CODEN: LUTFD2/(TFRT-5422)/1-38/(1990)

Investigations of the Potential Theory Model for Image Interpolation

Martin Pålsson

Department of Automatic Control
Lund Institute of Technology
February 1990

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
Master Thesis

Date of issue
February 1990

Document Number
CODEN:LUTFD2/(TFRT-5422)/1-38(1990)

Author(s)
Martin Pålsson

Supervisors
Lars Nielsen, Gunnar Sparr

Sponsoring organisation

Title and subtitle
Investigations of the Potential Theory Model for Image Interpolation

Abstract

Nowadays there are many sources for electronic images. Besides the CCD-cameras there are sources like scanners, satellites and computer tomography equipments. Consequently, there is a growing need for making papercopies of all kinds of electronic images. Making papercopies introduces the need of interpolation, since most of the electronic images do not have a resolution that is sufficient for making high quality papercopies. The critical part of an interpolation seems to be the treatment of the edges. This work builds upon a representation where edges are treated separately, and then in some way added to the rest of the image. The problem then falls into two subproblems. How to find the edges and how to make the interpolation.

This report is mainly devoted to the problem which involves edge detection as a preprocess for image interpolation. An algorithm based on potential theory is used on the edges. Some problems have been identified and some solutions to them are suggested. The main problem seems to be that almost all images are a bit blurred, and the chosen algorithm, in its current form, is not intended to handle such images. Some different interpolation schemes have been considered. It seems that a scheme based on bilinear interpolation, modified to preserve the edges, is an efficient way to make interpolations.

Key words
image interpolation, edge detection, potential theory

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
38

Recipient's notes

Security classification

Acknowledgements

This work had not been possible without the help from a lot of people. First of all I would like to thank my two supervisors: Lars Nielsen, who has always offered me a helping hand, and Gunnar Sparr who has been an inexhaustible source of good ideas. Anders Hansson has also contributed with some ideas besides those in his master thesis. Some contributions to the work have also been made by Carl-Gustav Werner.

When writing the program Dag Brück has really been to great help concerning programming problems. I am also indebted to Leif Andersson, who has solved a lot of practical problems concerning the computers. He has also been to great help when transferring data to the ink jet plotter. I would like to thank Johan Nilsson, at the Department of Electrical Measurements, for helping me print images on the ink jet plotter.

A lot of other people have also been to great help. Concerning images of any kind, Mats Lilja is the guru. Kjell Gustafsson has proved that he knows a lot about the UNIX operating system. Finally, I would like to thank Bo Bernhardsson, Jan-Eric Larsson, Sven-Erik Mattsson and Bernt Nilsson for being helpful and encouraging.

Lund February 1990

Martin Pålsson

Table of Contents

1. Introduction

The field of high quality printing of electronic images is currently in fast progress. It seems that the well known photographic techniques will get competition from the new electronic imaging techniques. During the last years several electronic cameras, which deliver the pictures on some kind of digital storage, have been introduced. One of the keys to this technique is the development of the CCD-chip, which produces a discrete image with a limited resolution. Other examples on sources for electronic images are scanners and computer tomography equipments. The resolution may suit a TV-screen, but is not sufficient for papercopies or for enlarging. Such applications use interpolation, since they need the values at points between the points in the coarse original grid. Satellite image-reproduction is another tempting area, that is in need of interpolation. Here one needs to magnify the images, or to make corrections by means of resampling. In this area there is a need for good and rapid algorithms.

It is quite obvious that the simple approach of copying each point in the coarse grid into several points in the fine grid will not give a satisfactory result, but for some few applications. This pixel-replication method introduces disturbing edges in the image. The second approach, which partly avoids this effect, is bilinear interpolation. The values of the pixels in the fine grid are then obtained by a two-dimensional linear interpolation in the coarse grid. This method gives a good result for images without sharp edges. Sharp edges however get blurred, since they are spread out over several pixels. Consequently, images with sharp edges need more sophisticated interpolation methods. An algorithm, based on potential theory, has been proposed in a master thesis by Hansson (1989). Some experimental results were presented. However the results were not fully satisfactory.

One purpose of this thesis has been to make an implementation of the algorithm that supports further development. A large problem was the long execution times for the program that Hansson used. The new implementation considerably reduces the execution time. Using this implementation, the properties of the algorithm have been explored. The main problem was the treatment of edges. Consequently, this thesis has been mainly devoted to this problem. A thorough investigation of the edge detection algorithm has been made and some improvements are suggested. A new scheme for interpolation is also proposed. Chapter 2 is an introduction to the concepts behind this kind of image interpolation. The third chapter is devoted to the edge detection problem. Chapter 4 discusses the interpolation methods and presents some experimental results. A presentation of the program can be found in the appendices.

2. Basic Concepts

This master thesis is based on the hypothesis that to achieve a good result from an interpolation it is necessary preserve the edges. Other parts of the image, that may contain diffuse edges or slow variations in intensity, are less critical.

The ideas, underlying this interpolation algorithm, are borrowed from the potential theory. The intensity in the image is interpreted as a potential, caused by distributions of charges. Then edges correspond to curves with a dipole layer. The interpolation should preserve these dipole curves. A way of doing this is to interpolate the edges in a sourceplane. The edges may then be added to the rest of the image, that have been interpolated by some less sophisticated method.

2.1 Analytic Formulation of an Image

From a mathematical point of view the intensities of an image can be considered as a function of two variables, $f(x), x \in R^2$, which is given the value 0 outside a bounded set Ω (= the image). Suppose that f is twice continuously differentiable, except for at a finite number of smooth curves, γ_i, where either f or its first derivatives has a finite discontinuity. Assume that the curves γ_i are non-intersecting. Let $\{\Delta f(y)\}$ denote the piecewise continuous function obtained by pointwise application of the Laplacian. Let $(f)_\pm$ denote the jump in the function values across γ_i, and let in the same way $(\partial f/\partial n)_\pm$ denote the jump in the normal derivative. Then holds

$$
\begin{aligned}
f(x) = & \int_\Omega \{\Delta f(y)\}\, g(x-y)dy \\
& + \sum_i \int_{\gamma_i} \left(\frac{\partial f}{\partial n}\right)_\pm (y) g(x-y) ds_y \\
& - \sum_i \int_{\gamma_i} (f)_\pm (y) \frac{\partial g}{\partial n}(x-y) ds_y
\end{aligned} \tag{2.1}
$$

where the two latter terms are curve integrals (Kellogg, 1954, p.219 ff; Sparr, 1984, p5.7 f). The function $g(x) = \frac{1}{2\pi}\log|x|$ is the fundamental solution to the Laplace operator $\Delta = \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2}$ that is:

$$f = g * \Delta f$$

In an electrostatic analogy, comparing with Poisson's equation $\Delta u = -\rho$, the three terms have the following interpretations:

- The first term is the potential caused by a piecewise continuous distribution of charges $\rho = -\{\Delta f\}$ (which may have discontinuities on $\cup\gamma_i$).
- The integral in the second term is the potential caused by a curve γ_i, covered with a single layer of charges of density $-\left(\frac{\partial f}{\partial n}\right)_\pm$.

- The integral in the third term is the potential caused by a curve γ_i, covered with a double layer of charges (i.e. dipoles) with a moment density $(f)_{\pm}$ in the normal direction of γ_i.

(Admitting discontinuity curves to intersect, one would also have to consider contributions from multipoles.) These parts of the function corresponds to the following visual effects:

- *background or illumination*, caused by the piecewise continuous charge distribution.
- *diffuse edges or shades*, caused by the "charge lines".
- *sharp edges*, caused by the "dipole lines".

The algorithm developed will assume that the contributions from the second term in equation 2.1 is negligible. This means that diffuse edges and shades should not be present in the images. This is a quite hard constraint, since explorations have shown that most images are blurred, and blurred edges correspond to the second term. Since the second and first terms are treated jointly in the algorithm, blurred images will give bad results. However, for images that contains no blurred edges a good result can be expected.

2.2 The Images

The original image is represented by pixels placed in a coarse grid. Interpolating the image transforms the coarse grid into a finer one. Some of the testimages are mondrian since the edge detection algorithm shall give optimal results for such images. Speaking in mathematical terms a mondrian image is a two-dimensional stepfunction. This means that the intensity is constant except at a limited number of curves where the image contains edges.

3. Edge Detection as a Preprocess for Interpolation

Since the treatment of edges is important to achieve a good result of an interpolation, the method for edge detection is a critical part of an edge-preserving interpolation-scheme. When a Laplace template is applied to an image we get a pattern where nonzero values indicate the presence of edges. The Laplace-image is treated as a two-dimensional array of squares rather than an array of pixels. By looking at a square it is possible to obtain information of an edge. It is possible to estimate its height, direction and also its behavior in a neighborhood of the square.

3.1 The Laplacian

A digital image is represented in a discrete form, so a suitable discretisation of the Laplacian has to be found. There are a number of possible ways to discretisize it. The following one has been used in this work:

$$\Delta = \begin{matrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{matrix}$$

Other possible choices of are:

$$\begin{matrix} 1 & 0 & 1 \\ 0 & -4 & 0 \\ 1 & 0 & 1 \end{matrix} \qquad \begin{matrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{matrix}$$

All the three alternatives give similar elementary cases for edges. However for the two latter choices, edges give contributions to larger areas than the first one. Edges get, so to say, spread out in the Laplace-image. This may complicate the edge detection. Therefore it is not obvious that the latter templates would give better result than the first one.

3.2 The Laplace-Image

Suppose the discretisized Laplacian is applied to the bilevel image in Figure 3.1. Then the image shown in Figure 3.2 is obtained. The Laplace-image is interesting since various patterns of numbers, not equal to zero, gather around the edges. The patterns depend on the direction of the edge, and the magnitude of the values depend on the height of the edge. In Figure 3.2 the same pattern can be found in several squares. Sometimes a rotation or reflection of a square will result in a pattern that can be found elsewhere. It is obvious that there are a limited number of possible patterns in a Laplace-image, when the original image only contain edges of unity height. This leeds to a division of the different patterns into elementary cases.

0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	1	1	1	1	1	1	1	0	0	0	0	0
0	0	1	1	1	1	1	1	1	1	0	0	0	0
0	0	1	1	1	1	1	1	1	1	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	0	0	0
0	0	0	0	0	0	1	1	1	1	1	0	0	0
0	0	0	0	0	0	0	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 3.1 Original bilevel image.

