2, pp. 243-260, March 1993.

APPENDIX A: MATLAB CODE

A.1. List of Programs

To run the road detection program using the Nevatia-Babu edge detector and edge linking, run 'mainnowave.m', which calls the following Matlab programs:

mainnowave.m:	99
• getimage.m	99
• getdirections.m	100
• findmax.m	101
• pickedgepts.m	102
• threshold.m	104
• smalllocalmax2.m	105
• rightdir.m	105
• singlepts.m	106
• checkforpts.m	106
• dothethin.m	107
• thin.m	108
• insidethin.m	109
• link.m	111
• endpts.m	116
• badendpts.m	117
• getbranch.m	118
• dirstep.m	119
• oppositedir.m	119
• searchforpts.m	120
• lastdirs.m	125
• dirstep.m	119
• oppositedir.m	119
• poll.m	126
• findmax.m	101
• getcombos.m	127
• findmin.m	130
• checklink.m	130
• takestep.m	134
• drawlink.m	135
• gostraight.m	143
• findmin.m	130
• dirdiff.m	144
• linepairs.m	147
• pairup.m	148
• takestep.m	132
• linepath.m	149
• gostraight.m	143

To run the road detection program using the wavelet edge detector and edge linking, run 'mainwave.m', which calls the following Matlab programs:

mainwave.m	157
• getimage.m	99
• getdirections.m	100
• findmax.m	101
• waveedgepts.m	158
• doconvs.m	159
• modphase.m	159
• localmax4.m	160
• grayvar.m	161
• link.m	111
• endpts.m	116
• badendpts.m	117
• getbranch.m	118
• dirstep.m	119
• oppositedir.m	119
• searchforpts.m	120
• lastdirs.m	125
• dirstep.m	119
• oppositedir.m	119
• poll.m	126
• findmax.m	101
• getcombos.m	127
• findmin.m	130
• checklink.m	130
• takestep.m	134
• drawlink.m	135
• gostraight.m	143
• findmin.m	130
• dirdiff.m	144
• linepairs.m	147
• pairup.m	148
• takestep.m	132
• linepath.m	149
• gostraight.m	143

A.2. Matlab Code

% mainnowave.m

```
% Serves as the main program when wavelets are not used.

% Get the aerial image:
getimage

% Get values from user that will be inputted into functions later:
T1 = input('Should the local maximum test be used? 1 for yes, 0 for no. ');
T2 = input('Should the angle test be used? 1 for yes, 0 for no. ');
if T2 == 1
    anglimit = input('For angle test: what limit on the difference in angles? ');
else
    anglimit = 0;
end
T3 = input('Should the threshold test be used? 1 for yes, 0 for no. ');
if T3 == 1
    p = input('For the threshold test: what percentage of edge energy to keep? ');
else
    p = 40;
end

%Input for linking edges - '
numdirs = input('How many directions should be polled to determine primary and secondary directions? ');
tooshort = input('How short should the shortest edge kept be? Any edge shorter than this will be deleted ');
maxgap = input('How long should the search area be? ');
maxsoc = input('How wide should the search area be? ');

%Input for pairing edges - '
dirstize = floor((input('How long an edge piece should be used to determine the local direction? '))/2);
minwidth = input('What is the minimum width of the roads? ');
maxwidth = input('What is the maximum width of the roads? ');
angtol = input('What is the maximum difference in direction for two edges to be paired? ');

% Get angles and edge values for each pixel:
[ang, edgevals] = getdirections(data);

% Pick the edge points:
edge = pickedgepts(ang, edgevals, p, anglimit, T1, T2, T3);

% Thin the edges:
thinedge = dothethin(edge);

% Link the edges:
[linkedge, branches, branchlength] = link(thinedge, ang, edgevals, numdirs, tooshort, maxgap, maxsoc);
% Pair up roughly parallel lines:
[pairs, newedge] = linepairs(edge, branches, branchlength, dirstize, minwidth, maxwidth, angtol);
```

% getimage.m

```
% Gets the image whose name is inputted at the prompt and puts the grayscale values into the % matrix 'data'. Must put single quotes around the image name. Called from 'mainnowave.m'.
```

```

image = input('Draw which image? ');
fd = fopen(image, 'r');
data = fread(fd, [500, 500], 'uchar');
data = data';

```

% **getdirections.m**

% Uses different size masks which represent ideal edges at various angles in order to decide % which direction to associate with each pixel. Looks at each pixel in 'data', calculates the % values of its neighborhood convolved with masks of a certain size (varies depending on the % value of 'level'), and finds the maximum of these values. The maximum becomes the edge % value at that point ('edgevals') and the angle corresponding to the mask which produced the % maximum becomes the angle value at that point ('ang'). Called from 'mainnowave.m'.

```

function [ang, edgevals] = getdirections(data)

```

```

% 5x5 masks

```

```

m0 = [ -100 -100  0 100 100;
       -100 -100  0 100 100;
       -100 -100  0 100 100;
       -100 -100  0 100 100;
       -100 -100  0 100 100];
m30 = [-100  32 100 100 100;
       -100 -78 92 100 100;
       -100 -100  0 100 100;
       -100 -100 -92 78 100;
       -100 -100 -100 -32 100];
m60 = [100 100 100 100 100;
       -32 78 100 100 100;
       -100 -92  0 92 100;
       -100 -100 -100 -78 32;
       -100 -100 -100 -100 -100];
m90 = [100 100 100 100 100;
       100 100 100 100 100;
       0  0  0  0  0;
       -100 -100 -100 -100 -100;
       -100 -100 -100 -100 -100];
m120 = [100 100 100 100 100;
        100 100 100 78 -32;
        100 92 0 -92 -100;
        32 -78 -100 -100 -100;
        -100 -100 -100 -100 -100];
m150 = [100 100 100 32 -100;
        100 100 92 -78 -100;
        100 100  0 -100 -100;
        100 78 -92 -100 -100;
        100 -32 -100 -100 -100];

```

```

% convolutions with the masks:

```

```

data0 = conv2(data, m0, 'same');
data30 = conv2(data, m30, 'same');
data60 = conv2(data, m60, 'same');
data90 = conv2(data, m90, 'same');
data120 = conv2(data, m120, 'same');
data150 = conv2(data, m150, 'same');

```

```

% find the maximums and the angles which produced the maximums:

```

```

[x, y] = size(data);

```



```

edgevals = zeros(x, y);
for i = 1:x
for j = 1:y
    vals(1) = data0(i, j);
    vals(2) = data30(i, j);
    vals(3) = data60(i, j);
    vals(4) = data90(i, j);
    vals(5) = data120(i, j);
    vals(6) = data150(i, j);

    [ind, edgevals(i, j)] = findmax(vals);

    if ind == 1
        ang(i, j) = 0;
    elseif ind == 2
        ang(i, j) = 30;
    elseif ind == 3
        ang(i, j) = 60;
    elseif ind == 4
        ang(i, j) = 90;
    elseif ind == 5
        ang(i, j) = 120;
    elseif ind == 6
        ang(i, j) = 150;
    end
end
end

```

% **findmax.m**

% Takes in a vector of values and returns the maximum value as well as the index associated % with the maximum value. Called from 'getdirections.m'.

```
function [maxind, maxval] = findmax(v)
```

```
v = abs(v);
```

```
[x, y] = size(v);
```

```
maxval = v(1);
```

```
maxind = 1;
```

```
i = 2;
```

```
while i <= y
```

```
    if v(i) > maxval
        maxval = v(i);
        maxind = i;
    end
```

```
    i = i + 1;
```

```
end
```

```
if maxval == 0
```

```
    maxind = 0;
```

```
end
```

% pickedgepts.m

% Decides which points of an image are edge points based on three tests: (1) Is the point (in % 'data') a local max with respect to its two neighbors in the direction normal to the point's % angle ('dataang')? (2) Are the angles of the two neighbors in the direction of the angle of the % point within 'anglimit' of the angle of the point? (3) Is the point value larger than some % threshold? Program can use any or all of these three tests: 'T1', 'T2' and 'T3' are either 0's or % 1's, indicating which tests to use. The value 'p' is the percentage of energy you want to keep % after thresholding the data values. 'data' is the matrix of edge values found in % 'getdirections.m'. Called from 'mainnowave.m'.

```
function edge = pickedgepts(dataang, data, p, anglimit, T1, T2, T3);
```

```
test1 = T1;      % 1 if using a certain test, 0 otherwise
test2 = T2;
test3 = T3;
    if test3 == 1      % for test 3, there needs to be a threshold:
        mmd = mean(mean(data));
        ddata = data./mmd;
        t = threshold(ddata, p);
        thresh = mmd*t;
    end

[s, t] = size(data);
numtests = 0;
if test1 == 1
    edge1 = zeros(s, t);
    numtests = numtests + 1;
end
if test2 == 1
    edge2 = zeros(s, t);
    numtests = numtests + 1;
end
if test3 == 1
    edge3 = zeros(s, t);
    numtests = numtests + 1;
end

for i = 3:s-2
    for j = 3:t-2
        wmbig = data(i-2:i+2, j-2:j+2);
        wm = data(i-1:i+1, j-1:j+1);
        wang = dataang(i-1:i+1, j-1:j+1);

        if test1 == 1
            edge1(i, j) = smalllocalmax2(wmbig, wang(2, 2));
        end

        if test2 == 1
            edge2(i, j) = rightdir(wang, anglimit);
        end

        if test3 == 1
            if data(i, j) > thresh
                edge3(i, j) = 1;
            end
        end
    end
end
end
```

```

% get rid of single points in individual images before combining:
if test1 == 1
    edge1 = singlepts(edge1);
end
if test2 == 1
    edge2 = singlepts(edge2);
end
if test3 == 1
    edge3 = singlepts(edge3);
end

% combine the edge images into one - if only one test is used, then the edge image from that % test is
% the final edge image - if two tests are used, then it looks at the two edge matrices - the % edge points may
% not be in exactly the same pixel spot in both matrices, so it finds an edge % point in the edge matrix
% from the first test, then searches the 3x3 neighborhood of that same % pixel spot in the second test matrix
% (allows for a 1 pixel offset) - if there are at least 2 edge % pixels in that neighborhood (2 seemed to work
% better than only 1), then the pixel % corresponding to that center point is designated an edge
% pixel.
if numtests == 1
    if (test1 == 1) & (test2 == 0) & (test3 == 0)
        edge = edge1;
    elseif (test1 == 0) & (test2 == 1) & (test3 == 0)
        edge = edge2;
    elseif (test1 == 0) & (test2 == 0) & (test3 == 1)
        edge = edge3;
    end
elseif numtests == 2
    if test1 == 1
        [x, y] = size(edge1);
    else
        [x, y] = size(edge2);
    end

    edge = zeros(x, y);

    if (test1 == 1) & (test2 == 1) & (test3 == 0)
        a = edge2;
        b = edge1;
    elseif (test1 == 1) & (test2 == 0) & (test3 == 1)
        a = edge3;
        b = edge1;
    elseif (test1 == 0) & (test2 == 1) & (test3 == 1)
        a = edge2;
        b = edge3;
    end

    for i = 2:x-1
        for j = 2:y-1
            if a(i, j) == 1
                g = b(i-1:i+1, j-1:j+1);
                count = checkforpts(g);
                if count >= 2
                    edge(i, j) = 1;
                end
            end
        end
    end

    edge = singlepts(edge);

```

```

elseif numtests == 3
    [x, y] = size(edge1);
    edge = zeros(x, y);

    for i = 2:x-1
        for j = 2:y-1
            if edge2(i, j) == 1
                g1 = edge1(i-1:i+1, j-1:j+1);
                g2 = edge3(i-1:i+1, j-1:j+1);
                count1 = checkforpts(g1);
                count2 = checkforpts(g2);
                if (count1 >= 2) & (count2 >= 2)
                    edge(i, j) = 1;
                end
            end
        end
    end

    edge = singlepts(edge);
end

```

% **threshold.m**

% Used to determine the threshold for a matrix which keeps the given percentage of energy 'p'. Called from 'pickedgepts.m'.

```
function t = threshold(m, p)
```

```
[x, y] = size(m);
```

```
mmax = max(max(m));
```

```

t = (p/100)*mmax;      % Starting guess is p percent of the max
b = sum(sum(m));       % Total energy of unthresholded matrix
e = 100;               % Percentage of energy left after thresholding

```

```
% While the percentage is not within 2:
```

```

while ((e < p-2) | (e > p+2))
    % threshold the matrix:
    for i = 1:x
        for j = 1:y
            if m(i, j) < t
                tm(i, j) = 0;
            else
                tm(i, j) = m(i, j);
            end
        end
    end
end

```

```

% find the energy left:
a = sum(sum(tm));

```

```

% find the percentage of energy left:
e = (a/b)*100;

```

```

% if there's not enough energy left, decrease the threshold:
if e < p-2
    tmpt = 0.7*t;

```

```

        t = tmpt;
    % if there's too much energy left, increase the threshold:
    elseif e > p+2
        tmpt = 1.3*t;
        t = tmpt;
    else
        t = t;
    end
end
end