0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	0	0	0	0	0
0	0	0	1	2	−2	−2	2	0	0	0	0	0	0
0	0	1	−2	−1	0	0	−2	2	0	0	0	0	0
0	0	2	−1	0	0	0	0	−2	1	0	0	0	0
0	1	−2	0	0	0	0	0	−1	2	0	0	0	0
0	1	−1	0	0	0	0	0	0	−2	1	0	0	0
0	1	−2	0	0	0	0	0	0	−1	2	0	0	0
0	0	2	−2	−1	−1	0	0	0	0	−2	1	0	0
0	0	0	1	1	2	−2	0	0	0	−1	2	0	0
0	0	0	0	0	0	2	−2	0	0	0	−2	1	0
0	0	0	0	0	0	0	2	−2	−1	−1	−2	1	0
0	0	0	0	0	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 3.2 The Laplace-Image.

3.3 The Elementary Cases

Choosing the Laplacian as in Section 3.1 some of the possible cases for edges are shown in Figure 3.3. This figure illustrates the seven elementary cases. All other cases can be created by rearranging the vertices in Figure 3.3, or through multiplication by -1. This makes a total of 68 different patterns. Note that the cases are chosen so that edges in adjacent squares will fit to each other. These cases cover neither thin lines (that are one pixel wide), nor intersecting edges. Neither does the catalog cover other strange formations, like U-turns around a single pixels. Consequently, if thin lines or intersections are present in an image, they will not be detected. *Thus a good result for the interpolation of such images can not be expected.* Actually, such patterns will be interpolated

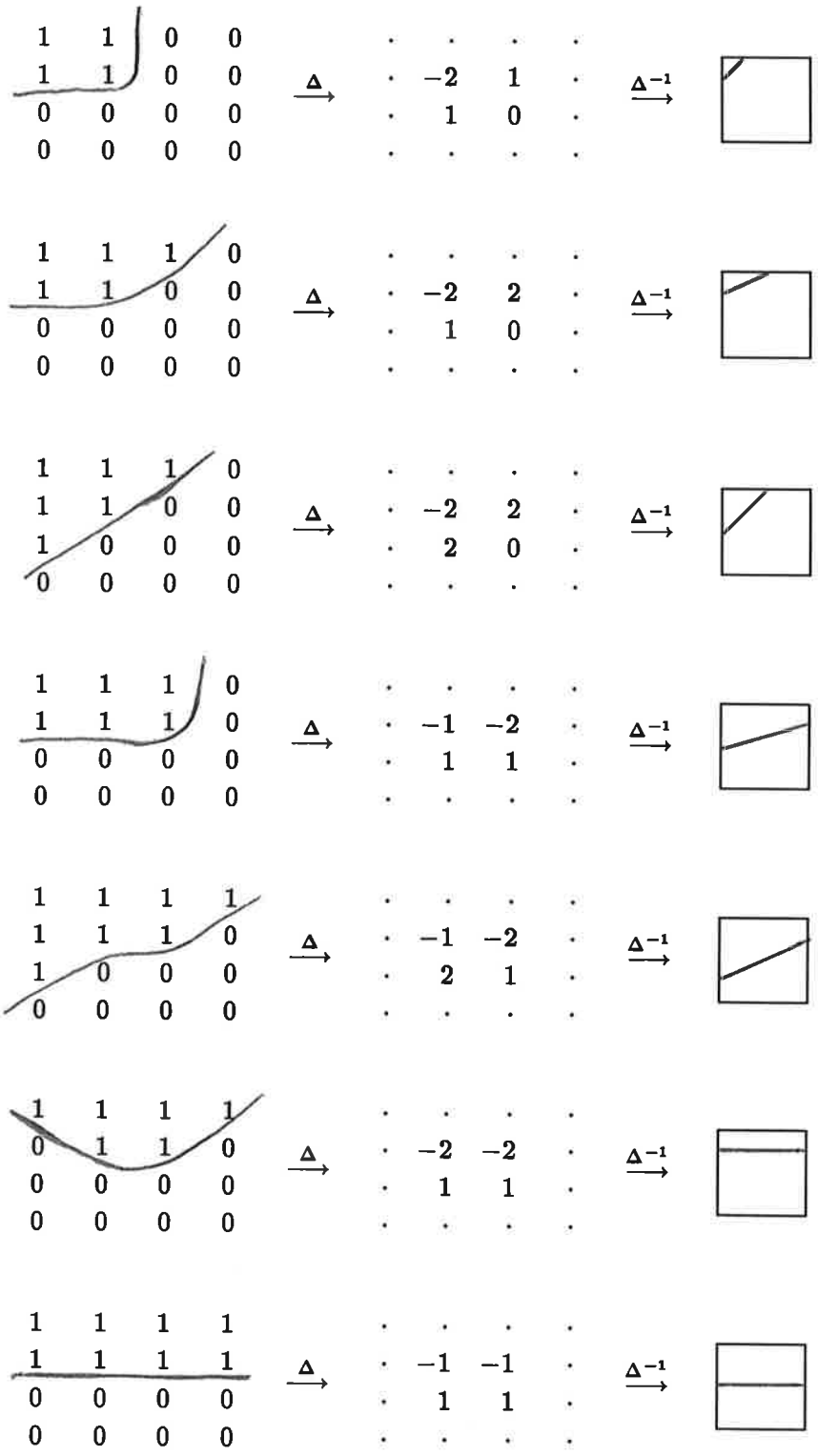

Figure 3.3 The elementary cases

together with the background. In the future, the algorithm may be extended to handle these kinds of edges. In that case the edge-catalog has to be extended to cover also intersections and thin lines.

An edge that is spread out over several pixels can be considered as a series of parallel edges separated by one pixel. Such edges have properties similar to the thin lines in the Laplace-plane. This is due to the fact that dipole-curves

separated by only a single pixel interferes with each other in the Laplace-plane. As mentioned in Chapter 2, blurred edges correspond to the second term in equation 2.1. Consequently, these edges will be not be detected.

3.4 Determination of Elementary Cases

For each square in the Laplace-image the corresponding elementary case must be determined. For each square there are three variables:

- The height of the edge.
- A background value in the Laplace-image, coming from the second term in equation (2.1). This should not be confused with the background in the original image.
- The index of the different elementary cases is a third, discrete variable.

The method of least squares is used to find an elementary case that best describes each square. It also gives the corresponding height and "background". The following algorithm requires quite a lot of calculations, so it is desirable to minimize the number of times it is executed. It is of no use to apply the algorithm to squares whose corners in the original image have about the same intensities. Therefore it is advisable to test whether the maximum difference between the four corner-values in a square is large enough, before applying the least squares method.

The Method of Least Squares

Arrange the four corners of a square in a vector:

$$\begin{matrix} a_{i,j} & a_{i+1,j} \\ a_{i,j+1} & a_{i+1,j+1} \end{matrix} \qquad s = \begin{pmatrix} a_{i,j} & a_{i+1,j} & a_{i+1,j+1} & a_{i,j+1} \end{pmatrix}^T$$

Let the elementary case k be stored in a vector:

$$e_k = \begin{pmatrix} e_{k_{0,0}} & e_{k_{1,0}} & e_{k_{1,1}} & e_{k_{0,1}} \end{pmatrix}^T$$

Create another vector which corresponds to the background in the Laplace-image:

$$f = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}^T$$

Group the latter two vectors in a matrix:

$$A_k = \begin{pmatrix} e_k & f \end{pmatrix}$$

The two variables to estimate are put into a second matrix:

$$\theta_k = \begin{pmatrix} h_k & b_k \end{pmatrix}$$

where h_k is the estimated height of the edge and b_k is the estimated background in the Laplace-image. The geometric interpretation of the least squares method is given in Figure 3.4 for the case of two dimensions (cf Sparr, 1984). The vectors h_k and b_k define a plane and the least square estimate of the vector s is its orthogonal projection onto that plane. The values on h_k and b_k are

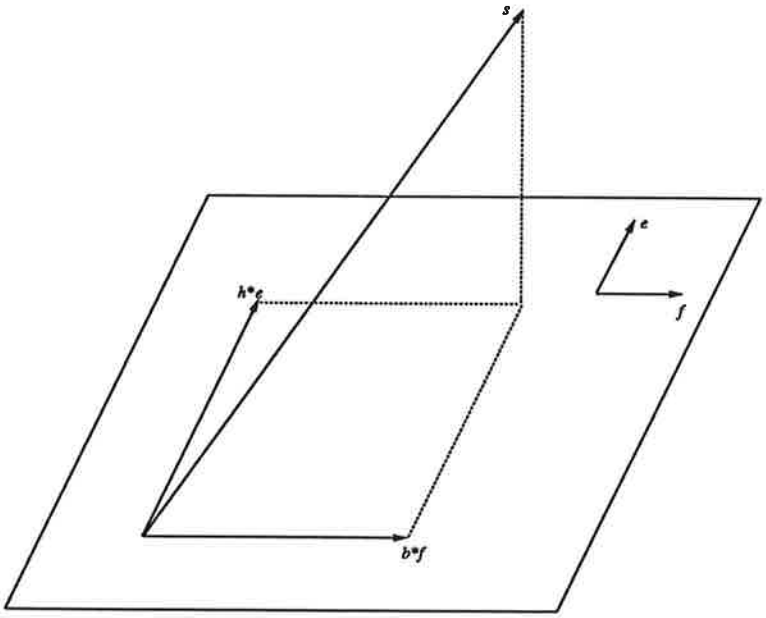

Figure 3.4 Least squares method.

then given by the projection. Different values on the index k means different planes. The plane that gives the least distance between the the vector s and its projection is the one to choose.

Let the vector with tail in s and arrow at its projection be labeled ϵ_k. The length of ϵ_k is calculated for every k. If a plane can be fitted exactly, the equation

$$s = A_k \cdot \theta_k$$

holds. Since the matrix A_k is not quadratic, the pseudoinverse is used to calculate θ_k. This is equivalent to a least squares estimate of the θ-matrix:

$$\hat{\theta}_k = (A_k^T \cdot A_k)^{-1} \cdot A_k^T \cdot s = R_k \cdot s$$

The vector ϵ_k is then given by:

$$\begin{aligned} \epsilon_k &= A_k \cdot \hat{\theta}_k - s \\ &= A_k \cdot (A_k^T \cdot A_k)^{-1} \cdot A_k^T \cdot s - s \\ &= (A_k \cdot (A_k^T \cdot A_k)^{-1} \cdot A_k^T - I) \cdot s \\ &= P_k \cdot s \end{aligned}$$

Hence $\hat{\theta}_k$ need not to be calculated to find the value on ϵ_k. Another useful property is that the matrices P_k and R_k may be calculated in advance and stored in a array. The calculation is repeated for each elementary case and its rotations. A multiplication by -1 of an elementary case can be handled by letting the variables h_k and b_k attain negative values. Consequently the index $i \in [0..33]$. The value ϵ_{min} is then calculated as:

$$\epsilon_{min} = min(|\epsilon_0|, |\epsilon_1|, ..., |\epsilon_{33}|)$$

and the index of the least ϵ_k is the value on k that minimizes the loss-function above. The estimates of the height and the background are then given by the equation

$$\hat{\theta}_k = (A_k^T \cdot A_k)^{-1} \cdot A_k^T \cdot s = R_k \cdot s$$

Here it must be decided whether the value of ϵ_{min} is small enough. A large value means that it is not possible to fit any elementary case to the actual square. If ϵ_{min} is sufficiently small then an edge is considered to be present in the square in question.

The algorithm may be designed to exactly fit mondrian images. Mondrian images contain no background, so the matrix A_k can be simplified to only contain the single e_k-vector. In geometrical terms this is equivalent to a least squares minimization with respect to a line instead of a plane.

Parameters in the Edge Detection Algorithm

The algorithm needs values for two parameters:

- the value which is used to single out those squares that contain an edge, eg. the minimum height to be detected (*mindiff*).
- The maximum tolerance for ϵ_{min} (*maxeps*).

The optimal situation is that all squares with edges are found and that elementary cases can be fit to each of these squares. The parameter values are not critical for mondrian images. For other images it is not easy to find satisfactory values on these parameters. Scanned images or images created with a video-camera are difficult to treat. Such images always contain some noise. Another complication is that the image may be blurred, which means that the edges are spread out over several adjacent pixels. The experience is that the value for *mindiff* should be rather large so that only the squares that contain significant edges are detected. Also the value for *maxeps* must be rather large, to allow quite big differences from the elementary cases. Otherwise a lot of squares with edges are found, but the corresponding elementary case for most of these squares can not be found. The values depend strongly on the particular image. Often one has to iterate several times before getting a satisfactory result.

3.5 Properties of the Edge Detection Algorithm

Ambiguity in the Edge-catalog

Introducing the possibility of background values in the Laplace-image makes it impossible to separate some of the elementary cases that covers horizontal and vertical edges, by just using the least squares method. Squares like the following one, that contains a vertical edge, fits three different cases in the edge-catalog:

$$\begin{pmatrix} 1 & -2 \\ 1 & -2 \end{pmatrix} \Longleftrightarrow \begin{cases} \begin{pmatrix} 1 & -2 \\ 1 & -2 \end{pmatrix} & height = 1.0, \quad background = 0.0 \\ \begin{pmatrix} 1 & -1 \\ 1 & -1 \end{pmatrix} & height = 1.5, \quad background = -0.5 \\ \begin{pmatrix} -2 & 1 \\ -2 & 1 \end{pmatrix} & height = -1.0, \quad background = -1.0 \end{cases}$$

The same of course also holds for the corresponding cases with horizontal edges. These squares have to be treated separately together with the ones that contain thin lines and intersections. The problem should however not

be to hard to solve - it is only necessary to examine the nearest neighbors of the square, to be able to resolve between the different cases. However, this ambiguity never appears when the background is set to zero.

It is also possible to make the following simplification, probably without disturbing the result too much. When running the examples in this report all cases that descend from:

$$\begin{matrix} 1 & -2 \\ 1 & -2 \end{matrix}$$

were removed from the edge-catalog, since they will fit the cases descending from:

$$\begin{matrix} 1 & -1 \\ 1 & -1 \end{matrix}$$

This implies that in some cases the edge-catalog does not make it possible to fit edges in adjacent squares.

Discretisation of the Angles

Figure 3.5 shows an image containing edges of different angles and the corresponding detected edges in a rather strong magnification.

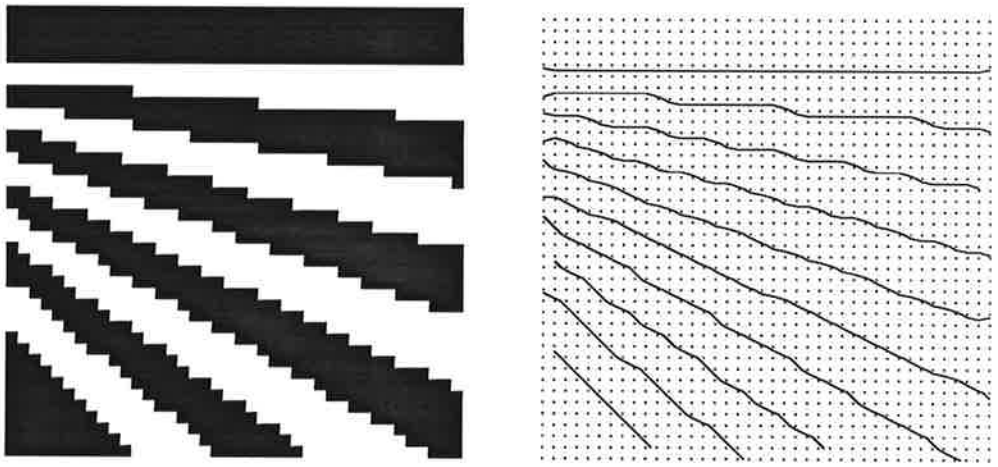

Figure 3.5 Edges of different angles and the result of the corresponding Edge detection.

Apparently, the treatment of an edge depends on its direction. Edges with directions that do not precisely fit the elementary cases get a bit ragged. Interpolating such an edge results in zig-zag-effects that get worse the more the image is enlarged. This effect is not as bad as the edges that are introduced by the pixel-replication method, but may just as well be quite unattractive.

A closer examination of the elementary cases explains the reason for this problem, namely that there are a limited number of angles that can be represented when making the inverse Laplacian. Moreover, these angles are not equally spread over the circle. Edges are represented by means of 16 different directions. These are described by the angles

$$\begin{aligned} \arctan(0.0) &= 0.0^\circ \\ \arctan(0.25) &= 14.0^\circ \\ \arctan(0.5) &= 26.6^\circ \\ \arctan(0.75) &= 36.9^\circ \\ \arctan(1.0) &= 45.0^\circ \end{aligned}$$

and their complementary and alternate angles.

This effect may be avoided if the algorithm is extended to examine larger areas of the image than only a single square at the time. The algorithm should then try to identify connected edges in the image. This implies that the algorithm should be able to recognize circles and other geometric patterns.

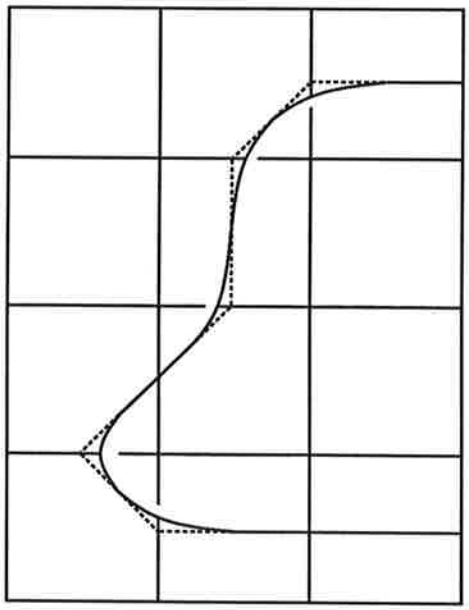

Figure 3.6 Straightening of edges in adjacent squares. The dotted line is the result of the edge detection.

Figure 3.6 illustrates a less sophisticated modification of the algorithm that may improve the result. Once the edge detection has been made, the edge image is scanned and one tries to straighten edges in adjacent squares. Such a treatment may, however, be dangerous if the image contains a lot of fine details.

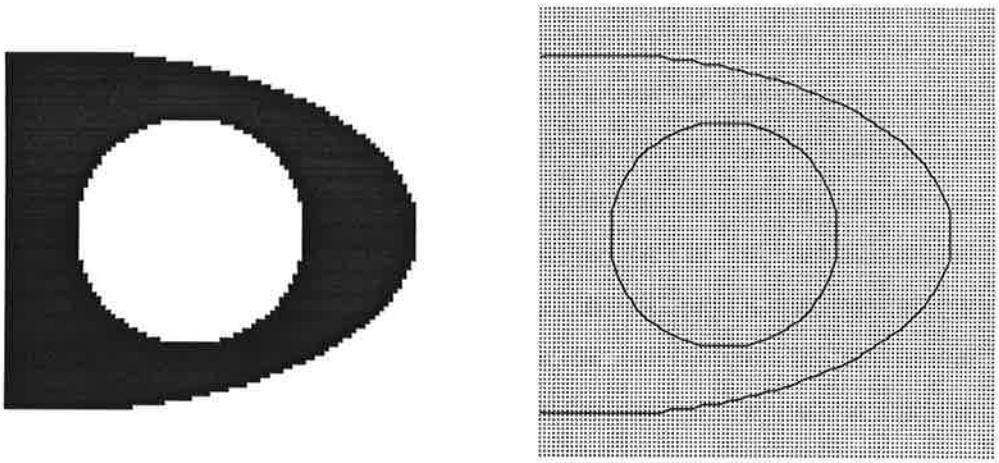

Figure 3.7 Edge Detection for a some oval patterns

Another interesting geometric form is a circle. Figure 3.7 shows some oval patterns and the resulting edges. The result seems rather pleasing, partly because this image is not magnified as much as the image in Figure 3.5. This implies that the algorithm will give satisfactory results as long as the image does not contain very fine details.

3.6 Effect of the Second Term in Equation 2.1

Edge detection of an image which originates from for example a video-camera or a satellite, introduces other problems. The experience is that these kinds of images always are a bit blurred. Figure 3.8 shows a test-image that was made artificially to simulate a blurred image. This image was used as the input to the Laplacian. The result is quite unpleasingly. This is due to the fact that edges are spread out over several pixels. This implies that the edge detection algorithm in some places has to deal with two or more parallel edges, i.e. a case for which the algorithm is not intended, in its present shape. Therefore, *maxeps* , has been chosen large to permit large errors. Otherwise one gets large gaps in the edges surrounding the pattern. On the other hand, the effect of the large value on *maxeps* is that edges that fit rather badly are accepted. This can be seen at several places in Figure 3.9, where edges do not cling together, or make some ladder-like patterns. Unsharp edges correspond to the second term in equation 2.1. It is obvious that when treating images of this kind, the influence of the second term is not negligible.

Why the result of the edge detection gets that bad can be explained in the following way. The part that is marked by the rectangle in Figure 3.9 contains

0.00	0.33	0.67	1.00
0.33	0.67	0.67	1.00
0.33	0.67	1.00	1.00
0.67	0.67	1.00	1.00

in the original image. Applying the Laplacian on the same part gives the

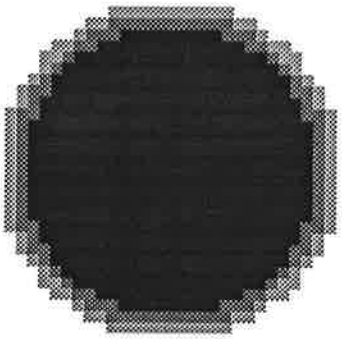

Figure 3.8 A simulation of a blurred image.