```

% smalllocalmax2.m

% Takes in a 3x3 matrix of values and a 3x3 matrix of angles (directions) and determines % whether or not the center pixel is a local max in the direction normal to its angle. It first % checks whether the angles of these two neighboring pixels have the same angle as the target % pixel, and if they do it then checks if they both have smaller values than the target pixel. % Returns 1 if it is a local max, 0 otherwise. Called from 'pickedgepts.m'.

```
function yon = smalllocalmax(m, dir)
```

```
target = m(3, 3);
ang = dir;
```

```
if ang == 0
```

```
    a = m(3, 1);
    b = m(3, 2);
    c = m(3, 4);
    d = m(3, 5);
```

```
elseif (ang == 30) | (ang == 60)
```

```
    a = m(5, 1);
    b = m(4, 2);
    c = m(2, 4);
    d = m(1, 5);
```

```
elseif ang == 90
```

```
    a = m(1, 3);
    b = m(2, 3);
    c = m(4, 3);
    d = m(5, 3);
```

```
elseif (ang == 120) | (ang == 150)
```

```
    a = m(1, 1);
    b = m(2, 2);
    c = m(4, 4);
    d = m(5, 5);
```

```
end
```

```
if (target > a) & (target > b) & (target > c) & (target > d)
    yon = 1;
```

```
else
```

```
    yon = 0;
```

```
end
```

% rightdir.m

% Takes a 3x3 matrix of angles and determines whether the pixels neighboring the center in the % direction of the center angle are within 'limit' degrees of the center angle. Returns 1 if they % are, 0 otherwise. Called from 'pickedgepts.m'.

```

function yon = rightdir(ang, limit)

target = ang(2, 2);

if target == 0
    a = ang(1, 2);
    b = ang(3, 2);
elseif (target == 30) | (target == 60)
    a = ang(1, 1);
    b = ang(3, 3);
elseif target == 90
    a = ang(2, 1);
    b = ang(2, 3);
else
    a = ang(3, 1);
    b = ang(1, 3);
end

if (abs(target - a) < limit+1) & (abs(target - b) < limit+1)
    yon = 1;
else
    yon = 0;
end

```

% **singlepts.m**

% Goes through the edge maps and takes out single edge points. Called from 'pickedgepts.m'.

```

function new = singlepts(old)

m = [0 0 0; 0 1 0; 0 0 0];

[x, y] = size(old);

new = old;

for i = 2:x-1
    for j = 2:y-1
        n = old(i-1:i+1, j-1:j+1);
        if m == n
            new(i, j) = 0;
        end
    end
end
end

```

% **checkforpts.m**

% Counts how many points in a 3x3 matrix are edge points. Called from 'pickedgepts.m'.

```

function count = checkforpoints(g)

count = 0;

for i = 1:3
    for j = 1:3
        if g(i, j) == 1
            count = count + 1;
        end
    end
end

```

```

        end
    end
end

```

% dothethin.m

```

% Thins the edges of the image. Calls 'thin.m' to do the thinning on the original image, its transpose, and the horizontal, vertical and diagonal flips of both the original and the transpose.
Called from 'mainnowave.m'

```

```

% Note: This program may need to be run more then once on an image to thin all edges. For the road images I've been using so far, once is enough. But if the edges are quite thick, this % needs to be run and rerun to shave the edges down, iterate until no more edge points can be thrown away.

```

```

function new = dothethin(old)

```

```

[x, y] = size(old);

```

```

% thin the original image
new = thin(old);

```

```

% thin the horizontal flip of the original
new = flipud(new);
new = thin(new);
new = flipud(new);

```

```

% thin the vertical flip of the original
new = fliplr(new);
new = thin(new);
new = fliplr(new);

```

```

% thin the diagonal flip of the original
new = (rot90(rot90(new)))';
new = thin(new);
new = (rot90(rot90(new)))';

```

```

% thin the transpose of the original
new = new';
new = thin(new);

```

```

% thin the horizontal flip of the transpose
new = flipud(new);
new = thin(new);
new = flipud(new);

```

```

% thin the vertical flip of the transpose
new = fliplr(new);
new = thin(new);
new = fliplr(new);

```

```

% thin the diagonal flip of the transpose
new = (rot90(rot90(new)))';
new = thin(new);
new = (rot90(rot90(new)))';

```

```

% transpose back to get completely thinned image
new = new';

```

% **thin.m**

% Thins the edges to a width of one pixel. Compares a series of masks to the 3x3 neighborhood % of each pixel to determine whether that edge point can be removed or not. Because of % symmetry, the matrix has to be passed over several ways to remove all the extra edge points. % Called from 'dothethin.m'.

function new = thin(old)

[x, y] = size(old);

new = old;
clear old

m1 = [1 1 0; 0 1 0; 0 0 0];
m2 = [0 1 0; 1 1 0; 0 0 0];
m3 = [1 1 0; 1 1 0; 0 0 0];
m4 = [1 1 1; 0 1 0; 0 0 0];
m5 = [0 1 1; 1 1 0; 0 0 0];
m6 = [0 1 0; 1 1 1; 0 0 0];
m7 = [1 1 1; 1 1 0; 0 0 0];
m8 = [1 1 0; 1 1 1; 0 0 0];
m9 = [1 1 0; 0 1 1; 0 0 1];
m10 = [0 1 0; 1 1 1; 0 0 1];
m11 = [1 1 1; 1 1 1; 0 0 0];
m12 = [1 1 1; 0 1 1; 0 0 1];
m13 = [1 1 1; 1 1 0; 0 1 0];
m14 = [1 1 0; 1 1 1; 0 0 1];
m15 = [0 1 0; 1 1 1; 1 0 1];
m16 = [1 1 1; 1 1 0; 0 1 1];
m17 = [1 1 1; 1 1 1; 1 0 1];

thinned = 0;

% from left to right, top to bottom:

for i = 2:x-1
for j = 2:y-1
if new(i, j) == 1
insidethin;
end
end
end

% from right to left, top to bottom:

for i = 2:x-1
for j = y-1:-1:2
if new(i, j) == 1
insidethin;
end
end
end

% from top to bottom, left to right:

for j = 2:y-1
for i = 2:x-1
if new(i, j) == 1
insidethin;
end
end


```

end

% from bottom to top, left to right:
for j = 2:y-1
for i = x-1:-1:2
    if new(i, j) == 1
        insidethin;
    end
end
end

% from left to right, bottom to top:
for i = x-1:-1:2
for j = 2:y-1
    if new(i, j) == 1
        insidethin;
    end
end
end

% from right to left, bottom to top:
for i = x-1:-1:2
for j = y-1:-1:2
    if new(i, j) == 1
        insidethin;
    end
end
end

% from top to bottom, right to left:
for j = y-1:-1:2
for i = 2:x-1
    if new(i, j) == 1
        insidethin;
    end
end
end

% from bottom to top, right to left:
for j = y-1:-1:2
for i = x-1:-1:2
    if new(i, j) == 1
        insidethin;
    end
end
end
end

```

% insidethin.m

% Code that is repeated many times inside 'thin.m'.

```

a = new(i-1:i+1, j-1:j+1);
if a == m1
    new(i, j) = 0;
    thinned = thinned + 1;
end
if a == m2
    new(i, j) = 0;

```

```

        thinned = thinned + 1;
    end
    if a == m3
        new(i, j) = 0;
        thinned = thinned + 1;
    end
    if a == m4
        new(i, j) = 0;
        thinned = thinned + 1;
    end
    if a == m5
        new(i, j) = 0;
        thinned = thinned + 1;
    end
    if a == m6
        new(i, j) = 0;
        thinned = thinned + 1;
    end
    if a == m7
        new(i, j) = 0;
        thinned = thinned + 1;
    end
    if a == m8
        new(i, j) = 0;
        thinned = thinned + 1;
    end
    if a == m9
        new(i, j) = 0;
        thinned = thinned + 1;
    end
    if a == m10
        new(i, j) = 0;
        thinned = thinned + 1;
    end
    if a == m11
        new(i, j) = 0;
        thinned = thinned + 1;
    end
    if a == m12
        new(i, j) = 0;
        thinned = thinned + 1;
    end
    if a == m13
        new(i, j) = 0;
        thinned = thinned + 1;
    end
    if a == m14
        new(i, j) = 0;
        thinned = thinned + 1;
    end
    if a == m15
        new(i, j) = 0;
        thinned = thinned + 1;
    end
    if a == m16
        new(i, j) = 0;
        thinned = thinned + 1;
    end
    if a == m17

```

```

        new(i, j) = 0;
        thinned = thinned + 1;
end

```

% link.m

% Links by considering the last twenty steps of an edge to the endpoint. Starting at an endpoint, a tree is made consisting of several branches, each branch having either an endpoint or a branch point for each end. Combinations of branches are found which lead from each endpoint in the tree to all the others. For each endpoint, the straightest combination is chosen, then a poll is taken of the last twenty step directions in this edge. The most prevalent direction is considered the primary direction, the second most prevalent direction is the secondary direction. The confidences of these directions are also recorded, being the number of steps from the poll that were in that direction. Then using these directions and confidences, a search is made for the endpoint(s) to link with using 'checklink.m'. A search area is designed using the primary and secondary directions and endpoints within that area are linked if they pass certain conditions. Input arguments are: 'numdirs', number of steps used to find the principal and secondary directions; 'tooshort', edges shorter than this are deleted; 'maxgap', longest gap that can possibly be linked (length % of the search area); 'maxsoc', width of the search area. Called from 'mainnowave.m' and 'mainwave.m'.

```
function [edge, branches, branchlength] = link(old, ang, edgevals, numdirs, tooshort, maxgap, maxsoc)
```

```

% find end points:
[endedge, numendpts] = endpts(old);

```

```

% get info on each endpoint and its longest edge:

```

```

[x, y] = size(endedge);
laststeps = zeros(1, 6);
branchnum = 0;
branches = 0;
branchlength = 0;
branchdir = 0;

```

```

% first, make the 'tree' of the edge:

```

```

for i = 2:x-1
    for j = 2:y-1
        if endedge(i, j) == 2
            tmpbranchnum = 0;
            tmpbranches = 0;
            tmpbranchlength = 0;
            tmpbranchdir = 0;

            % if this point was a part of a previous tree, don't need to do this all over again:
            [sols, whatever] = size(laststeps);
            out = 0;
            for k = 1:sols
                if (i == laststeps(k, 1)) & (j == laststeps(k, 2))
                    out = 1;
                elseif (i == laststeps(k, 2)) & (j == laststeps(k, 1))
                    out = 1;
                end
            end

            if out == 0 % point hasn't been done before
                % start the search for branches:
                used = zeros(x, y);
                used(i, j) = 1;
            end
        end
    end
end

```

```

% search around the entire 1 pixel wide neighborhood of the
% root endpoint for first step:
for s = -1:1:1
for t = -1:1:1
    if (s ~= 0) | (t ~= 0)
        if endedge(i+s, j+t) ~= 0
            firststep = [i+s, j+t];
        end
    end
end

stack = [i j firststep 0];
stackheight = 1;
pairs = [0 0];

% get the information for all the branches:
while stackheight ~= 0
    start = stack(stackheight, 1:2);
    firststep = stack(stackheight, 3:4);

    [b, l, toe, used] = getbranch(start, firststep, endedge, used);

    if toe == 1
        stack(stackheight, :) = zeros(1, 5);
        stackheight = stackheight - 1;
    else
        tmpbranchnum = tmpbranchnum + 1;
        tmpbranches(tmpbranchnum, 1:2*(l+1)) = b;
        tmpbranchlength(tmpbranchnum) = l;
        tmpbranchdir(tmpbranchnum) =
            (atan((b(1)b(2*l+1))/(b(2)-b(2*l+2))))*(180/pi);
        pairs = [pairs; stack(stackheight, 5) tmpbranchnum];
        stack(stackheight, :) = zeros(1, 5);
        stackheight = stackheight - 1;

        I = b(2*l - 1:2*l);
        J = b(2*l + 1:2*l + 2);

        if toe == 3 % new branch
            % search around the entire 1 pixel wide neighborhood for next points:
            count1 = 0;
            nextstep = 0;
            for s = -1:1:1
            for t = -1:1:1
                if (s ~= 0) | (t ~= 0)
                    if endedge(J(1)+s, J(2)+t) ~= 0
                        if (J(1)+s ~= I(1)) | (J(2)+t ~= I(2))
                            if used(J(1)+s, J(2)+t) ~= 1
                                used(J(1)+s, J(2)+t) = 1;
                                count1 = count1 + 1;
                                nextstep = [nextstep J(1)+s
J(2)+t];
                            end
                        end
                    end
                end
            end
            end
        end
    end
end