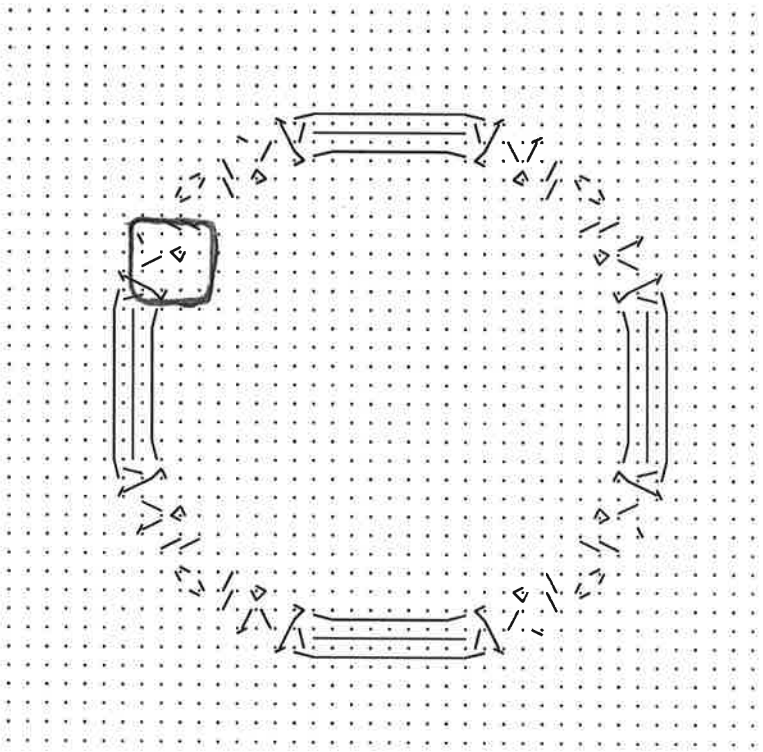

Figure 3.9 The result of the Edge Detection of a blurred image.

following result:

$$\begin{array}{rrrr} 0.66 & 0.02 & -0.35 & -0.66 \\ -0.32 & -0.68 & 0.66 & -0.33 \\ 0.35 & -0.01 & -0.66 & 0.00 \\ -0.68 & 0.66 & -0.33 & 0.00 \end{array}$$

A comparison of the values in the Laplace-image and the result of the edge detection gives that squares, where edges with erroneous directions have been fitted, contain patterns that can not be found in the edge-catalog. However, since the *maxeps* has been set to a large value, even erroneous edges are accepted. There is no simple solution to the problem. From another point of view there is no problem, since the algorithm is intended to find sharp edges, and not blurred ones. As a matter of fact, if an edge is blurred in the original image it should also get blurred in the interpolated image. Consequently, one

solution is to set *maxeps* to a reasonable small value, to avoid that the blurred edges are detected at all. A better approach may of course be to alter the algorithm in some way so that even blurred edges can be handled.

An interesting property of the edge detection algorithm is shown in Figure 3.10. It illustrates the result of the edge detection when the value on *maxeps* has been set to a quite small value.

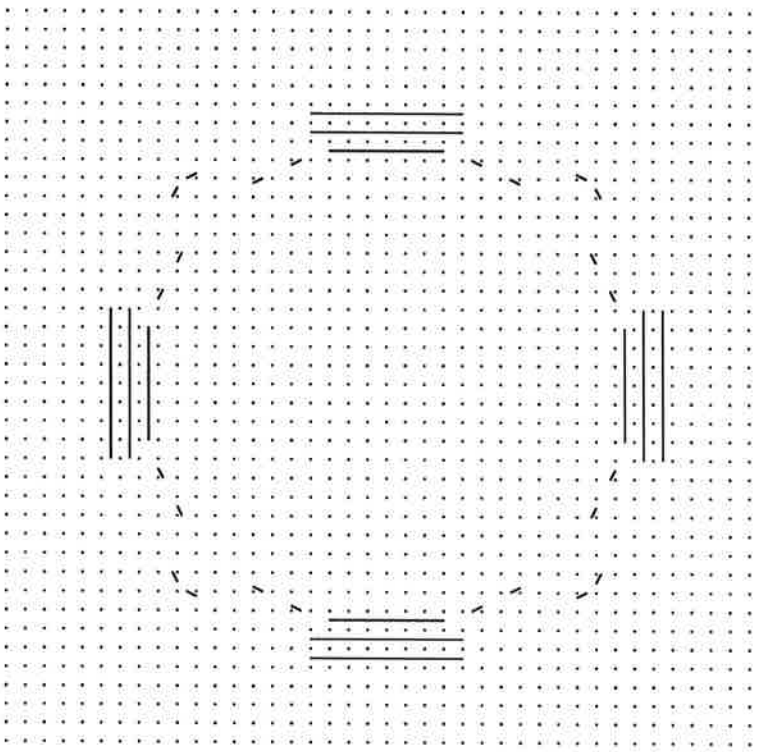

Figure 3.10 Result of the edge detection of the image in Figure 3.8 when *mineps* has been set to a small value.

Only edges that are horizontal or vertical and some few others are detected. This is due to the fact that such edges are transformed into patterns that can be exactly fitted to the edge-catalog.

4. Interpolation Schemes

Some different interpolation schemes have been considered. It is quite obvious that the simplest ones, like pixel-replication and bilinear interpolation (Pratt, 1978), cannot give a satisfactory result for images that contain anything but smooth variations. This is at least the case if one wants magnification by factors greater than two or maybe three. A bilinear method extended with some edge handling has been suggested by Hansson (1989) and is quite attractive since it is fast and seems to give a good result for at least mondrian images.

In this thesis a scheme that divides the original image into two images is proposed. One image contains the edges, and the other contains only smooth variations. Such methods need some more computing time, but might give better result than the former one for non-mondrian images. The qualities of this scheme have however not been thoroughly explored.

4.1 Bilevel Edge

The basic idea for this method is to use a simple bilinear interpolation method, but to avoid interpolation over edges. First detect the edges in the original image using the algorithm in Chapter 3. In the square in Figure 4.1, where an edge is present, two surfaces above the square are created, one on the upper level and one on the lower level.

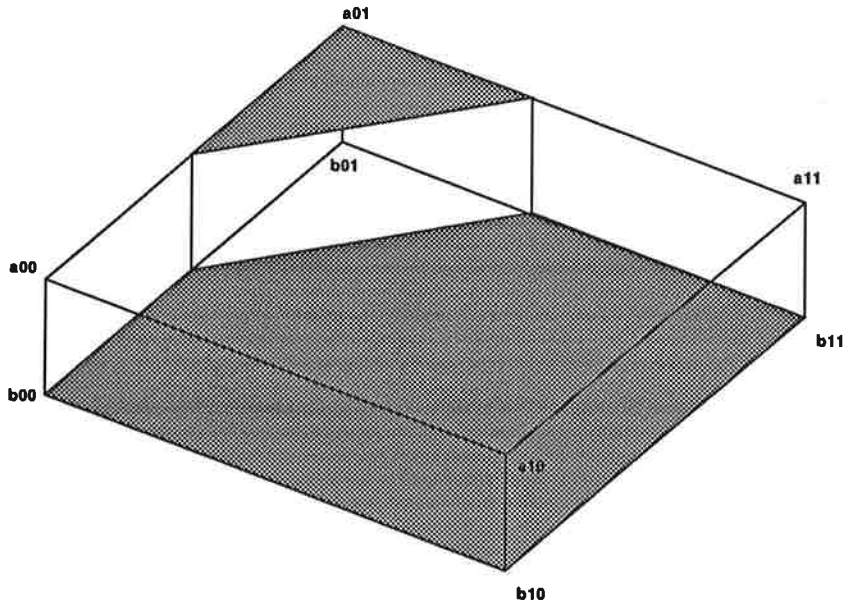

Figure 4.1 The bilevel edge interpolation scheme.

When calculating the value of the pixels on the upper side of the edge, make a bilinear interpolation in the square (a00, a01, a10, a11). For pixels on the lower side, use the square (b00, b01, b10, b11). The values of the corners of the two squares are calculated by adding or subtracting the height of the edge on the lower and upper side respectively. The figure illustrates the case

of a mondrian image, but the method will obviously also work for images with nonvanishing first derivatives.

4.2 Edge Interpolation

Using the theory of electromagnetic fields, a method for interpolation of the edges may be developed. From a mathematical point of view, the edges are interpolated in the Laplace-plane.

Assume that the image can be considered as mondrian. This means that the image shall have its first derivative equal to zero except in a limited number of places where edges are present. It can then be shown that each pixel in the image can be calculated as

$$f_{finegrid}(x) = \sum_i \frac{\beta(\gamma_i)}{2\pi}\alpha(x, \gamma_i)$$

(Hansson, 1989) where β is the height, counted with the signs of the edges. α is the angle under which the endpoints of the edge, γ_i is seen from the pixel x. However this formula implies that when calculating the intensity of every single pixel in the fine grid, all squares must be examined. It is also necessary to have a special treatment of the edges that reach the border of the image. Finally, some uniform background value must be added to the pixels to finish the interpolation. Speaking in mathematical terms, what one does is to solve the Poisson equation analytically.

One can not afford to scan the whole image for each single pixel, so some simplification must be done. A reasonable simplification is to introduce a circular window around each pixel. The reason for making the window circular is that the properties of a circular window are independent of the direction of the edges.

The Figure 4.2 shows the situation for an image that is magnified by a factor four in each direction. The radius of the window in the figure is chosen as twelve pixels in the fine grid. The straight line represents an edge of unit height which crosses the whole image. Since the window is limited, the contribution from an edge decreases with the distance between the edge and the pixel. When reaching the border of the window a step is introduced in the fine grid. This interpolation results in an image that only contains edges. There are several possible continuations of this scheme. One way includes the solving of the Poisson equation numerically for the rest of the image and is extremely slow. This thesis proposes another, faster way.

Examining the result of an edge interpolation, the similarity to a highpass-filtered image is easily seen. The image only contains edges. Having also a lowpass-filtered version of the same image, one could add these two filtered images to create the interpolated image.

Suppose the result of the edge interpolation is sampled. Then the sampled image can be viewed as the highpass-filtered, original image on the coarse grid. Consequently, if this image is subtracted from the original image the original image has been divided into one highpass-filtered and one lowpass-filtered version. Since the lowpass-filtered image does not contain any sharp edges, it may be interpolated by the simple bilinear method. Finally, the result of the bilinear interpolation is added to the result of the edge interpolation. The scheme is illustrated in Figure 4.3.

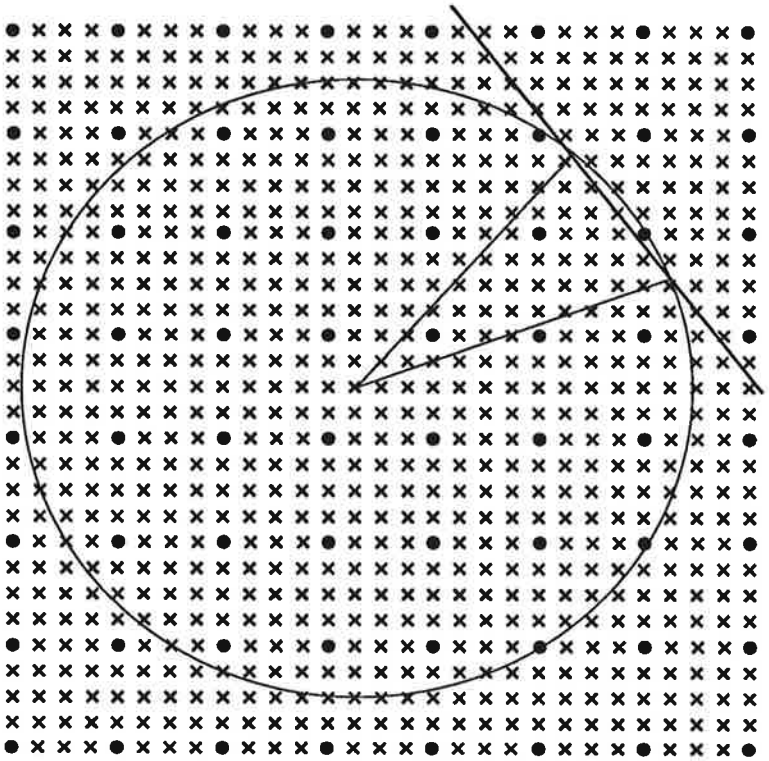

Figure 4.2 Interpolation of edges using a window.

Unfortunately this method has a disadvantage due to the edges that was introduced by the limited window. Since the lowpass-image is bilinearly interpolated these jumps will cause some shadows in the final image which may be disturbing. This indicates the need of some more complicated window-function. (As edges get closer to the circle their weight shall be reduced.)

4.3 Experimental Results from Interpolations

This section contains some results from interpolations made by different methods. The result should be compared with the result of the pixel-replication and bilinear interpolation respectively.

Interpolation of an Oval Pattern

Figure 4.4 shows the result from an interpolation of the image in Figure 3.6, using the modified bilinear scheme of the image in Figure 3.6. The magnification factors are chosen to be four in each direction. The result is quite satisfactory at least compared to the pixel-replication image. One can observe that the interpolated image does not become quite oval where it ought to be. It is easily understood why, when looking at the pixel-replicated image. This is an indication on the fact that one can not expect a result from the interpolation that is better than the original image.

Highpass Lowpass Interpolation of Images

The origin to the image in Figure 4.5 is a bilevel-image with unit height. In the image, where no weighting was made, one can see a slight shadowing mainly on the brighter side of the edge. Apart from this shadow the result

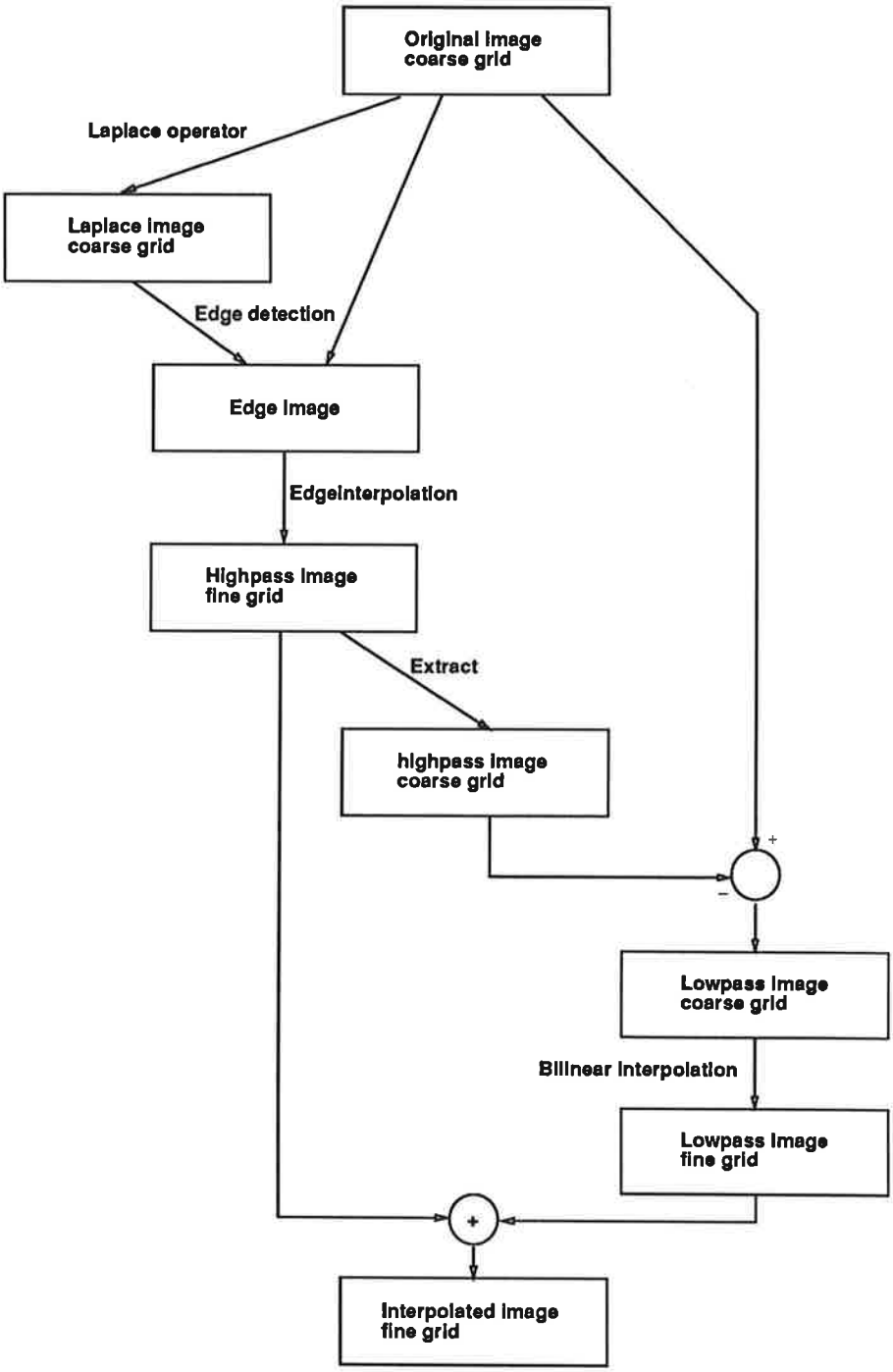

Figure 4.3 An interpolation scheme.

is satisfactory and the edges are sharp. Using a correction of the window-function the shadowing is reduced. Due to the bad reproduction technique the difference can hardly be observed, in this report.

Interpolation of a Satellite Image

The image in Figure 4.6 shows a part of northern Skåne. To be exact it is an image of Söderåsen filtered trough a green filter. When examining the images below, one shall have in mind that the reproduction technique of this report tend to blur the images.

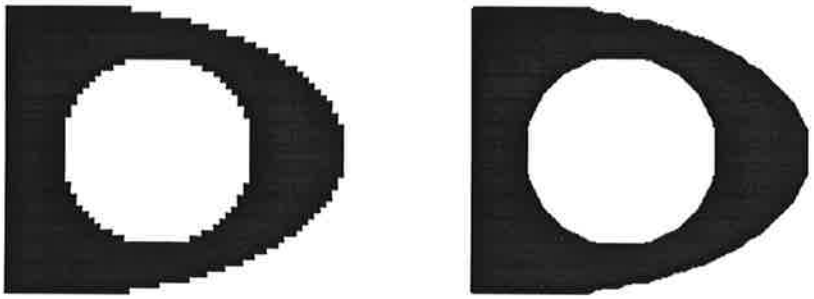

Figure 4.4 Left: Pixel-Replication method. Right: Modified bilinear method.

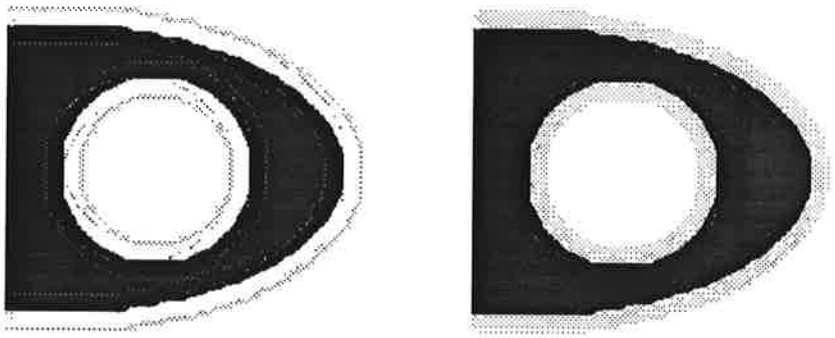

Figure 4.5 Interpolated image: Left: Without correction. Right: With correction.

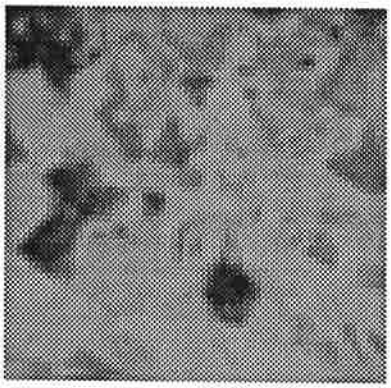

Figure 4.6 A satellite image of a part of northern Skåne.