```

```

end

brpt = J;
nextstep = nextstep(2:(2*count1 + 1));
for z = 1:count1
    stackheight = stackheight + 1;
    stack(stackheight, :) = [brpt
nextstep((2*z - 1):(2*z)) tmpbranchnum];
end
end
end

% find the total length, total number of pixels in the whole edge:
total = 1;
for u = 1:tmpbranchnum
    total = total + tmpbranchlength(u);
end

% get rid of edges 10 pixels or shorter:
if total < tooshort
    for u = 1:tmpbranchnum
        for v = 1:tmpbranchlength+1
            if (tmpbranches(u, 2*v-1) ~= 0) & (tmpbranches(u,
2*v) ~= 0)
                endedge(tmpbranches(u, 2*v-1),
tmpbranches(u, 2*v)) = 0;
            end
        end
    end
else
    % add the branch to the master list for later use:
    for u = 1:tmpbranchnum
        branchnum = branchnum + 1;
        branchlength(branchnum) = tmpbranchlength(u);
        branches(branchnum, 1:2*(branchlength(branchnum)+1)) =
tmpbranches(u, 1:2*(branchlength(branchnum)+1));
        branchdir(branchnum) = tmpbranchdir(u);
    end

    % if there is only one branch, things are simple:
    [p1, p2] = size(pairs);
    if p1 < 3
        rightcombo = 1;

        s1 = i;
        t1 = j;
        ud = zeros(1, tmpbranchlength(1));
        maxlen = tmpbranchlength(1);
        [a, s, t] = lastdirs(numdirs, rightcombo, ud, tmpbranches,
tmpbranchlength, maxlen);

        [dir1, conf1, dir2, conf2] = poll(a);
        laststeps = [laststeps; s1 t1 dir1 dir2 conf1 conf2];

        s2 = tmpbranches(1, 2*tmpbranchlength(1)+1);
        t2 = tmpbranches(1, 2*tmpbranchlength(1)+2);
        ud = ones(1, tmpbranchlength(1));
    end
end
end

```

```

tmpbranchlength, maxlength);

[a, s, t] = lastdirs(numdirs, rightcombo, ud, tmpbranches,

else
    [dir1, conf1, dir2, conf2] = poll(a);
    laststeps = [laststeps; s2 t2 dir1 dir2 conf1 conf2];

    % otherwise (if there is some branching).....
    % find the combinations of branches that lead to full %edges:
    [combos, ud] = getcombos(pairs);
    [c, whatever] = size(combos);

    % find the length of each combo:
    for u = 1:c
        v = 1;
        edglength(u) = 0;
        br = combos(u, v);
        while br ~= 0
            edglength(u) = edglength(u) +

        v = v + 1;
        br = combos(u, v);
    end

    % find the angle off of straight for each combo:
    for u = 1:c
        count1 = 1;
        tc = combos(u, 1);
        while tc ~= 0
            sc(count1) = tc;
            count1 = count1 + 1;
            tc = combos(u, count1);
        end

        lc = count1 - 1;

        bigstart = tmpbranches(sc(1), 1:2);
        bigfinish = tmpbranches(sc(lc),
2*tmpbranchlength(sc(lc))+1:2*tmpbranchlength(sc(lc))+2);
        bigdir = (atan((bigstart(1)-bigfinish(1))/(bigstart(2)-
bigfinish(2))))*(180/pi);

        edgedir(u) = 0;
        for v = 1:lc
            edgedir(u) = edgedir(u) + abs(bigdir -

        end
    end

    % make a list of the ends of the branches:
    ends = combos(1, 1);
    for u = 2:c
        t = combos(u, 1);
        add = 1;
        for v = 1:length(ends)
            if t == ends(v)
                add = 0;
            end
        end
        if add == 1
            ends = [ends t];

```

```

end
end

% for each end, find the best edge starting at that end % and
% use this edge to get the primary and secondary % directions
% and their confidences:
e = length(ends);
for u = 1:e
    startbranch = ends(u);

    l = 0;
    for v = 1:c
        if combos(v, 1) == startbranch
            l = [l edgedir(v)];
        else
            l = [l 1000];
        end
    end
    l = l(2:length(l));

    % find the minimum direction difference:
    [minind, mindir] = findmin(l);
    maxlength = edgelength(minind);

    % use the edge with maximum length to get %
    % edge directions and take a poll:
    w = 1;
    n = combos(minind, w);
    rightcombo = 0;
    while n ~= 0
        rightcombo = [rightcombo n];
        w = w + 1;
        n = combos(minind, w);
    end
    rightcombo = rightcombo(2:length(rightcombo));
    righttud = ud(minind, :);

    [a, s, t] = lastdirs(numdirs, rightcombo, righttud,

tmpbranches, tmpbranchlength, maxlength);

    [dir1, conf1, dir2, conf2] = poll(a);
    laststeps = [laststeps; s t dir1 dir2 conf1 conf2];
end
end
end
end
end

% link:
links = endedge;

[n, whatever] = size(laststeps);

for f = 2:n
    i = laststeps(f, 1);
    j = laststeps(f, 2);
    primdir = laststeps(f, 3);
    secdir = laststeps(f, 4);

```

```

        primconf = laststeps(f, 5);
        secconf = laststeps(f, 6);

        % use the primary and secondary directions to check for possible endpoints to link with:
        [tmp, branches, branchlength, branchnum] = checklink(links, ang, edgevals, laststeps, f, maxgap,
maxsoc, branches, branchlength, branchnum);
        links = tmp;
    end

[x, y] = size(links);
edge = zeros(x, y);

for i = 1:x
    for j = 1:y
        if links(i, j) ~= 0
            edge(i, j) = 1;
        end
    end
end
end

```

% **endpts.m**

% Goes through and puts a 2 in the place of every end point. Called from 'link.m'.

```
function [new, numendpts] = endpts(old)
```

```
[x, y] = size(old);
```

% first make two outer squares of pixels zero:

```

for i = 1:x
    old(i, 1) = 0;
    old(i, 2) = 0;
    old(i, y-1) = 0;
    old(i, y) = 0;
end

```

```

for j = 1:y
    old(1, j) = 0;
    old(2, j) = 0;
    old(x-1, j) = 0;
    old(x, j) = 0;
end

```

```

new = old;
numendpts = 0;

```

% masks used to determine if a point is an endpoint:

```

m1 = [1 0 0; 0 1 0; 0 0 0];
m2 = [0 1 0; 0 1 0; 0 0 0];
m3 = [0 0 1; 0 1 0; 0 0 0];
m4 = [0 0 0; 0 1 1; 0 0 0];
m5 = [0 0 0; 0 1 0; 0 0 1];
m6 = [0 0 0; 0 1 0; 0 1 0];
m7 = [0 0 0; 0 1 0; 1 0 0];
m8 = [0 0 0; 1 1 0; 0 0 0];

```

```

for i = 2:x-1
    for j = 2:y-1

```



```

    if old(i, j) == 1
        a = old(i-1:i+1, j-1:j+1);

        if a == m1
            new(i, j) = 2;
            numendpts = numendpts + 1;
        end
        if a == m2
            new(i, j) = 2;
            numendpts = numendpts + 1;
        end
        if a == m3
            new(i, j) = 2;
            numendpts = numendpts + 1;
        end
        if a == m4
            new(i, j) = 2;
            numendpts = numendpts + 1;
        end
        if a == m5
            new(i, j) = 2;
            numendpts = numendpts + 1;
        end
        if a == m6
            new(i, j) = 2;
            numendpts = numendpts + 1;
        end
        if a == m7
            new(i, j) = 2;
            numendpts = numendpts + 1;
        end
        if a == m8
            new(i, j) = 2;
            numendpts = numendpts + 1;
        end
    end
end
end

```

```

% get rid of useless end points:
new = badendpts(new);

```

% **badendpts.m**

% For the purpose of linking, get rid of bad end points, ones that branch after only one step. % Called from 'endpts.m'.

```
function m = badendpts(m)
```

```
[x, y] = size(m);
```

```
for i = 1:x
```

```
for j = 1:y
```

```
    if m(i, j) == 2
```

```
        firststep = 0;
```

```
        count = 0;
```

```
        points = 0;
```

```

        for s1 = -1:1:1
            for t1 = -1:1:1
                if (s1 ~= 0) | (t1 ~= 0)
                    if m(i+s1, j+t1) ~= 0
                        firststep = [i+s1, j+t1];
                    end
                end
            end
        end

        for s2 = -1:1:1
            for t2 = -1:1:1
                if (s2 ~= 0) | (t2 ~= 0)
                    if m(firststep(1)+s2, firststep(2)+t2);
                        count = count + 1;
                        points = [points firststep(1)+s2 firststep(2)+t2];
                    end
                end
            end
        end

        if count > 2
            m(i, j) = 0;
        end
    end
end
end

```

% **getbranch.m**

% This function records the starting point and ending point of a branch and the coordinates of % the points in between. Returns a value 'typeofend' which indicates whether the branch ended % in another branch point, an endpoint or a plain edge point. Also returns the length of this % branch, including the end point but not the start point. Called from 'link.m'.

```
function [branch, length, typeofend, used] = getbranch(start, firststep, new, used)
```

```

branch = [start];
length = 0;
newbranch = 0;           % indicates whether we've come to a new branch point
endbranch = 0;           % indicates whether we've come to a dead end

```

```

lastpoint = start;
nextpoint = firststep;

```

```

while (newbranch == 0) & (endbranch == 0)
    dir1 = dirstep(lastpoint, nextpoint);
    branch = [branch nextpoint];
    length = length + 1;
    lastpoint = nextpoint;

    dir2 = oppositedir(dir1);
    [c, numc, used] = searchforpts(dir2, lastpoint, new, used);

    if numc > 1
        newbranch = 1;
        typeofend = 3;           % branch point
    end
end

```

```

        for i = 1:numc
            a = c(2*i - 1);
            b = c(2*i);
            used(a, b) = 0;
        end
    elseif numc == 0
        endbranch = 1;
        if new(lastpoint(1), lastpoint(2)) == 2
            typeofend = 2;          % endpoint
        else
            typeofend = 1;          % regular edge point
        end
    else
        nextpoint = c;
    end
end
end

```

% dirstep.m

% Given two points, determine what direction was taken to get from one to two: 1 = NW, 2 = N, % 3 = NE, 4 = W, 5 = E, 6 = SW, 7 = S, 8 = SE. Called from 'getbranch.m'.

function dir = dirstep(two, one)

```

x1 = one(1);
y1 = one(2);
x2 = two(1);
y2 = two(2);

if x2 < x1
    if y2 < y1
        dir = 1;
    elseif y2 == y1
        dir = 2;
    else
        dir = 3;
    end
elseif x2 == x1
    if y2 < y1
        dir = 4;
    else
        dir = 5;
    end
elseif x2 > x1
    if y2 < y1
        dir = 6;
    elseif y2 == y1
        dir = 7;
    else
        dir = 8;
    end
end
end

```

% oppositedir.m

% Returns the opposite direction from that which is inputted. Called from 'getbranch.m'.

```
function newdir = oppositedir(olddir)
```

```
if olddir == 1
    newdir = 8;
elseif olddir == 2
    newdir = 7;
elseif olddir == 3
    newdir = 6;
elseif olddir == 4
    newdir = 5;
elseif olddir == 5
    newdir = 4;
elseif olddir == 6
    newdir = 3;
elseif olddir == 7
    newdir = 2;
elseif olddir == 8
    newdir = 1;
end
```

% searchforpts.m

% Used to search the proper area for edge points and return a vector that includes the necessary % information. Called from 'getbranch.m'

```
function [c, numc, used] = searchforpts(dir, target, new, used)
```

```
x = target(1);
y = target(2);
[s, t] = size(new);
numc = 0;
c = 0;

if (x == 1) | (y == 1) | (x == s) | (y == t)
    return;
else
    if dir == 1 % NW search
        if new(x+1, y-1) ~= 0
            if used(x+1, y-1) == 0
                c = [c x+1 y-1];
                numc = numc + 1;
                used(x+1, y-1) = 1;
            end
        end
        if new(x, y-1) ~= 0
            if used(x, y-1) == 0
                c = [c x y-1];
                numc = numc + 1;
                used(x, y-1) = 1;
            end
        end
        if new(x-1, y-1) ~= 0
            if used(x-1, y-1) == 0
                c = [c x-1 y-1];
                numc = numc + 1;
                used(x-1, y-1) = 1;
            end
        end
    end
end
```

```

    if new(x-1, y) ~= 0
    if used(x-1, y) == 0
        c = [c x-1 y];
        numc = numc + 1;
        used(x-1, y) = 1;
    end
    end
    if new(x-1, y+1) ~= 0
    if used(x-1, y+1) == 0
        c = [c x-1 y+1];
        numc = numc + 1;
        used(x-1, y+1) = 1;
    end
    end
elseif dir == 2 % N search
    if new(x-1, y-1) ~= 0
    if used(x-1, y-1) == 0
        c = [c x-1 y-1];
        numc = numc + 1;
        used(x-1, y-1) = 1;
    end
    end
    if new(x-1, y) ~= 0
    if used(x-1, y) == 0
        c = [c x-1 y];
        numc = numc + 1;
        used(x-1, y) = 1;
    end
    end
    if new(x-1, y+1) ~= 0
    if used(x-1, y+1) == 0
        c = [c x-1 y+1];
        numc = numc + 1;
        used(x-1, y+1) = 1;
    end
    end
    if new(x, y-1) ~= 0
    if used(x, y-1) == 0
        c = [c x y-1];
        numc = numc + 1;
        used(x, y-1) = 1;
    end
    end
    if new(x, y+1) ~= 0
    if used(x, y+1) == 0
        c = [c x y+1];
        numc = numc + 1;
        used(x, y+1) = 1;
    end
    end
elseif dir == 3 % NE search
    if new(x-1, y-1) ~= 0
    if used(x-1, y-1) == 0
        c = [c x-1 y-1];
        numc = numc + 1;
        used(x-1, y-1) = 1;
    end
    end