The image is run trough a lowpass-filter to reduce the influence of noise and then sampled by excluding every second pixel. Sampling the image reduces the problem with blurred edges. However all problems are not eliminated. The sampled image is used as input to the interpolation schemes, where it is magnified by a factor eight in each dimension. The image in Figure 4.7 has been produced by pixel-replication and is shown mostly for comparison. Figure 4.8 contains an image that was interpolated using the simple bilinear method. It can be seen that all edges get quite blurred. Figure 4.9 shows the result of an image that was interpolated using the modified bilinear scheme discussed in Section 4.1. In some parts of the image it can be seen that the edges are a bit sharper than in the image in Figure 4.8. In other parts of the

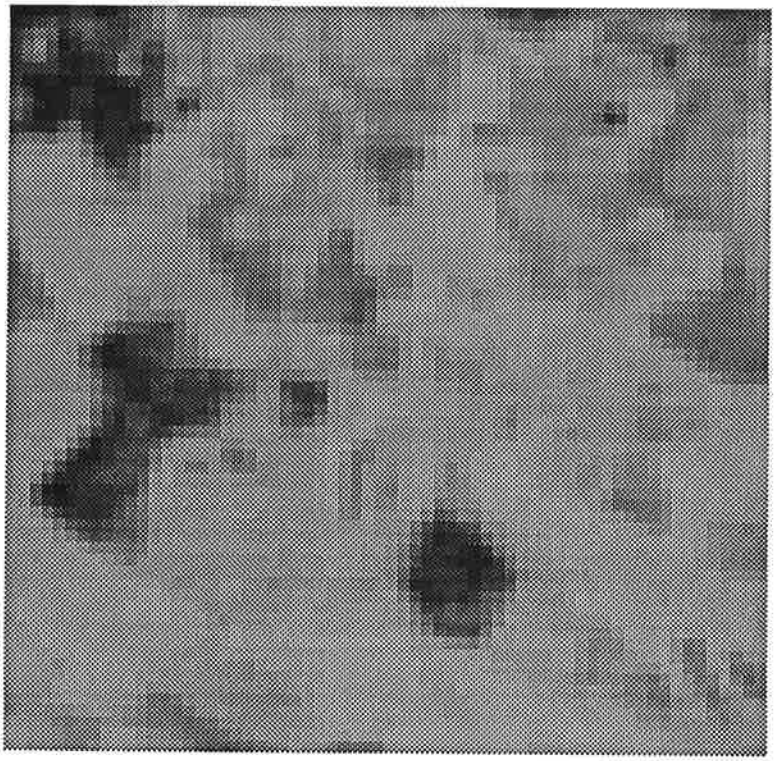

Figure 4.7 A pixel-replicated satellite image.

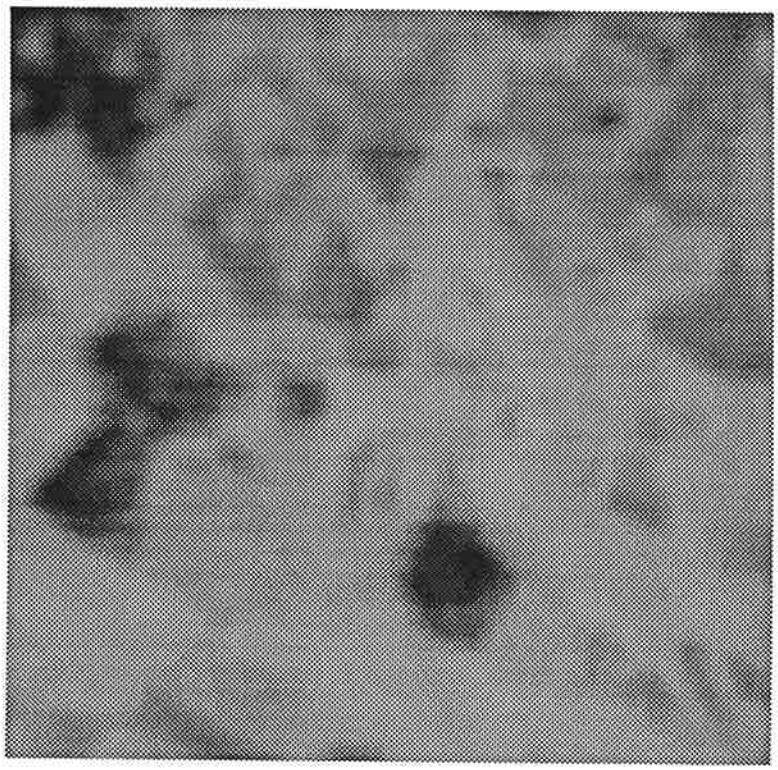

Figure 4.8 A result of the simple bilinear interpolation method.

image it is obvious that the edge detection has failed due to the blurred edges. A very close look may reveal that the structure of the edges in Figure 4.9 differ from the edges in Figure 4.8. No image is shown which has been produced using the scheme mentioned in Section 4.2. This is because the result of this

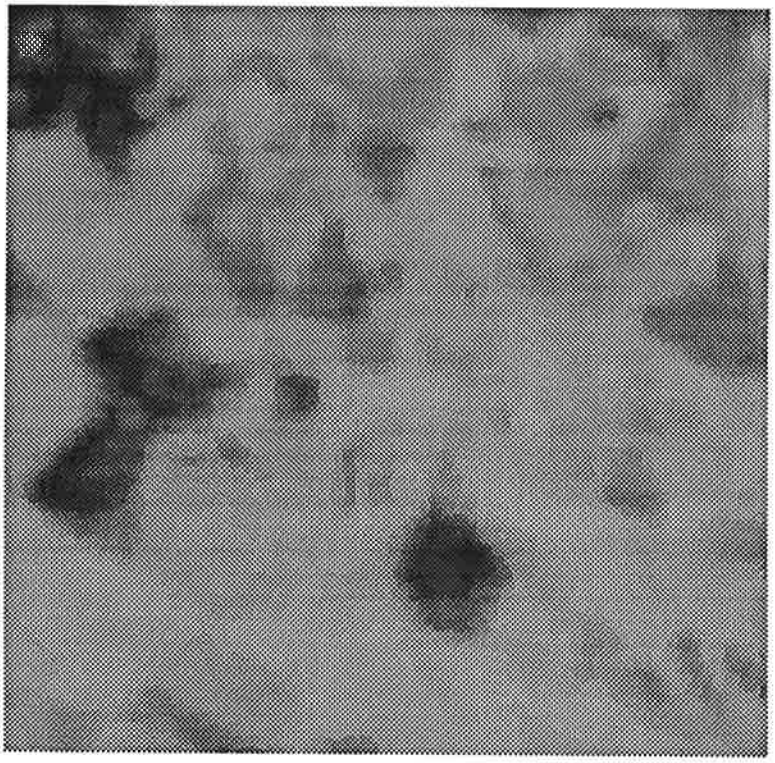

Figure 4.9 A result of the modified bilinear method.

scheme only slightly differs from the result of the modified bilinear method, at least as long as the edge detection does not give a better result.

5. Conclusions

The results in this master thesis indicate that one important condition for getting a good result of an interpolation is that the edges are preserved. Consequently, the problem can be divided into two subproblems. The first one is how to find the edges in the original image, and the second is how to interpolate.

This thesis has been mainly devoted to the edge detection problem. It has been shown that the algorithm works well for mondrian images. As long as a mondrian image contains neither intersecting edges, nor very fine details, the result gets rather satisfactory. Images that comes from sources like satellites or video-cameras are harder to handle, since they always are a bit blurred. The results for such images are not very satisfactory. This is not surprising, since the corresponding part of the image model has been omitted in the present implementation. To get really useful for ordinary images, the algorithm needs first of all to be extended to handle blurred edges. Then, in order to improve the results even further, it may be necessary to extend the method to handle intersections and thin lines.

6. References

HANSSON, A (1989): "Nonlinear interpolation from video images to high quality printers," Master thesis CODEN: LUTFD2/(TFRT-5397)/1-28/(1989), Department of Automatic Control, Lund Institute of Technology.

KELLOGG, O. D. (1954): *Foundations of Potential Theory*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, USA.

KERNIGHAN, B. W. (1978): *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, USA.

LIPPMAN, S. B. (1989): *C++ Primer*, Addison-Wesley, Reading, Massachusets, USA.

PRATT, W. K. (1978): *Digital Image Processing*, John Wiley & Sons, Inc., New York / Chichester / Brisbane / Toronto.

SPARR G. (1984): *Kontinuerliga System*, Department of Mathematics, Lund Institute of Technology, Lund, Sweden.

SPARR, G., A. HANSSON and L. NIELSEN (1989): "Nonlinear interpolation for image expansion based on potential theory," *6th Scandinavian Conference on Image Analysis*, Oulu, Finland.

SPARR, G., A. HANSSON and L. NIELSEN (1990): "Discontinuity preserving visual reconstruction by means of potential theory," *To appear in Pattern Recognition Letters.*

A. An Image Processing Program

All the examples in this report have been produced by an image processing program. The program was been written especially to support the operations needed for interpolations. It replaces an older one written by Hansson (1989). Many ideas from that program have been transferred to this one, and the programs support mainly the same features. There are, however, some important differences. The new program was intended to run under the X-windows-system, which makes it possible to execute it from remote terminals. Also it needs less memory than the former. An important advantage is that the program can be executed on a SPARC-station, which increases the speed by almost a factor of ten.

A.1 The Images

All images are stored as files. This means that nothing is lost, but the results of the running operation, if a program is interrupted. Internally the program only deals with files stored as a matrix of floating numbers between 0.0 and 1.0. These files have the extension '.float'. When making an edge detection the edge-image is stored in a special type of file with the extension '.edge'.

However, there are interfaces to other systems, that use other representations of images. The interface uses a number of different extensions for files as it handles images of various types:

- Images may be imported or exported as a matrix of bytes. This kind of images is used by the videointerface ('.byte').
- Printing images on an ink jet plotter demands a transformation to a special format, that fits the plotter ('.ink').
- Images may be transformed to a series of numbers in a textfile which can be printed or examined in an editor ('.ascii').
- The program also contains a PostScript interface which makes it possible to present images and edgefiles on a laserprinter ('.ps'). Files that are supposed to be included in a tex-document are labeled ('.PS').
- It is also possible to create .pgm-files that fit to a general image processing system. Actually these files are identical to the .byte-files apart from that they also contain a header ('.pgm').
- Finally it is possible to import images in a Fortran format used for satellite images ('.tm').

The program requires the user to be aware of the different types of files. For example, images may have the same name as long as they are of different type. However, the user need not know the type of images that are needed for a certain operation. The program itself opens and creates only files of the correct types.

A.2 User Interface

Not so much work has been devoted to the user interface. It is very simple using menus in an xterm window. Sometimes it may ask too often whether to continue or not. However, it gives you the opportunity to cancel operations, which you otherwise would have to wait on for minutes, or to interrupt.

The structure of the user interface is shown in Figure A.1. When the program is started two questions must be answered before the main menu is shown on the screen. The first question tells the program if the X-windows system is available. This means that the program may be executed from any kind of terminal, but only those that run X-windows make it possible to display images on the screen. The second question offers a possibility to change the directory where the images are to be stored and fetched.

File Handler

This submenu makes it possible to handle files from inside the program. The menu requires knowledge of the type of the file. Apart from the elementary options, like copying, listing and deleting files, there are options for compressing and uncompressing files. Images require quite a lot of diskspace, so it is advisable to store images, that are not in current use, in a compressed format. This compression of images is the one supported in the UNIX-operating system.

Export and Import of Images

This menu contains all the interfaces to other systems. The options offers conversion between different formats.

When importing or exporting bytefiles the default size is 512 times 512 pixels. This can be changed. There are no checking of the correctness of the values. Erroneous values may cause an error or an unpredictable result.

This option may also be used when importing files of type '.pgm' The fileheader must be removed from the file in an editor. It is of course necessary to remember the size of the image since it must be specified if it differs from 512 times 512 pixels.

Converting images to PostScript-format is quite simple. The answer to the question about tex-documents has no greater importance. Both kinds of files can be directed to a laserprinter using the 'lpr'-command. However converting edge-images to PostScript contains a couple of options. Firstly there is a question of scaling the image. This means the scaling of each square in the image. An excessive scale does not permit the whole edge-image to be shown. Then it is possible to decide whether the exact position of the pixels in the coarse grid is needed. It takes the printer quite a while to compute all the pixels, so if they are not absolutely needed, answer 'no' to the question about dots in the background.

When formatting an image for the ink jet plotter the plotter maximizes the width to 2400 pixels and the height to 3600 pixels. Larger images are truncated. The plotter actually permits 2800 times 4000, but the program gives some margin. The file created can not be directly sent to an ink jet plotter. It has to be run through a VAX-program that creates 5600 bytes large records. Then it shall be written to a tape with the block size set to 5600 bytes.

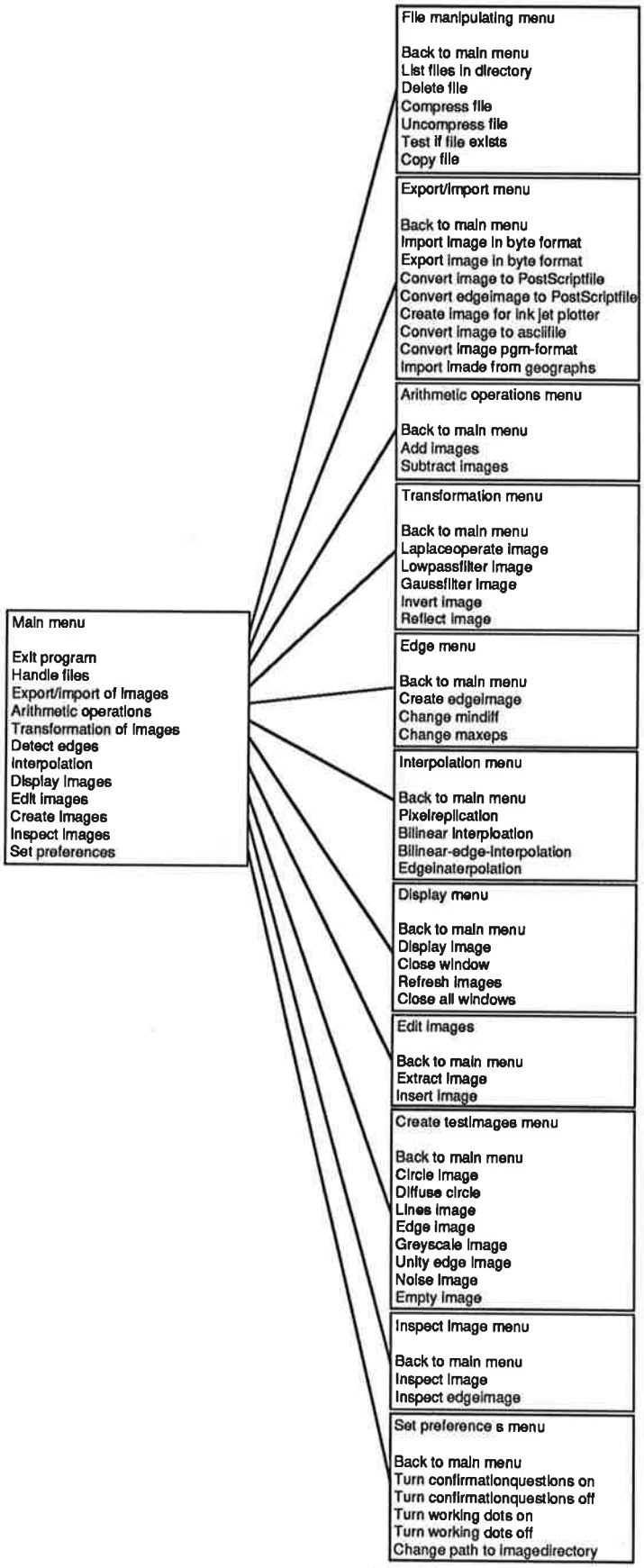

Figure A.1 The user interface

Converting images to ascii-file only permits images that are less than 20 pixels wide and 25 pixels high, since larger images will not fit on one paper. Too large images are truncated.

The '.tm'-files are supposed to be in a Fortran format where each pixel is represented by three digits. To import this kind of images one has to remove the numbers in the file that gives the size of the image. Then all spaces must be replaced by underscore. This can be done in an editor like emacs.

Arithmetic Operations

This submenu contains addition and subtraction of images. Note that the same file-name must not be specified more than once. Consequently, the name of the two operands must not be the same. Neither is the name of the result allowed to be the same as the name of any of the operands. Of course the two operands must have the same dimensions.

Transformation of Images

The operations in this menu transform images, by using various operators, to images of the same size. The Laplacian is the one chosen in Section 3.1.

Two different lowpass-filters have been implemented. Both filters are two-dimensional and noncausal. The first one is a simple rectangular filter. It requires you to specify the size of the filter. The size must be odd and greater or equal to three. The second filter is a 3 times 3 or 5 times 5 filter with the following weighting functions:

$$\frac{1}{4.9}\begin{pmatrix} 0.368 & 0.607 & 0.368 \\ 0.607 & 1.000 & 0.607 \\ 0.368 & 0.607 & 0.368 \end{pmatrix}$$

$$\frac{1}{8.016}\begin{pmatrix} 0.000 & 0.000 & 0.368 & 0.000 & 0.000 \\ 0.000 & 0.607 & 0.779 & 0.607 & 0.000 \\ 0.368 & 0.779 & 1.000 & 0.779 & 0.368 \\ 0.000 & 0.607 & 0.779 & 0.607 & 0.000 \\ 0.000 & 0.000 & 0.368 & 0.000 & 0.000 \end{pmatrix}$$

These filterfunction were created since there was a suspicion that the rectangular filter affected the edges in a bad way.

Inverting an image is identical to subtract the image from an image filled with 1.0's. Reflecting an image yields the image reflected in a vertical line located at the center of the image.

Edge Detection

When detecting edges the variables *mindiff* and *maxeps* should be chosen in accordance with the discussion in Section 3.3. The values are critical for the result of the edge detection, unless the image is a simple bilevel image or mondrian image. Reasonable values for simple video images are 0.05 on *mindiff* and 0.05 to 0.5 on *maxeps* . When an edge detection is finished one gets some statistics on the result. From the statistics one may find out if the edge detection has given a satisfactory result, although the best way is to convert the edge-image to PostScript and print it. It is possible to get a list of the locations where the program has found an edge, but has not been able to fit any of the elementary cases. Usually it is necessary to iterate a couple of times before satisfactory values on *mindiff* and *maxeps* can be found.

Interpolation

Four different kinds of interpolation are supported. The simple 'pixelreplication' scheme has been implemented mostly as a comparison. The 'bilinear' method creates a fine grid by a two-dimensional linear interpolation within each squares in the coarse grid. The 'bilinearedge' is the modified bilinear method described in Section 4.1. Finally 'edge-interpolation' is the method described in Section 4.2. All the methods permit the magnification in each dimension to be chosen as any value larger than or equal to one. The values need not be integers. The 'edge-interpolation' also requires a value on the radius of the window-function.

Display Images

This menu makes it possible to display images on the screen. A dithermatrix is mapped over the image to create a bitmap. However, a bilevel image is not a satisfactory way to evaluate the interpolation-methods. Sometimes the image may never show up, or disappear from the screen for some reason. In that case the option 'Refresh images' can be used.

Edit Images

This is not editing in the usual manner. The 'extract image' makes it possible to cut out a smaller part of an image and sample it. It is also possible to make a sampling of the entire image. 'Insert image' means pasting a smaller image on a specified position in a larger image. An image that does not fit will get truncated.

Create Testimages

Sometimes it is useful to have some testimages to use as input to different procedures. This menu produces some that have been found useful. The 'noise image' is produced by running a random number generator (0 to 255) to each pixel. If the value is less than the specified level the pixel is set to black (1.0), otherwise it is set to white (0.0). Also the last one called 'empty image' requires a comment. When making, for example, ink-jet-plots it is desirable to place images beside each other in the same image. An empty image is useful as the location of such an image.

Inspect Images

It is useful to be able to examine the value of single pixels in images. The option 'inspect images' requires the name of the image and a pair of coordinates, and then returns four values in the corresponding square. One can also inspect edge images, and in that case the program returns the contents of the specified square in the edgefile.

Set Preferences

The last menu makes it possible to switch on and off the dots that are printed while the procedures are running. One can also reduce the number of questions that asks if one wants to continue. Finally, one can change the directory where images are fetched and stored.

B. The Implementation

This chapter discusses the implementation of the program in more detail. Understanding the program requires some knowledge of programming and the language C++. The chapter is mainly devoted to parts of the program that may be of general or certain interest.

B.1 Structure of the Program

The program consists of more than 6000 lines of source code and is divided into 37 different files. Each file usually contains an option in one of the menus. Consequently, the structure of the program corresponds well to the structure of the menus shown in Figure A.1. However, some utility routines are found in their own files.

One may think that the number of files is too large, but my experience is that it makes it easier to make changes in the program. Compiling is no problem, since the 'make'-utility in UNIX has been used.