```

```

end
if new(x-1, y) ~= 0
if used(x-1, y) == 0
    c = [c x-1 y];
    numc = numc + 1;
    used(x-1, y) = 1;
end
end
if new(x-1, y+1) ~= 0
if used(x-1, y+1) == 0
    c = [c x-1 y+1];
    numc = numc + 1;
    used(x-1, y+1) = 1;
end
end
if new(x, y+1) ~= 0
if used(x, y+1) == 0
    c = [c x y+1];
    numc = numc + 1;
    used(x, y+1) = 1;
end
end
if new(x+1, y+1) ~= 0
if used(x+1, y+1) == 0
    c = [c x+1 y+1];
    numc = numc + 1;
    used(x+1, y+1) = 1;
end
end
end
elseif dir == 4 % W search
if new(x-1, y-1) ~= 0
if used(x-1, y-1) == 0
    c = [c x-1 y-1];
    numc = numc + 1;
    used(x-1, y-1) = 1;
end
end
if new(x-1, y) ~= 0
if used(x-1, y) == 0
    c = [c x-1 y];
    numc = numc + 1;
    used(x-1, y) = 1;
end
end
if new(x, y-1) ~= 0
if used(x, y-1) == 0
    c = [c x y-1];
    numc = numc + 1;
    used(x, y-1) = 1;
end
end
if new(x+1, y-1) ~= 0
if used(x+1, y-1) == 0
    c = [c x+1 y-1];
    numc = numc + 1;
    used(x+1, y-1) = 1;
end
end
end
end

```

```

        if new(x+1, y) ~= 0
        if used(x+1, y) == 0
            c = [c x+1 y];
            numc = numc + 1;
            used(x+1, y) = 1;
        end
    end

elseif dir == 5 % E search
    if new(x-1, y) ~= 0
    if used(x-1, y) == 0
        c = [c x-1 y];
        numc = numc + 1;
        used(x-1, y) = 1;
    end
    end
    if new(x-1, y+1) ~= 0
    if used(x-1, y+1) == 0
        c = [c x-1 y+1];
        numc = numc + 1;
        used(x-1, y+1) = 1;
    end
    end
    if new(x, y+1) ~= 0
    if used(x, y+1) == 0
        c = [c x y+1];
        numc = numc + 1;
        used(x, y+1) = 1;
    end
    end
    if new(x+1, y) ~= 0
    if used(x+1, y) == 0
        c = [c x+1 y];
        numc = numc + 1;
        used(x+1, y) = 1;
    end
    end
    if new(x+1, y+1) ~= 0
    if used(x+1, y+1) == 0
        c = [c x+1 y+1];
        numc = numc + 1;
        used(x+1, y+1) = 1;
    end
    end

elseif dir == 6 % SW search
    if new(x-1, y-1) ~= 0
    if used(x-1, y-1) == 0
        c = [c x-1 y-1];
        numc = numc + 1;
        used(x-1, y-1) = 1;
    end
    end
    if new(x, y-1) ~= 0
    if used(x, y-1) == 0
        c = [c x y-1];
        numc = numc + 1;
        used(x, y-1) = 1;
    end
    end
end

```

```

end
if new(x+1, y-1) ~= 0
if used(x+1, y-1) == 0
    c = [c x+1 y-1];
    numc = numc + 1;
    used(x+1, y-1) = 1;
end
end
if new(x+1, y) ~= 0
if used(x+1, y) == 0
    c = [c x+1 y];
    numc = numc + 1;
    used(x+1, y) = 1;
end
end
if new(x+1, y+1) ~= 0
if used(x+1, y+1) == 0
    c = [c x+1 y+1];
    numc = numc + 1;
    used(x+1, y+1) = 1;
end
end
end
end

elseif dir == 7          % S search
if new(x, y-1) ~= 0
if used(x, y-1) == 0
    c = [c x y-1];
    numc = numc + 1;
    used(x, y-1) = 1;
end
end
if new(x+1, y-1) ~= 0
if used(x+1, y-1) == 0
    c = [c x+1 y-1];
    numc = numc + 1;
    used(x+1, y-1) = 1;
end
end
if new(x+1, y) ~= 0
if used(x+1, y) == 0
    c = [c x+1 y];
    numc = numc + 1;
    used(x+1, y) = 1;
end
end
if new(x+1, y+1) ~= 0
if used(x+1, y+1) == 0
    c = [c x+1 y+1];
    numc = numc + 1;
    used(x+1, y+1) = 1;
end
end
if new(x, y+1) ~= 0
if used(x, y+1) == 0
    c = [c x y+1];
    numc = numc + 1;
    used(x, y+1) = 1;
end
end
end
end

```



```

else                                % SE search
    if new(x+1, y-1) ~= 0
    if used(x+1, y-1) == 0
        c = [c x+1 y-1];
        numc = numc + 1;
        used(x+1, y-1) = 1;
    end
    end
    if new(x+1, y) ~= 0
    if used(x+1, y) == 0
        c = [c x+1 y];
        numc = numc + 1;
        used(x+1, y) = 1;
    end
    end
    if new(x+1, y+1) ~= 0
    if used(x+1, y+1) == 0
        c = [c x+1 y+1];
        numc = numc + 1;
        used(x+1, y+1) = 1;
    end
    end
    if new(x, y+1) ~= 0
    if used(x, y+1) == 0
        c = [c x y+1];
        numc = numc + 1;
        used(x, y+1) = 1;
    end
    end
    if new(x-1, y+1) ~= 0
    if used(x-1, y+1) == 0
        c = [c x-1 y+1];
        numc = numc + 1;
        used(x-1, y+1) = 1;
    end
    end
end

c = c(2:length(c));
end

```

% lastdirs.m

% Given a vector of numbers 'combo' which represent the branches of an edge beginning with % an endpoint, this function records the last 'n' step directions up to the endpoint. Called from % 'link.m'.

function [a, s, t] = lastdirs(n, rightcombo, righttud, branches, branchlength, maxlength)

```

count1 = 1;    % keeps track of how many directions we've recorded
count2 = 1;    % keeps track of where in the combo we are
count3 = 1;    % keeps track of how many directions we've used from a
               % single branch

```

```

if maxlength < n
    numdirs = maxlength;
else

```

```

        numdirs = n;
    end

    branchnum = rightcombo(count2);
    ud = righttud(count2);

    if ud == 1
        s = branches(branchnum, 2*branchlength(branchnum) + 1);
        t = branches(branchnum, 2*branchlength(branchnum) + 2);
    else
        s = branches(branchnum, 1);
        t = branches(branchnum, 2);
    end

    while count1 <= numdirs
        branch = branches(branchnum, 1:2*(branchlength(branchnum) + 1));
        if ud == 1
            for i = 1:branchlength(branchnum)+1
                tmpbranch(2*i-1:2*i) = branch(2*(branchlength(branchnum)-i+2)-
1:2*(branchlength(branchnum)-i+2));
            end
            branch = tmpbranch;
        end

        if count3 < branchlength(branchnum)+1
            dir = dirstep([branch(2*count3+1) branch(2*count3+2)], [branch(2*count3-1)
branch(2*count3)]);
            if ud == 1
                a(count1) = oppositedir(dir);
            else
                a(count1) = dir;
            end
            count1 = count1 + 1;
            count3 = count3 + 1;
        else
            count2 = count2 + 1;
            count3 = 1;
            branchnum = rightcombo(count2);
            ud = righttud(count2);
        end
    end
end

```

% poll.m

% Takes a poll of the directions in the vector 'a', returning the first and second most used % directions
and their confidences (how often they popped up). Called from 'link.m'.

```
function [dir1, conf1, dir2, conf2] = poll(a)
```

```
count = zeros(1, 8);
```

```
for i = 1:length(a)
    if a(i) ~= 0
        d = a(i);
        count(d) = count(d) + 1;
    end
end
end

```

```
[dir1, conf1] = findmax(count);
```

```
count(dir1) = 0;
```

```
[dir2, conf2] = findmax(count);
```

% getcombos.m

% Used to figure out the combinations of branches that lead from one endpoint to another. % Called from 'link.m'.

```
function [combos, ud] = getcombos(pairs);
```

```
% first get the combos leading from the root endpoint:
```

```
[numpairs, whatever] = size(pairs);
```

```
firstnumber = 0;
```

```
for u = 1:numpairs
```

```
    n = pairs(u, 1);
```

```
    [whatever, c] = size(firstnumber);
```

```
    add = 1;
```

```
    for v = 1:c
```

```
        if n == firstnumber(v)
```

```
            add = 0;
```

```
        end
```

```
    end
```

```
    if add == 1
```

```
        firstnumber = [firstnumber n];
```

```
    end
```

```
end
```

```
for u = 1:numpairs
```

```
    endbranch = 1;
```

```
    n = pairs(u, 2);
```

```
    [whatever, c] = size(firstnumber);
```

```
    for v = 1:c
```

```
        if n == firstnumber(v)
```

```
            endbranch = 0;
```

```
        end
```

```
    end
```

```
    if endbranch == 1
```

```
        combos(u, 1) = pairs(u, 2);
```

```
        combos(u, 2) = pairs(u, 1);
```

```
        next = pairs(u, 1);
```

```
        count1 = 3;
```

```
        while next ~= 1
```

```
            for w = 1:numpairs
```

```
                if next == pairs(w, 2)
```

```
                    newnext = pairs(w, 1);
```

```
                end
```

```
            end
```

```
            combos(u, count1) = newnext;
```

```
            next = newnext;
```

```
            count1 = count1 + 1;
```

```
        end
```

```
    end
```

```
end
```

```

oldcombos = combos;
[s, t] = size(oldcombos);
combos = zeros(1, t);
count2 = 1;
for i = 1:s
    if oldcombos(i, 1) ~= 0
        combos(count2, :) = oldcombos(i, :);
        count2 = count2 + 1;
    end
end

% get all other combos from these ones:
[c, whatever] = size(combos);
ud = ones(size(combos));
combos = [combos zeros(c, 1)];
d = c + 1;

for i = 1:c
    for j = i+1:c
        length1 = 1;
        w1 = combos(i, 1);
        a = w1;
        while w1 ~= 0
            length1 = length1 + 1;
            w1 = combos(i, length1);
            a = [a w1];
        end
        length1 = length1 - 1;
        a = a(1:length1);

        length2 = 1;
        w2 = combos(j, 1);
        b = w2;
        while w2 ~= 0
            length2 = length2 + 1;
            w2 = combos(j, length2);
            b = [b w2];
        end
        length2 = length2 - 1;
        b = b(1:length2);

        if length1 <= length2
            while length1 > 0
                if a(length1) == b(length2)
                    a(length1) = 0;
                    b(length2) = 0;
                end
                length1 = length1 - 1;
                length2 = length2 - 1;
            end
        else
            while length2 > 0
                if a(length1) == b(length2)
                    a(length1) = 0;
                    b(length2) = 0;
                end
                length1 = length1 - 1;
                length2 = length2 - 1;
            end
        end
    end
end

```

```

        end
    end

    v = 0;
    w = 0;
    for k = 1:length(a)
        if a(k) ~= 0
            v = [v a(k)];
            w = [w 1];
        end
    end
    for k = length(b):-1:1
        if b(k) ~= 0
            v = [v b(k)];
            w = [w 0];
        end
    end

    for l = 2:length(v)
        combos(d, l-1) = v(l);
        ud(d, l-1) = w(l);
    end
    d = d + 1;
end

end

% also add the flip of each combo
[c, whatever] = size(combos);
combos = [combos zeros(c, 1)];
d = c + 1;
for i = 1:c
    v = combos(i, 1);
    t = 2;
    vadd = combos(i, t);
    while vadd ~= 0
        v = [v vadd];
        t = t + 1;
        vadd = combos(i, t);
    end

    v = fliplr(v);

    w = ud(i, 1:length(v));
    w = fliplr(w);

    for j = 1:length(v)
        combos(d, j) = v(j);
        if w(j) == 1
            ud(d, j) = 0;
        elseif w(j) == 0
            ud(d, j) = 1;
        end
    end
    d = d + 1;
end
end

```

% findmin.m

% Takes in a vector of values and returns the minimum value as well as the index associated % with the minimum value. Called from 'link.m' and 'checklink.m'.

```
function [minind, minval] = findmin(v)
```

```
v = abs(v);
```

```
[x, y] = size(v);
```

```
minval = v(1);
```

```
minind = 1;
```

```
i = 2;
```

```
while i <= y
```

```
    if v(i) < minval
```

```
        minval = v(i);
```

```
        minind = i;
```

```
    end
```

```
    i = i + 1;
```

```
end
```

% checklink.m

% Called to find links. Searches an area determined by the primary direction, the secondary % direction, 'maxgap' (which indicates how many steps to take outward), and 'maxsoc' (which % indicates how many steps to take sideways) and lists all the endpoints in this area. Then, % based on how many steps off center an endpoint lies, how far it is from the root endpoint, % how aligned the two points are, the program decides whether to link or not. Called from % 'link.m'.

```
function [m, branches, branchlength, branchnum] = checklink(m, ang, edgevals, laststeps, f, maxgap, maxsoc, branches, branchlength, branchnum)
```

```
i = laststeps(f, 1);
```

```
j = laststeps(f, 2);
```

```
leftstart = [i j];
```

```
rightstart = [i j];
```

```
linkstart = [i j];
```

```
linkfinish = [i j];
```

```
endpoints = [i j 0 0 ang(i, j)];
```

```
num = 0;
```

```
primdir = laststeps(f, 3);
```

```
secdir = laststeps(f, 4);
```

```
primconf = laststeps(f, 5);
```

```
seccconf = laststeps(f, 6);
```

% left and right step directions determined by primary and secondary directions:

```
if (((primdir == 1) & ((secdir == 1) | (secdir == 2) | (secdir == 3))) |  
    ((primdir == 2) & ((secdir == 1) | (secdir == 2) | (secdir == 3))) |  
    ((primdir == 3) & ((secdir == 1) | (secdir == 2) | (secdir == 3))) |  
    ((primdir == 6) & ((secdir == 6) | (secdir == 7) | (secdir == 8))) |  
    ((primdir == 7) & ((secdir == 6) | (secdir == 7) | (secdir == 8))) |  
    ((primdir == 8) & ((secdir == 6) | (secdir == 7) | (secdir == 8))))
```

```
    leftdir = 4;
```

```
    rightdir = 5;
```

```
elseif (((primdir == 1) & ((secdir == 4) | (secdir == 6))) |
```

```

        ((primdir == 3) & ((secdir == 5) | (secdir == 8))) |
        ((primdir == 4) & ((secdir == 1) | (secdir == 4) | (secdir == 6))) |
        ((primdir == 5) & ((secdir == 3) | (secdir == 5) | (secdir == 8))) |
        ((primdir == 6) & ((secdir == 1) | (secdir == 4))) |
        ((primdir == 8) & ((secdir == 3) | (secdir == 5)))
    leftdir = 2;
    rightdir = 7;
elseif (((primdir == 2) & (secdir == 5)) | ((primdir == 4) & (secdir == 7)) |
        ((primdir == 5) & (secdir == 2)) | ((primdir == 7) & (secdir == 4)))
    leftdir = 1;
    rightdir = 8;
elseif (((primdir == 2) & (secdir == 4)) | ((primdir == 4) & (secdir == 2)) |
        ((primdir == 5) & (secdir == 7)) | ((primdir == 7) & (secdir == 5)))
    leftdir = 3;
    rightdir = 6;
else
    return;
end

```

% main path which determines the shape of the search area is designed using the primary and % secondary directions for an edge - this path is repeated during the search:

```

total = primconf + secconf;
ratio = floor(total/secconf);
stepdir = zeros(1, total);
usedsec = 0;
for t = 1:total
    if (mod(t, ratio) == 0) & (usedsec < secconf)
        stepdir(t) = secdir;
        usedsec = usedsec + 1;
    else
        stepdir(t) = primdir;
    end
end
end

```

% first the center path is searched for endpoints, then with each iteration a step is taken either % to the left or right before again searching the same main path steps from the new starting % point - when an endpoint is found, it is added to the matrix 'endpts' which keeps track of its % position, the number of steps off center, its distance to the root endpoint, and its angle:

```

v = 0;
count = 0;

while count < (maxsoc+1) % indicates how many steps to either side the search includes
    if mod(count, 2) == 0
        start = leftstart;
    else
        start = rightstart;
    end

    gap = 0;
    while v ~= 3
        for t = 1:total
            dir = stepdir(t);
            [v, finish] = takestep(start, dir, m);
            start = finish;
            gap = gap + 1;

            if v == 2
                if mod(count, 2) == 0
                    soc = count/2;

```

```

        else
            soc = (count + 1)/2;
        end

        dist = sqrt(((i - finish(1))^2) + ((j - finish(2))^2));

        endpoints = [endpoints; finish soc dist ang(finish(1), finish(2))];
        num = num + 1;
    end

    if gap > maxgap
        v = 3;
    end

    if v == 3
        break;
    end
end
end

count = count + 1;
if mod(count, 2) == 0
    [v, finish] = takestep(leftstart, leftdir, m);
    leftstart = finish;
else
    [v, finish] = takestep(rightstart, rightdir, m);
    rightstart = finish;
end
if v == 2
    if mod(count, 2) == 0
        soc = count/2;
    else
        soc = (count + 1)/2;
    end

    dist = sqrt(((i - finish(1))^2) + ((j - finish(2))^2));

    endpoints = [endpoints; finish soc dist ang(finish(1), finish(2))];
    num = num + 1;
end
end

% list of endpoints is searched to find which one(s) to link to:

% first check if any lie directly on the center path - if so, and if the link has not been made % before,
link them:
for s = 2:num+1
    soc = endpoints(s, 3);
    if soc == 0
        linkfinish = endpoints(s, 1:2);

        for k = 1:branchnum
            a = branches(branchnum, 1:2);
            b = branches(branchnum,
2*branchlength(branchnum)+1:2*branchlength(branchnum)+2);
            if ((a == linkstart) & (b == linkfinish)) | ((a == linkfinish) & (b == linkstart))
                % link has been made before
                return;
            end
        end
    end
end

```



```

        end

        [m, branches, branchlength, branchnum] = drawlink(linkstart, linkfinish, m, edgevals,
branches, branchlength, branchnum);
        return;
    end
end

% if no link was made above, find the endpoint with the shortest distance:
for s = 1:num+1
    dist(s) = endpoints(s, 4);
end

dist(1) = maxgap;

while 1 == 1
    [minind, mindist] = findmin(dist);

    if mindist == maxgap
        return;
    else
        linkfinish = endpoints(minind, 1:2);
        stepsoffcenter = endpoints(minind, 3);

        al = 0;
        for k = 1:branchnum
            a = branches(branchnum, 1:2);
            b = branches(branchnum,
2*branchlength(branchnum)+1:2*branchlength(branchnum)+2);
            if ((a == linkstart) & (b == linkfinish)) | ((a == linkfinish) & (b == linkstart))
                % link has been made before
                al = al + 1;
            end
        end

        if al > 2 % if link has been made before
            if mindist > 10 % if the length of the gap is large, end here
                return;
            else % if the length of the gap is small, repeat the step
                dist(minind) = maxgap;
            end
        else % if the link has not been made before
            if stepsoffcenter > 3
                % check orientations of two endpoints using primary direction:
                primdirstart = primdir;
                primdirfinish = primdir;

                [l, whatever] = size(laststeps);
                for z = 1:l
                    if (laststeps(z, 1) == linkfinish(1)) & (laststeps(z, 2) ==
linkfinish(2))
                        if laststeps(z, 4) ~= 0
                            primdirfinish = laststeps(z, 4);
                        end
                    end
                end

                primdiff = dirdiff(primdirstart, primdirfinish);
            end
        end
    end
end

```

```

        if (primdiff < 3)
            if stepsoffcenter < 6
                [m, branches, branchlength, branchnum] =
drawlink(linkstart, linkfinish, m, edgevals, branches, branchlength, branchnum);
            else
                if mindist > 20
                    [m, branches, branchlength, branchnum] =
drawlink(linkstart, linkfinish, m, edgevals, branches, branchlength, branchnum);
                end
            end
        end
    else
        [m, branches, branchlength, branchnum] = drawlink(linkstart, linkfinish,
m, edgevals, branches, branchlength, branchnum);
    end

    if mindist > 10
        return;
    else
        dist(minind) = maxgap;
    end
end
end
end
end

```

% **takestep.m**

% Takes one step in the given direction and returns the value and the coordinates of that % location.
 Called from 'checklink.m' and 'pairup.m'.

```
function [a, finish] = takestep(start, dir, m)
```

```
x = start(1);
y = start(2);
```

```
[s, t] = size(m);
```

```
if (x == 1) | (x == s) | (y == 1) | (y == t)
```

```
    a = 3;
    finish = start;
```

```
else
```

```
    if dir == 1
        a = m(x-1, y-1);
        finish = [x-1 y-1];
```

```
    elseif dir == 2
        a = m(x-1, y);
        finish = [x-1, y];
```

```
    elseif dir == 3
        a = m(x-1, y+1);
        finish = [x-1, y+1];
```

```
    elseif dir == 4
        a = m(x, y-1);
        finish = [x, y-1];
```

```
    elseif dir == 5
        a = m(x, y+1);
        finish = [x, y+1];
```

```
    elseif dir == 6
        a = m(x+1, y-1);
```

```

        finish = [x+1, y-1];
elseif dir == 7
    a = m(x+1, y);
    finish = [x+1, y];
else
    a = m(x+1, y+1);
    finish = [x+1, y+1];
end
end
end

```

% drawlink.m

% Given the start and the end points, this draws a link between them, not necessarily the best % link, but approximately a straight line. Before it actually draws the link, it checks whether the % proposed link crosses any other lines. If so, the link is not made. Called from 'checklink.m'.

```
function [m, branches, branchlength, branchnum] = drawlink(start, finish, m, edgevals, branches,
branchlength, branchnum)
```

```

path = start;

if (start(1) == finish(1)) & (start(2) == finish(2))
    return;
elseif start(1) == finish(1) % horizontal path
    next(1) = start(1);

    if start(2) > finish(2) % move to the left
        next(2) = start(2) - 1;
        while next(2) ~= finish(2)
            path = [path; next(1) next(2)];
            next(2) = next(2) - 1;
        end
        path = [path; next(1) next(2)];
    else % move to the right
        next(2) = start(2) + 1;
        while next(2) ~= finish(2)
            path = [path; next(1) next(2)];
            next(2) = next(2) + 1;
        end
        path = [path; next(1) next(2)];
    end

elseif start(2) == finish(2) % vertical path
    next(2) = start(2);

    if start(1) > finish(1) % move up
        next(1) = start(1) - 1;
        while next(1) ~= finish(1)
            path = [path; next(1) next(2)];
            next(1) = next(1) - 1;
        end
        path = [path; next(1) next(2)];
    else % move down
        next(1) = start(1) + 1;
        while next(1) ~= finish(1)

```

```

        path = [path; next(1) next(2)];
        next(1) = next(1) + 1;
    end
    path = [path; next(1) next(2)];
end

elseif start(1) > finish(1)
    if start(2) > finish(2) % up & left path
        xdiff = start(1) - finish(1);
        ydiff = start(2) - finish(2);

        if xdiff == ydiff % move ul-diagonal
            next(1) = start(1) - 1;
            next(2) = start(2) - 1;
            while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
                path = [path; next(1) next(2)];
                next(1) = next(1) - 1;
                next(2) = next(2) - 1;
            end
            path = [path; next(1) next(2)];
            path = gostraight(next, finish, m, path);
        elseif xdiff < ydiff % move ul-diagonal and left
            numdiag = xdiff;
            numstr = ydiff - xdiff;
            total = ydiff;

            if numdiag <= numstr % more left steps than ul-diagonal
                ratio = floor(total/numdiag);
                used = 0;
                s = 1;
                next(1) = start(1);
                next(2) = start(2) - 1;
                while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
                    path = [path; next(1) next(2)];
                    if (mod(s, ratio) == 0) & (used < numdiag)
                        next(1) = next(1) - 1;
                        used = used + 1;
                    end
                    next(2) = next(2) - 1;
                    s = s + 1;
                end
                path = [path; next(1) next(2)];
                path = gostraight(next, finish, m, path);
            elseif numdiag > numstr % more ul-diagonal steps than left
                ratio = floor(total/numstr);
                used = 0;
                s = 1;
                next(1) = start(1) - 1;
                next(2) = start(2) - 1;
                while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
                    path = [path; next(1) next(2)];
                    if (mod(s, ratio) == 0) & (used < numstr)
                        used = used + 1;
                    else
                        next(1) = next(1) - 1;
                    end
                end
            end
        end
    end
end

```

```

        next(2) = next(2) - 1;
        s = s + 1;
    end
    path = [path; next(1) next(2)];
    path = gostraight(next, finish, m, path);
end

elseif xdiff > ydiff % move ul-diagonal and up
    numdiag = ydiff;
    numstr = xdiff - ydiff;
    total = xdiff;

    if numdiag <= numstr % more up steps than ul-diagonal
        ratio = floor(total/numdiag);
        used = 0;
        s = 1;
        next(1) = start(1) - 1;
        next(2) = start(2);
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numdiag)
                next(2) = next(2) - 1;
                used = used + 1;
            end
            next(1) = next(1) - 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);

    elseif numdiag > numstr % more ul-diagonal steps than up
        ratio = floor(total/numstr);
        used = 0;
        s = 1;
        next(1) = start(1) - 1;
        next(2) = start(2) - 1;
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numstr)
                used = used + 1;
            else
                next(2) = next(2) - 1;
            end
            next(1) = next(1) - 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);
    end
end

elseif start(2) < finish(2) % up & right path
    xdiff = start(1) - finish(1);
    ydiff = finish(2) - start(2);

    if xdiff == ydiff % move ur-diagonal
        next(1) = start(1) - 1;
        next(2) = start(2) + 1;
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))