B.2 Description of the Files

This section describes the files in alphabetical order.

arithmetics.C

The file contains the routines for addition and subtraction of images.

bilinear.C

The routine for the simple bilinear interpolation.

bilinearedge.C

The routine for the modified bilinear interpolation. When calculating the value of a pixel, its location in the coarse grid must be found, since it is necessary to know which square to use for the calculation. If there is an edge present in the square, two surfaces above the square are created, one on the upper level and one the lower. For pixels on the upper level, the upper square is used for the interpolation and vice versa. Consequently, interpolation over the edges is avoided according to the discussion in Section 4.1.

byteinterface.C

This file contains the procedures for conversion between the byte format of images and the internal format of images. In the byteimages 255 means white and in the program 1.0 means black so a subtraction has to be made for each pixel. The conversion from float to byte may seem more complicated than the conversion from byte to float. The reason is that images that do not fit to the specified size, when converting from float to byte, are centered.

createimage.C

The file contains routines for creating testimages. The direction of the edges in the 'lines-image' are chosen so that they are equally spread over 1/8 of a circle. The basic idea for the 'edgeimage' is that pixels that lie closer to the center of the image than any of the points in the array, are set to black. Also observe that no extra code needs to be written to create an empty image, since the class 'DestImages' automatically initiates an new image to zero. All images are of course of type '.float'.

edgeclass.C

This is a utility file that contains the classes 'Edge', 'SourceEdgeImage' and 'DestEdgeImage', which are used when reading and writing edge-images. The class Edge is prepared to permit several edges in one square. Only the type of the edge is stored, not its coordinates, since the type only is needed to find the coordinates of the edge in the global edge-library.

A simple buffer has been implemented in the classes 'SourceEdgeImage' and 'DestEdgeImage' to reduce the need of memory. This buffer makes it possible to keep only a small number of rows in the memory at the time. However, the files are sequential so rows older than the ones in the buffer cannot be accessed, unless the member 'ReRead' of the class 'SourceEdgeImage' is called. The class 'DestEdgeImage' does not contain the member 'ReRead', so these images must always be accessed in order, except for the lines kept in the buffer. When creating image-objects the size of the buffer may be specified. 'DestEdgeImage'-objects also need a specification of the size of the image. The buffer automatically initiates the edges so that squares that have not been accessed by 'PutEdge' will contain no edge.

The file format for the edge images is text. Hence, the files can be examined in editors like emacs. Two numbers in the beginning of the file specify the size of the image. Each row then represents one square. A zero in the left column indicates that no edge is present in the corresponding square.

edgedetect.C

This file contains the routine that performs the edge detection. The algorithm is the one presented in Chapter 3. Squares that do not fit any elementary case are stored in the 'SpecialList', so that they can be treated later on. The background is never calculated since it is not used at the time being.

edgeintp.C

This is an implementation of the edge interpolation with window. For each pixel the squares that need to be examined are found and then the program enter the inner loops. For each square, that contains an edge, there are four possibilities:

- Both endpoints of the edge lie on the outside of the circle, and the edge does not intersect with the circle.
- Both endpoints lie on the outside of the circle, but the edge intersects with the circle.
- One endpoint lies on the inside and the other on the outside of the circle.
- The edge lies entirely inside the circle.

In the last case the endpoints shall not be changed. In the rest of the cases the endpoints that lie on the outside are moved along the edge until the edge crosses the circle. This is performed by solving a second order equation. For the first case the equation has no solution and this identifies the case.

If a square still contains an edge-segment, the angle under which the edge-segment is seen is calculated by the function 'Angle' which implements the cosines theorem. A crossproduct is calculated to determine the sign of the angle. Finally, a correction factor is found. The correction is based on the distance between the middle of the edge-segment and the actual pixel. Edges that lie closer to the pixel should get a larger weight than edges that lie far away from the pixel. The correction used in the program is

$$CorrFact = 0.5 * (1 - \sin(d/R * \pi))$$

where d denotes the distance between the center of the edge segment and the pixel and R denotes the radius of the window. It gives rather good results although one could ask for better.

One may be led to believe that a better result could be achieved if the orthogonal distance between the edge-segment and the pixel is used, instead of the distance between the center of the edge and the pixel. This gives an optimal result for straight edges. However, this is not the case for edges that are not straight as shown in figure B.1.

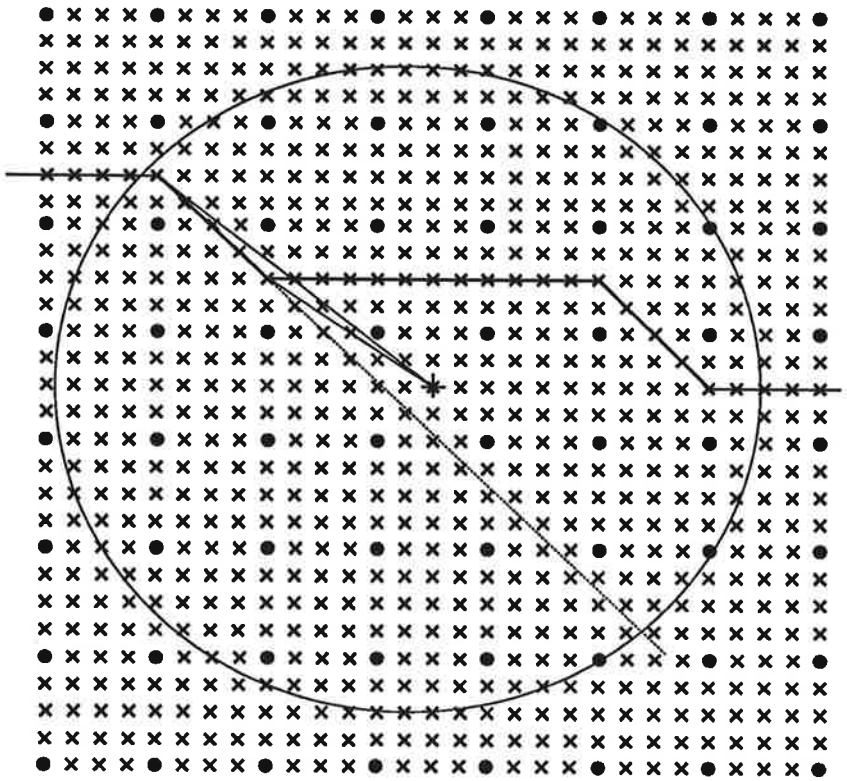

Figure B.1 The consequences of a filter-algorithm, based on the orthogonal distance between the pixel and the edge-segment. The angle marked in the figure gets the wrong sign. Besides the sign it gets a large weight, since the distance between the marked point and the dotted line is small (the orthogonal distance).

Is is obvious that the edge segment marked in the figure adds a value with the wrong sign. The value also gets large compared to the rest of the edge, since the orthogonal distance is short.

edgelib.C

All the matrices that are used in the handling of edges are stored in this utility file. All matrices have been created using MatLab.

edgetops.C

Conversion of edge images to PostScript-code which can be directly output to a laserwriter. The scale asked for in the program is used to determine the size of each square on the paper.

editmenu.C

This file should really not need any comments. It only implements the edit images menu.

exportimport.C

This file should not need any comments. It is the menu for export and import of images.

extract.C

This image implements the extract function. It is quite straightforward.

filehandler.C

This is one of the utility files. It contains procedures that are used when manipulating files in the program.

fileserver.C

The procedures in this file build a user interface to the procedures in the file 'filehandler.C'.

floattoascii.C

Conversion of an image to an asciifile. Such a file can be sent to a printer or examined in emacs.

floattoink.C

Conversion of images to a format that fits the ink jet plotter. Each pixel on the plotter corresponds to 16 bits, four for each color. Since only black is used all bits are set to zero but the lowest four. The intensity produced by the plotter

is not proportional to the value but follows the following table (Hansson 1989):

bitcode	*intensity*
0	0.000
1	0.068
2	0.134
3	0.206
4	0.288
5	0.378
6	0.468
7	0.558
8	0.642
9	0.722
10	0.784
11	0.848
12	0.892
13	0.914
14	0.956
15	1.000

An error diffusion technique is used to translate the image to sixteen levels.

floattops.C

This procedure converts an image to PostScript-code. PostScript contains an operation 'image' that makes a rastration of an image. The image should be input to the operation as a series of bytes in hexformat.

gaussfilter.C

This is the implementation of a lowpass filter with the weighting factors mentioned in Section A.2.

imageclass.C

This utility file contains the classes 'SourceImage' and 'DestImage'. From the file 'imageheader.H' the class 'ImageHeader' is inherited. The class 'ImageHeader' is supposed to be able to administrate a header for each image. This header may contain data and some history about the image. The class 'ImageHeader' has not been implemented.

Here also some simple buffers have been implemented. They have the same properties that the buffers in 'edgeclass.C'. Note that pixels, that are not explicitly set to a value, automatically are set to zero.

The imagefiles are stored in byteformat with four bytes per pixel. In the beginning of each file there are 32-bit words that represent the size of the image.

imageserver.C

This utility file contains the most used routines in the program. It takes care of the input of names of images from the user. It is supposed to give the user an opportunity to cancel the operations.

insertimage.C

Code for inserting small images into larger ones.

inspectimage.C

Code for inspecting images at pixel level. Also edge images can be inspected. The code gets a bit complicated, since the procedures have to use the 'ReRead' operation to allow one to examine pixels in any order.

interpolation.C

This implements only a menu.

laplaceoperation.C

This is a fast implementation of the Laplacian. Code has been written to handle every special cases, that arises in the corners and at the borders of the image.

lowpassfilter.C

This a straightforward implementation of the two-dimensional rectangular low-passfilter.

main.C

The code for the main menu is found in this file. Pleasingly short, just the way a main program should be.

opcom.C

This utility file contains the visible part of the user interface. If a more sophisticated user interface is required, this file is to be rewritten. The routines for input are quite complicated, since the are supposed to take care of erroneous input.

pixelrepl.C

Code for pixel-replication of images.

preference.C

Code for changing preferences.

transform.C

Apart from the transform-menu this file also contains code for inverting and reflecting images. There is no option to rotate images, since such a routine would require a lot of memory.

xinterface.C

This utility file contains the code used for exposing images on the screen. To create a bilevel image the rastermatrix

$$\frac{1}{17}\begin{pmatrix} 16 & 8 & 14 & 6 \\ 4 & 12 & 2 & 10 \\ 13 & 5 & 15 & 7 \\ 1 & 9 & 3 & 11 \end{pmatrix}$$

is mapped over the image. Every window on the screen is implemented as a classobject. The procedure 'ClearMemory' is used to clear the underlying bitmap which corresponds to each image. After a call of the procedure 'ClearMemory', calling 'RefreshImage' has no effect and may cause an program error.

B.3 The Program's Environment

The program also includes files from the system. To use the X-windows system the files 'X.h', 'Xlib.h' and 'Xutil.h' are required in the C++ versions. Besides those, the program uses a string-packet in the file 'strings.C' and some definitions in 'defs.H', both written by Dag Brück. Of course the program also uses some of the standard libraries for maths, filehandling and standard input/output.