```

```

        path = [path; next(1) next(2)];
        next(1) = next(1) - 1;
        next(2) = next(2) + 1;
    end
    path = [path; next(1) next(2)];
    path = gostraight(next, finish, m, path);

elseif xdiff < ydiff          % move ur-diagonal and right
    numdiag = xdiff;
    numstr = ydiff - xdiff;
    total = ydiff;

    if numdiag <= numstr      % more right steps than ur-diagonal
        ratio = floor(total/numdiag);
        used = 0;
        s = 1;
        next(1) = start(1);
        next(2) = start(2) + 1;
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numdiag)
                next(1) = next(1) - 1;
                used = used + 1;
            end
            next(2) = next(2) + 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);

    elseif numdiag > numstr % more ur-diagonal steps than right
        ratio = floor(total/numstr);
        used = 0;
        s = 1;
        next(1) = start(1) - 1;
        next(2) = start(2) + 1;
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numstr)
                used = used + 1;
            else
                next(1) = next(1) - 1;
            end
            next(2) = next(2) + 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);
    end

elseif xdiff > ydiff          % move ur-diagonal and up
    numdiag = ydiff;
    numstr = xdiff - ydiff;
    total = xdiff;

    if numdiag <= numstr      % more up steps than ur-diagonal
        ratio = floor(total/numdiag);
        used = 0;
        s = 1;

```

```

        next(1) = start(1) - 1;
        next(2) = start(2);
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numdiag)
                next(2) = next(2) + 1;
                used = used + 1;
            end
            next(1) = next(1) - 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);

    elseif numdiag > numstr % more ur-diagonal steps than up
        ratio = floor(total/numstr);
        used = 0;
        s = 1;
        next(1) = start(1) - 1;
        next(2) = start(2) + 1;
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numstr)
                used = used + 1;
            else
                next(2) = next(2) + 1;
            end
            next(1) = next(1) - 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);
    end
end

end

elseif start(1) < finish(1)
    if start(2) > finish(2) % down & left path
        xdiff = finish(1) - start(1);
        ydiff = start(2) - finish(2);

        if xdiff == ydiff % move dl-diagonal
            next(1) = start(1) + 1;
            next(2) = start(2) - 1;
            while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
                path = [path; next(1) next(2)];
                next(1) = next(1) + 1;
                next(2) = next(2) - 1;
            end
            path = [path; next(1) next(2)];
            path = gostraight(next, finish, m, path);

        elseif xdiff < ydiff % move dl-diagonal and left
            numdiag = xdiff;
            numstr = ydiff - xdiff;
            total = ydiff;

            if numdiag <= numstr % more left steps than dl-diagonal
                ratio = floor(total/numdiag);

```

```

        used = 0;
        s = 1;
        next(1) = start(1);
        next(2) = start(2) - 1;
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numdiag)
                next(1) = next(1) + 1;
                used = used + 1;
            end
            next(2) = next(2) - 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);

    elseif numdiag > numstr % more dl-diagonal steps than left
        ratio = floor(total/numstr);
        used = 0;
        s = 1;
        next(1) = start(1) + 1;
        next(2) = start(2) - 1;
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numstr)
                used = used + 1;
            else
                next(1) = next(1) + 1;
            end
            next(2) = next(2) - 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);
    end

elseif xdiff > ydiff % move dl-diagonal and down
    numdiag = ydiff;
    numstr = xdiff - ydiff;
    total = xdiff;

    if numdiag <= numstr % more down steps than dl-diagonal
        ratio = floor(total/numdiag);
        used = 0;
        s = 1;
        next(1) = start(1) + 1;
        next(2) = start(2);
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numdiag)
                next(2) = next(2) - 1;
                used = used + 1;
            end
            next(1) = next(1) + 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);
    end
end

```



```

elseif numdiag > numstr % more dl-diagonal steps than down
    ratio = floor(total/numstr);
    used = 0;
    s = 1;
    next(1) = start(1) + 1;
    next(2) = start(2) - 1;
    while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
        path = [path; next(1) next(2)];
        if (mod(s, ratio) == 0) & (used < numstr)
            used = used + 1;
        else
            next(2) = next(2) - 1;
        end
        next(1) = next(1) + 1;
        s = s + 1;
    end
    path = [path; next(1) next(2)];
    path = gostraight(next, finish, m, path);
end

end

elseif start(2) < finish(2) % down & right path
    xdiff = finish(1) - start(1);
    ydiff = finish(2) - start(2);

    if xdiff == ydiff % move dr-diagonal
        next(1) = start(1) + 1;
        next(2) = start(2) + 1;
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            next(1) = next(1) + 1;
            next(2) = next(2) + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);

    elseif xdiff < ydiff % move dr-diagonal and right
        numdiag = xdiff;
        numstr = ydiff - xdiff;
        total = ydiff;

        if numdiag <= numstr % more right steps than dr-diagonal
            ratio = floor(total/numdiag);
            used = 0;
            s = 1;
            next(1) = start(1);
            next(2) = start(2) + 1;
            while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
                path = [path; next(1) next(2)];
                if (mod(s, ratio) == 0) & (used < numdiag)
                    next(1) = next(1) + 1;
                    used = used + 1;
                end
                next(2) = next(2) + 1;
                s = s + 1;
            end
            path = [path; next(1) next(2)];
            path = gostraight(next, finish, m, path);
        end
    end
end

```

```

elseif numdiag > numstr % more dr-diagonal steps than right
    ratio = floor(total/numstr);
    used = 0;
    s = 1;
    next(1) = start(1) + 1;
    next(2) = start(2) + 1;
    while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
        path = [path; next(1) next(2)];
        if (mod(s, ratio) == 0) & (used < numstr)
            used = used + 1;
        else
            next(1) = next(1) + 1;
        end
        next(2) = next(2) + 1;
        s = s + 1;
    end
    path = [path; next(1) next(2)];
    path = gostraight(next, finish, m, path);
end

elseif xdiff > ydiff % move dr-diagonal and down
    numdiag = ydiff;
    numstr = xdiff - ydiff;
    total = xdiff;

    if numdiag <= numstr % more down steps than dr-diagonal
        ratio = floor(total/numdiag);
        used = 0;
        s = 1;
        next(1) = start(1) + 1;
        next(2) = start(2);
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numdiag)
                next(2) = next(2) + 1;
                used = used + 1;
            end
            next(1) = next(1) + 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);
    end

    elseif numdiag > numstr % more dr-diagonal steps than down
        ratio = floor(total/numstr);
        used = 0;
        s = 1;
        next(1) = start(1) + 1;
        next(2) = start(2) + 1;
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numstr)
                used = used + 1;
            else
                next(2) = next(2) + 1;
            end
            next(1) = next(1) + 1;
            s = s + 1;
        end
    end
end

```

```

                                path = [path; next(1) next(2)];
                                path = gostraight(next, finish, m, path);
                                end
                            end
                        end
                    end

% if the path crosses another edge, ignore it:
[p, whatever] = size(path);
for q = 5:p-5
    value = m(path(q, 1), path(q, 2));
    leftvalue = m(path(q, 1), path(q, 2)-1);
    rightvalue = m(path(q, 1), path(q, 2)+1);
    upvalue = m(path(q, 1)-1, path(q, 2));
    downvalue = m(path(q, 1)+1, path(q, 2));
    if (value == 1) | (leftvalue == 1) | (rightvalue == 1) | (upvalue == 1) | (downvalue == 1) |
(value == 4) | (leftvalue == 4) | (rightvalue == 4) | (upvalue == 4) | (downvalue == 4)
        return;
    end
end

% check that the edge values of the points in the path do not vary too much:
thresh = 0.5;
for q = 1:p
    vals(q) = edgevals(path(q, 1), path(q, 2));
end
valstd = std(vals);
nvalstd = valstd/mean(vals);

% if the average is above the threshold, link the path:
if nvalstd < thresh
    newbranch = path(1, :);
    newlength = 0;

    for q = 1:p
        if m(path(q, 1), path(q, 2)) ~= 2
            m(path(q, 1), path(q, 2)) = 4;
        end

        if q > 1
            newbranch = [newbranch path(q, :)];
            newlength = newlength + 1;
        end
    end

    newbranch = [newbranch path(p, :)];
    newlength = newlength + 1;

    branchnum = branchnum + 1;
    branchlength(branchnum) = newlength;
    branches(branchnum, 1:2*(branchlength(branchnum)+1)) = newbranch;
end

```

% gostraight.m

% Used to finish up a path by going straight, either horizontally or vertically, to the endpoint. % Called from 'drawlink.m' and 'linepath.m'.

```

function path = gostraight(next, finish, m, path)

if next(1) < finish(1)
    while next(1) ~= finish(1)
        next(1) = next(1) + 1;
        if m(next(1), next(2)) ~= 2
            path = [path; next(1) next(2)];
        end
    end
elseif next(1) > finish(1)
    while next(1) ~= finish(1)
        next(1) = next(1) - 1;
        if m(next(1), next(2)) ~= 2
            path = [path; next(1) next(2)];
        end
    end
elseif next(2) < finish(2)
    while next(2) ~= finish(2)
        next(2) = next(2) + 1;
        if m(next(1), next(2)) ~= 2
            path = [path; next(1) next(2)];
        end
    end
elseif next(2) > finish(2)
    while next(2) ~= finish(2)
        next(2) = next(2) - 1;
        if m(next(1), next(2)) ~= 2
            path = [path; next(1) next(2)];
        end
    end
end
end

```

% **dirdiff.m**

% Finds the difference between two step directions, given in how many shifts there are between % them -
the best is if they are pointing in the exact opposite direction. Called from % 'checklink.m'.

```
function diff = dirdiff(dir1, dir2);
```

```

if dir1 == 1
    if dir2 == 1
        diff = 4;
    elseif dir2 == 2
        diff = 3;
    elseif dir2 == 3
        diff = 2;
    elseif dir2 == 4
        diff = 3;
    elseif dir2 == 5
        diff = 1;
    elseif dir2 == 6
        diff = 2;
    elseif dir2 == 7
        diff = 1;
    elseif dir2 == 8
        diff = 0;
    end
elseif dir1 == 2

```

```

    if dir2 == 1
        diff = 3;
    elseif dir2 == 2
        diff = 4;
    elseif dir2 == 3
        diff = 3;
    elseif dir2 == 4
        diff = 2;
    elseif dir2 == 5
        diff = 2;
    elseif dir2 == 6
        diff = 1;
    elseif dir2 == 7
        diff = 0;
    elseif dir2 == 8
        diff = 1;
    end
elseif dir1 == 3
    if dir2 == 1
        diff = 2;
    elseif dir2 == 2
        diff = 3;
    elseif dir2 == 3
        diff = 4;
    elseif dir2 == 4
        diff = 1;
    elseif dir2 == 5
        diff = 3;
    elseif dir2 == 6
        diff = 0;
    elseif dir2 == 7
        diff = 1;
    elseif dir2 == 8
        diff = 2;
    end
elseif dir1 == 4
    if dir2 == 1
        diff = 3;
    elseif dir2 == 2
        diff = 2;
    elseif dir2 == 3
        diff = 1;
    elseif dir2 == 4
        diff = 4;
    elseif dir2 == 5
        diff = 0;
    elseif dir2 == 6
        diff = 3;
    elseif dir2 == 7
        diff = 2;
    elseif dir2 == 8
        diff = 1;
    end
elseif dir1 == 5
    if dir2 == 1
        diff = 1;
    elseif dir2 == 2
        diff = 2;
    elseif dir2 == 3

```

```

        diff = 3;
elseif dir2 == 4
    diff = 0;
elseif dir2 == 5
    diff = 4;
elseif dir2 == 6
    diff = 1;
elseif dir2 == 7
    diff = 2;
elseif dir2 == 8
    diff = 3;
end
elseif dir1 == 6
    if dir2 == 1
        diff = 2;
    elseif dir2 == 2
        diff = 1;
    elseif dir2 == 3
        diff = 0;
    elseif dir2 == 4
        diff = 3;
    elseif dir2 == 5
        diff = 1;
    elseif dir2 == 6
        diff = 4;
    elseif dir2 == 7
        diff = 3;
    elseif dir2 == 8
        diff = 2;
    end
elseif dir1 == 7
    if dir2 == 1
        diff = 1;
    elseif dir2 == 2
        diff = 0;
    elseif dir2 == 3
        diff = 1;
    elseif dir2 == 4
        diff = 2;
    elseif dir2 == 5
        diff = 2;
    elseif dir2 == 6
        diff = 3;
    elseif dir2 == 7
        diff = 4;
    elseif dir2 == 8
        diff = 3;
    end
elseif dir1 == 8
    if dir2 == 1
        diff = 0;
    elseif dir2 == 2
        diff = 1;
    elseif dir2 == 3
        diff = 2;
    elseif dir2 == 4
        diff = 1;
    elseif dir2 == 5
        diff = 3;

```

```

elseif dir2 == 6
    diff = 2;
elseif dir2 == 7
    diff = 3;
elseif dir2 == 8
    diff = 4;
end
end

```

% linepairs.m

% Searches to the left and right of all branches for other roughly parallel branches. Draws points % in between paired branches. Called from 'mainnowave.m'.

```
function [pairs, newedge] = linepairs(edge, branches, branchlength, dirsiz, minwidth, maxwidth, angtol)
```

```

[branchnum, whatever] = size(branches);
pairs = [0 0 0];
newedge = edge;
dirsiz = 4;

```

```

for i = 1:branchnum
    length = branchlength(i) + 1;

    for j = 1:branchlength(i) + 1
        start = branches(i, 2*j-1:2*j);

        if j < (dirsiz + 1)
            sdirend1 = branches(i, 1:2);
            sdirend2 = branches(i, 2*(j+dirsiz)-1:2*(j+dirsiz));
        elseif j > branchlength(i)-(dirsiz-1)
            sdirend1 = branches(i, 2*(j-dirsiz)-1:2*(j-dirsiz));
            sdirend2 = branches(i, 2*(branchlength(i)+1)-1:2*(branchlength(i)+1));
        else
            sdirend1 = branches(i, 2*(j-dirsiz)-1:2*(j-dirsiz));
            sdirend2 = branches(i, 2*(j+dirsiz)-1:2*(j+dirsiz));
        end

        sdir = (atan((sdirend1(1)-sdirend2(1))/(sdirend1(2) sdirend2(2))))*(180/pi) + 90;

        if (sdir <= 68) & (sdir > 22)
            leftdir = 3;
            rightdir = 6;
        elseif (sdir <= 22) & (sdir > -23)
            leftdir = 2;
            rightdir = 7;
        elseif (sdir <= -23) & (sdir > -68)
            leftdir = 1;
            rightdir = 8;
        else
            leftdir = 4;
            rightdir = 5;
        end
    end
end

```

```

[pairs, newedge] = pairup(start, leftdir, sdir, i, j, dirsiz, minwidth, maxwidth, angtol,
branches, branchlength, edge, pairs, newedge);
[pairs, newedge] = pairup(start, rightdir, sdir, i, j, dirsiz, minwidth, maxwidth, angtol,
branches, branchlength, edge, pairs, newedge);

```

```

        end
    end
end

```

% pairup.m

% Called from 'linepairs.m'.

```

function [pairs, newedge] = pairup(start, stepdir, sdir, i, J, dirsiz, minwidth, maxwidth, angtol, branches,
branchlength, edge, pairs, newedge)

```

```

[branchnum, whatever] = size(branches);

```

```

a = 0;
begin = start;
while a == 0
    [a, finish] = takestep(begin, stepdir, edge);

```

```

    [la, lfinish] = takestep(finish, 4, edge);
    if la == 1
        a = 1;
        finish = lfinish;
    end

```

```

    [ra, rfinish] = takestep(finish, 5, edge);
    if ra == 1
        a = 1;
        finish = rfinish;
    end

```

```

    begin = finish;

```

```

end

```

```

if a == 1
    diff = sqrt((start(1) - finish(1))^2 + (start(2) - finish(2))^2);

```

```

    if (diff >= minwidth) & (diff <= maxwidth)
        targetx = finish(1);
        targety = finish(2);
        bn = i;
        pn = J;
        out1 = 0;
        for s = 1:branchnum
            for t = 1:branchlength(s)+1
                if (branches(s, 2*t-1) == targetx) & (branches(s, 2*t) == targety)
                    bn = s;
                    pn = t;
                    out1 = 1;
                    break
                end
            end
        end
        if out1 == 1
            break
        end
    end

```

```

end

```

```

if pn < dirsiz + 1
    fdirend1 = branches(bn, 1:2);
    fdirend2 = branches(bn, 2*(pn+dirsiz)-1:2*(pn+dirsiz));

```



```

elseif pn > branchlength(bn)-(dirsize-1)
    fdirend1 = branches(bn, 2*(pn-dirsize)-1:2*(pn-dirsize));
    fdirend2 = branches(bn, 2*(branchlength(bn)+1)-1:2*(branchlength(bn)+1));
else
    fdirend1 = branches(bn, 2*(pn-dirsize)-1:2*(pn-dirsize));
    fdirend2 = branches(bn, 2*(pn+dirsize)-1:2*(pn+dirsize));
end
fdir = (atan((fdirend1(1)-fdirend2(1))/(fdirend1(2)-fdirend2(2))))*(180/pi) + 90;

v = abs(sdir - fdir);
if v > 90
    v = abs((sdir + 180) - fdir);
end
if v > 90
    v = abs(sdir - (fdir + 180));
end

if v < angtol
    [pr, whatever] = size(pairs);
    out2 = 0;
    for s = 1:pr
        x = pairs(s, 1);
        y = pairs(s, 2);
        if [x y] == [i bn]
            pairs(s, 3) = pairs(s, 3) + 1;
            out2 = 1;
        end
    end
    if out2 == 0
        pairs = [pairs; i bn 1];
    end

    path = linepath(start, finish, newedge);
    [pth, whatever] = size(path);
    midpath = ceil(pth/2);
    newedge(path(midpath, 1), path(midpath, 2)) = 2;
end
end
end
end

```

% linepath.m

% Similar to 'drawlink.m' but only returns the path of the line. Called from 'pairup.m'.

function path = linepath(start, finish, m)

path = start;

if (start(1) == finish(1)) & (start(2) == finish(2))

return;

elseif start(1) == finish(1) % horizontal path

next(1) = start(1);

if start(2) > finish(2) % move to the left

next(2) = start(2) - 1;

while next(2) ~= finish(2)

path = [path; next(1) next(2)];

next(2) = next(2) - 1;

```

        end
        path = [path; next(1) next(2)];

    else % move to the right
        next(2) = start(2) + 1;
        while next(2) ~= finish(2)
            path = [path; next(1) next(2)];
            next(2) = next(2) + 1;
        end
        path = [path; next(1) next(2)];
    end

elseif start(2) == finish(2) % vertical path
    next(2) = start(2);

    if start(1) > finish(1) % move up
        next(1) = start(1) - 1;
        while next(1) ~= finish(1)
            path = [path; next(1) next(2)];
            next(1) = next(1) - 1;
        end
        path = [path; next(1) next(2)];

    else % move down
        next(1) = start(1) + 1;
        while next(1) ~= finish(1)
            path = [path; next(1) next(2)];
            next(1) = next(1) + 1;
        end
        path = [path; next(1) next(2)];
    end

elseif start(1) > finish(1)
    if start(2) > finish(2) % up & left path
        xdiff = start(1) - finish(1);
        ydiff = start(2) - finish(2);

        if xdiff == ydiff % move ul-diagonal
            next(1) = start(1) - 1;
            next(2) = start(2) - 1;
            while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
                path = [path; next(1) next(2)];
                next(1) = next(1) - 1;
                next(2) = next(2) - 1;
            end
            path = [path; next(1) next(2)];
            path = gostraight(next, finish, m, path);

        elseif xdiff < ydiff % move ul-diagonal and left
            numdiag = xdiff;
            numstr = ydiff - xdiff;
            total = ydiff;

            if numdiag <= numstr % more left steps than ul-diagonal
                ratio = floor(total/numdiag);
                used = 0;
                s = 1;
            end
        end
    end
end

```

```

next(1) = start(1);
next(2) = start(2) - 1;
while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
    path = [path; next(1) next(2)];
    if (mod(s, ratio) == 0) & (used < numdiag)
        next(1) = next(1) - 1;
        used = used + 1;
    end
    next(2) = next(2) - 1;
    s = s + 1;
end
path = [path; next(1) next(2)];
path = gostraight(next, finish, m, path);

elseif numdiag > numstr % more ul-diagonal steps than left
    ratio = floor(total/numstr);
    used = 0;
    s = 1;
    next(1) = start(1) - 1;
    next(2) = start(2) - 1;
    while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
        path = [path; next(1) next(2)];
        if (mod(s, ratio) == 0) & (used < numstr)
            used = used + 1;
        else
            next(1) = next(1) - 1;
        end
        next(2) = next(2) - 1;
        s = s + 1;
    end
    path = [path; next(1) next(2)];
    path = gostraight(next, finish, m, path);
end

elseif xdiff > ydiff % move ul-diagonal and up
    numdiag = ydiff;
    numstr = xdiff - ydiff;
    total = xdiff;

    if numdiag <= numstr % more up steps than ul-diagonal
        ratio = floor(total/numdiag);
        used = 0;
        s = 1;
        next(1) = start(1) - 1;
        next(2) = start(2);
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numdiag)
                next(2) = next(2) - 1;
                used = used + 1;
            end
            next(1) = next(1) - 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);
    end

    elseif numdiag > numstr % more ul-diagonal steps than up
        ratio = floor(total/numstr);

```

```

        used = 0;
        s = 1;
        next(1) = start(1) - 1;
        next(2) = start(2) - 1;
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numstr)
                used = used + 1;
            else
                next(2) = next(2) - 1;
            end
            next(1) = next(1) - 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);
    end
end

elseif start(2) < finish(2) % up & right path
    xdiff = start(1) - finish(1);
    ydiff = finish(2) - start(2);

    if xdiff == ydiff % move ur-diagonal
        next(1) = start(1) - 1;
        next(2) = start(2) + 1;
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            next(1) = next(1) - 1;
            next(2) = next(2) + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);
    end

    elseif xdiff < ydiff % move ur-diagonal and right
        numdiag = xdiff;
        numstr = ydiff - xdiff;
        total = ydiff;

        if numdiag <= numstr % more right steps than ur-diagonal
            ratio = floor(total/numdiag);
            used = 0;
            s = 1;
            next(1) = start(1);
            next(2) = start(2) + 1;
            while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
                path = [path; next(1) next(2)];
                if (mod(s, ratio) == 0) & (used < numdiag)
                    next(1) = next(1) - 1;
                    used = used + 1;
                end
                next(2) = next(2) + 1;
                s = s + 1;
            end
            path = [path; next(1) next(2)];
            path = gostraight(next, finish, m, path);
        end

        elseif numdiag > numstr % more ur-diagonal steps than right
            ratio = floor(total/numstr);

```

```

        used = 0;
        s = 1;
        next(1) = start(1) - 1;
        next(2) = start(2) + 1;
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numstr)
                used = used + 1;
            else
                next(1) = next(1) - 1;
            end
            next(2) = next(2) + 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);
    end

elseif xdiff > ydiff % move ur-diagonal and up
    numdiag = ydiff;
    numstr = xdiff - ydiff;
    total = xdiff;

    if numdiag <= numstr % more up steps than ur-diagonal
        ratio = floor(total/numdiag);
        used = 0;
        s = 1;
        next(1) = start(1) - 1;
        next(2) = start(2);
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numdiag)
                next(2) = next(2) + 1;
                used = used + 1;
            end
            next(1) = next(1) - 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);
    end

elseif numdiag > numstr % more ur-diagonal steps than up
    ratio = floor(total/numstr);
    used = 0;
    s = 1;
    next(1) = start(1) - 1;
    next(2) = start(2) + 1;
    while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
        path = [path; next(1) next(2)];
        if (mod(s, ratio) == 0) & (used < numstr)
            used = used + 1;
        else
            next(2) = next(2) + 1;
        end
        next(1) = next(1) - 1;
        s = s + 1;
    end
    path = [path; next(1) next(2)];
    path = gostraight(next, finish, m, path);
end

```

```

end
end
end
elseif start(1) < finish(1)
    if start(2) > finish(2) % down & left path
        xdiff = finish(1) - start(1);
        ydiff = start(2) - finish(2);

        if xdiff == ydiff % move dl-diagonal
            next(1) = start(1) + 1;
            next(2) = start(2) - 1;
            while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
                path = [path; next(1) next(2)];
                next(1) = next(1) + 1;
                next(2) = next(2) - 1;
            end
            path = [path; next(1) next(2)];
            path = gostraight(next, finish, m, path);

        elseif xdiff < ydiff % move dl-diagonal and left
            numdiag = xdiff;
            numstr = ydiff - xdiff;
            total = ydiff;

            if numdiag <= numstr % more left steps than dl-diagonal
                ratio = floor(total/numdiag);
                used = 0;
                s = 1;
                next(1) = start(1);
                next(2) = start(2) - 1;
                while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
                    path = [path; next(1) next(2)];
                    if (mod(s, ratio) == 0) & (used < numdiag)
                        next(1) = next(1) + 1;
                        used = used + 1;
                    end
                    next(2) = next(2) - 1;
                    s = s + 1;
                end
                path = [path; next(1) next(2)];
                path = gostraight(next, finish, m, path);

            elseif numdiag > numstr % more dl-diagonal steps than left
                ratio = floor(total/numstr);
                used = 0;
                s = 1;
                next(1) = start(1) + 1;
                next(2) = start(2) - 1;
                while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
                    path = [path; next(1) next(2)];
                    if (mod(s, ratio) == 0) & (used < numstr)
                        used = used + 1;
                    else
                        next(1) = next(1) + 1;
                    end
                    next(2) = next(2) - 1;
                    s = s + 1;
                end
            end
        end
    end
end

```

```

        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);
    end

elseif xdiff > ydiff % move dl-diagonal and down
    numdiag = ydiff;
    numstr = xdiff - ydiff;
    total = xdiff;

    if numdiag <= numstr % more down steps than dl-diagonal
        ratio = floor(total/numdiag);
        used = 0;
        s = 1;
        next(1) = start(1) + 1;
        next(2) = start(2);
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numdiag)
                next(2) = next(2) - 1;
                used = used + 1;
            end
            next(1) = next(1) + 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);

    elseif numdiag > numstr % more dl-diagonal steps than down
        ratio = floor(total/numstr);
        used = 0;
        s = 1;
        next(1) = start(1) + 1;
        next(2) = start(2) - 1;
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numstr)
                used = used + 1;
            else
                next(2) = next(2) - 1;
            end
            next(1) = next(1) + 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);
    end

end

elseif start(2) < finish(2) % down & right path
    xdiff = finish(1) - start(1);
    ydiff = finish(2) - start(2);

    if xdiff == ydiff % move dr-diagonal
        next(1) = start(1) + 1;
        next(2) = start(2) + 1;
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            next(1) = next(1) + 1;
            next(2) = next(2) + 1;
        end
    end
end

```

```

end
path = [path; next(1) next(2)];
path = gostraight(next, finish, m, path);

elseif xdiff < ydiff          % move dr-diagonal and right
    numdiag = xdiff;
    numstr = ydiff - xdiff;
    total = ydiff;

    if numdiag <= numstr      % more right steps than dr-diagonal
        ratio = floor(total/numdiag);
        used = 0;
        s = 1;
        next(1) = start(1);
        next(2) = start(2) + 1;
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numdiag)
                next(1) = next(1) + 1;
                used = used + 1;
            end
            next(2) = next(2) + 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);

    elseif numdiag > numstr % more dr-diagonal steps than right
        ratio = floor(total/numstr);
        used = 0;
        s = 1;
        next(1) = start(1) + 1;
        next(2) = start(2) + 1;
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
            path = [path; next(1) next(2)];
            if (mod(s, ratio) == 0) & (used < numstr)
                used = used + 1;
            else
                next(1) = next(1) + 1;
            end
            next(2) = next(2) + 1;
            s = s + 1;
        end
        path = [path; next(1) next(2)];
        path = gostraight(next, finish, m, path);
    end

elseif xdiff > ydiff          % move dr-diagonal and down
    numdiag = ydiff;
    numstr = xdiff - ydiff;
    total = xdiff;

    if numdiag <= numstr      % more down steps than dr-diagonal
        ratio = floor(total/numdiag);
        used = 0;
        s = 1;
        next(1) = start(1) + 1;
        next(2) = start(2);
        while (next(1) ~= finish(1)) & (next(2) ~= finish(2))

```



```

        path = [path; next(1) next(2)];
        if (mod(s, ratio) == 0) & (used < numdiag)
            next(2) = next(2) + 1;
            used = used + 1;
        end
        next(1) = next(1) + 1;
        s = s + 1;
    end
    path = [path; next(1) next(2)];
    path = gostraight(next, finish, m, path);

elseif numdiag > numstr % more dr-diagonal steps than down
    ratio = floor(total/numstr);
    used = 0;
    s = 1;
    next(1) = start(1) + 1;
    next(2) = start(2) + 1;
    while (next(1) ~= finish(1)) & (next(2) ~= finish(2))
        path = [path; next(1) next(2)];
        if (mod(s, ratio) == 0) & (used < numstr)
            used = used + 1;
        else
            next(2) = next(2) + 1;
        end
        next(1) = next(1) + 1;
        s = s + 1;
    end
    path = [path; next(1) next(2)];
    path = gostraight(next, finish, m, path);
end
end
end
end
end
end

```

% mainwave.m

% Serves as the main program when wavelets are not used.

% Get the aerial image:
getimage

% Get values from user that will be inputted into functions later:
dlevel = input('What decomposition level should be used? ');
glevel = input('What block size for grayscale variation? ');

numdirs = input('How many directions should be polled to determine primary and secondary directions? ');
tooshort = input('How short should the shortest edge kept be? Any edge shorter than this will be deleted ');
maxgap = input('How long should the search area be? ');
maxsoc = input('How wide should the search area be? ');

dirsize = floor((input('How long an edge piece should be used to determine the local direction? '))/2);
minwidth = input('What is the minimum width of the roads? ');
maxwidth = input('What is the maximum width of the roads? ');
angtol = input('What is the maximum difference in direction for two edges to be paired? ');

% Get angles and edge values for each pixel:
[ang, edgevals] = getdirections(data);

```

% Pick the edge points and thin:
thinedge = waveedgepts(data, dlevel, glevel);

% First link, then pair:
[linkedge, branches, branchlength] = link(thinedge, ang, edgevals, numdirs, tooshort, maxgap, maxsoc);
[pairs, pairedge] = linepairs(linkedge, branches, branchlength, dirsiz, minwidth, maxwidth, angtol);

```

```

% waveedgepts.m

% Finds the edge points of an image using the Mallat-Zhong code (local maxima) combined with % gray
scale variation. 'dlevel' is the level of decomposition used to get the edge points - must % be 1, 2, 3, 4, or
5. 'glevel' is the size of the neighborhood used in grayvar.m - must be odd. % Called from 'mainwave.m'

function edge = waveedgepts(data, dlevel, glevel)

h = [0.125 0.375 0.375 0.125];
g = [-2 2];
l = [1.50 1.12 1.03 1.01 1.00];

[w11, w12, s1] = doconvs(data, h, g, l, 1);
[m1, a1] = modphase(w11, w12);

if dlevel == 1
    lm = localmax4(m1, a1);
else
    [w21, w22, s2] = doconvs(s1, h, g, l, 2);
    [m2, a2] = modphase(w21, w22);

    if dlevel == 2
        lm = localmax4(m2, a2);
    else
        [w31, w32, s3] = doconvs(s2, h, g, l, 3);
        [m3, a3] = modphase(w31, w32);

        if dlevel == 3
            lm = localmax4(m3, a3);
        else
            [w41, w42, s4] = doconvs(s3, h, g, l, 4);
            [m4, a4] = modphase(w41, w42);

            if dlevel == 4
                lm = localmax4(m4, a4);
            else
                [w51, w52, s5] = doconvs(s4, h, g, l, 5);
                [m5, a5] = modphase(w51, w52);

                lm = localmax4(m5, a5);
            end
        end
    end
end

end

tg = grayvar(data, glevel);

[x, y] = size(lm);
edge = zeros(x, y);

```

```

for i = 2:x-1
for j = 2:y-1
    if lm(i, j) == 1
        tmp = tgv(i-1:i+1, j-1:j+1);
        yon = 0;

        for s = 1:3
        for t = 1:3
            if tmp(s, t) == 0
                yon = 1;
            end
        end
        end
        end

        if yon == 1
            edge(i, j) = 1;
        end
    end
end
end

```

% **doconvs.m**

% This takes the smoothed image at level j and does the convolutions so get the smoothed % image,
the horizontal edges, and the vertical edges at level j+1. Called from % 'waveedgepts.m'.

```
function [w1, w2, snw] = doconvs(sold, h, g, l, level)
```

```

[n, m] = size(sold);
lh = length(h);
lg = length(g);

```

```

w1 = zeros(n, m+lg-1);
w2 = zeros(n+lg-1, m);
smid = zeros(n, m+lh-1);
snw = zeros(n+lh-1, m+lh-1);

```

```

for i = 1:n
    w1(i, :) = (1/l(level))*conv2(sold(i, :), g);
    smid(i, :) = conv2(sold(i, :), h);
end

```

```

for i = 1:m
    w2(:, i) = (1/l(level))*conv2(sold(:, i), g');
end

```

```

for i = 1:(m+lh-1)
    snw(:, i) = conv2(smid(:, i), h');
end

```

```

w1 = w1(1:500, 1:500);
w2 = w2(1:500, 1:500);
snw = snw(2:501, 2:501);

```

% **modphase.m**

% Finds the modulus image and the phase image at level j. Called from 'waveedgepts.m'.

```
function [m, a] = modphase(w1, w2)
```

```
m = sqrt(abs(w1).^2 + abs(w2).^2);
a = angle(w1 + i*w2);
```

```
% localmax4.m
```

```
% Finds the local maximums in the modulus images. Called from 'waveedgepts.m'.
```

```
function lm = localmax4(m, a)
```

```
H = 1;
D1 = 2;
V = 3;
D2 = 4;
```

```
[x, y] = size(a);
lm = zeros(x, y);
```

```
for i = 5:x-4
```

```
for j = 5:y-4
```

```
    if ((a(i, j) <= 7*pi/8) & (a(i, j) > 5*pi/8))
```

```
        % diagonal 1
```

```
        if ((m(i, j) > m(i+1, j-1)) & (m(i, j) > m(i-1, j+1)) & (m(i, j) > m(i+2, j-2)) & (m(i, j) >
m(i-2, j+2)) & (m(i, j) > m(i+3, j-3)) & (m(i, j) > m(i-3, j+3)) & (m(i, j) > m(i+4, j-4)) & (m(i, j) > m(i-4,
j+4)))
```

```
            lm(i, j) = 1;
```

```
        end
```

```
    elseif ((a(i, j) <= 5*pi/8) & (a(i, j) > 3*pi/8))
```

```
        % horizontal
```

```
        if ((m(i, j) > m(i-1, j)) & (m(i, j) > m(i+1, j)) & (m(i, j) > m(i-2, j)) & (m(i, j) > m(i+2, j))
& (m(i, j) > m(i-3, j)) & (m(i, j) > m(i+3, j)) & (m(i, j) > m(i-4, j)) & (m(i, j) > m(i+4, j)))
```

```
            lm(i, j) = 1;
```

```
        end
```

```
    elseif ((a(i, j) <= 3*pi/8) & (a(i, j) > pi/8))
```

```
        % diagonal 2
```

```
        if ((m(i, j) > m(i-1, j-1)) & (m(i, j) > m(i+1, j+1)) & (m(i, j) > m(i-2, j-2)) & (m(i, j) >
m(i+2, j+2)) & (m(i, j) > m(i-3, j-3)) & (m(i, j) > m(i+3, j+3)) & (m(i, j) > m(i-4, j-4)) & (m(i, j) > m(i+4,
j+4)))
```

```
            lm(i, j) = 1;
```

```
        end
```

```
    elseif ((a(i, j) <= -pi/8) & (a(i, j) > -3*pi/8))
```

```
        % diagonal 1
```

```
        if ((m(i, j) > m(i+1, j-1)) & (m(i, j) > m(i-1, j+1)) & (m(i, j) > m(i+2, j-2)) & (m(i, j) >
m(i-2, j+2)) & (m(i, j) > m(i+3, j-3)) & (m(i, j) > m(i-3, j+3)) & (m(i, j) > m(i+4, j-4)) & (m(i, j) > m(i-4,
j+4)))
```

```
            lm(i, j) = 1;
```

```
        end
```

```
    elseif ((a(i, j) <= -3*pi/8) & (a(i, j) > -5*pi/8))
```

```
        % horizontal
```

```
        if ((m(i, j) > m(i-1, j)) & (m(i, j) > m(i+1, j)) & (m(i, j) > m(i-2, j)) & (m(i, j) > m(i+2, j))
& (m(i, j) > m(i-3, j)) & (m(i, j) > m(i+3, j)) & (m(i, j) > m(i-4, j)) & (m(i, j) > m(i+4, j)))
```

```
            lm(i, j) = 1;
```

```
        end
```

```
    elseif ((a(i, j) <= -5*pi/8) & (a(i, j) > -7*pi/8))
```

```
        % diagonal 2
```

```

        if ((m(i, j) > m(i-1, j-1)) & (m(i, j) > m(i+1, j+1)) & (m(i, j) > m(i-2, j-2)) & (m(i, j) >
m(i+2, j+2)) & (m(i, j) > m(i-3, j-3)) & (m(i, j) > m(i+3, j+3)) & (m(i, j) > m(i-4, j-4)) & (m(i, j) > m(i+4,
j+4)))
            lm(i, j) = 1;
        end
    else
        % vertical
        if ((m(i, j) > m(i, j-1)) & (m(i, j) > m(i, j+1)) & (m(i, j) > m(i, j-2)) & (m(i, j) > m(i, j+2))
& (m(i, j) > m(i, j-3)) & (m(i, j) > m(i, j+3)) & (m(i, j) > m(i, j-4)) & (m(i, j) > m(i, j+4)))
            lm(i, j) = 1;
        end
    end
end
end
end

```

% **grayvar.m**

% Does something with the variance of grayscale levels in an image 'm'. 'w' is the size of the % block
used. Called from 'waveedgepts.m'.

```
function tgv = grayvar(m, w)
```

```
[x, y] = size(m);
gv = zeros(x, y);
```

```
hw = (w - 1)/2;
```

```
for i = (1+hw):(x-hw)
for j = (1+hw):(y-hw)
    lm = m(i-hw:i+hw, j-hw:j+hw);
    me1 = mean(mean(lm));
    lgv = 0;
    for s = 1:w
        for t = 1:w
            lgv = lgv + (me1 - lm(s, t))^2;
        end
    end
    gv(i, j) = lgv;
end
end

```

```
end
end
```

```
me2 = mean(mean(gv));
ngv = gv/me2;
```

```
tgv = zeros(size(ngv));
```

```
for i = 1:x
for j = 1:y
    if ngv(i, j) > 1
        tgv(i, j) = 0;
    else
        tgv(i, j) = 1;
    end
end
end

```

```
end
end
```
