Nonlinear Smoothers for Digital Image Processing

by

Eric Cloete

**Submitted in partial fulfilment of the requirements
for the degree**

D.Tech

in the
School of Business Informatics
Cape Technikon
Cape Town

December 1997

This thesis is my own original work.

Acknowledgements

I am grateful to the following people and institutions for assistance, guidance and constructive criticism. Without their help this research would not have been possible.

Dr. C.H. Rohwer - *external project leader*

Prof. D.J. van Schalkwyk - *internal project leader*

Cape Technikon

Freie Universität Berlin

FRD Foundation

My **parents** for their continual support and faith in my career.

Abstract

Modern applications in computer graphics and telecommunications command high performance filtering and smoothing to be implemented. The recent development of a new class of *max-min* selectors for digital image processing is investigated with special emphasis on the practical implications for hardware and software design.

Contents

CHAPTER ONE

Introduction

Computer visualisation can loosely be defined as an interface between computers and humans with the aim of simplifying understanding of complex problems. Many important scientific problems can only be solved satisfactorily if some form of cognitive human recognition takes place to identify problems which are embedded in larger systems. Visualisation should therefore be seen as a method to transform the symbolic into the geometric domain. This is the key to productivity and puts emphasis on computer applications for most computationally complex problems. It is therefore not surprising that this field is one of the most rapidly expanding disciplines of computer science with a promising future for hardware and software development.

Image processing deals with the display and modification of existing pictures, such as digitised photographs, X-rays or television scans [38,79][1]. It should not be confused with the related field, *computer graphics.* Although there is a fair amount of overlapping between these two topics, computer graphics is the term normally used to *create* a computer picture [83]. Image processing and computer graphics should thus be seen as sub-sections of computer visualisation. People like to interpret the *final product* of scientific processes. If weather prediction is selected as an example, few persons would think about the intricate numerical processes that had to be devised to arrive at the weather chart. Once this final product is available, it serves as an easy tool for interpretation and communication and it is seldom necessary for the end-user to revert to the underlying data of the image. Note that weather prediction is a fine example where both methods, image processing (*satellite images)* and computer graphics (*graphics*

[1] Footnotes are presented in superscript form, while references are always in square brackets.

effects and symbols added to the template), are employed to visualise the computer simulation.

Image enhancement plays an important role in acquiring and modifying a picture [54,77]. Enhancement processes are well known analogue processes, such as tuning a television set or modifying a photograph for visual improvement. Typical examples are those of contrast or brightness manipulation which are illustrated in chapter two.

Most computer images are created by point sampling of the image space and then reconstructed digitally from the sampled data. During the process of digitising an image, unwanted signals, often random in nature, may appear due to physical phenomena or defects in electronic circuits used in the process. Many procedures exist to reduce or even eliminate the visible noise spots in such a picture, but they are often costly due to extra processing and tend to reduce the original picture quality.

The aim of this research is to illustrate new digital methods for noise reduction and to compare them with some state-of-the-art procedures. The relative success that was obtained recently with a new class of one-dimensional *smoothers*[1], the so-called **LULU**[2] selectors, prompted an in-depth research effort to discover and analyse similar tools for two-dimensional application. The objective was to write application programs for personal computers to assist with visualisation methods for these new techniques. Eventually a number of filters and smoothers were selected as candidates to illustrate noise reduction and image enhancement in a programming environment. Although the programming is mainly aimed at binary and grey-scale examples for modelling reasons, colour graphics and speed-up techniques are also discussed where relevant. The programs were validated on synthesised as well as real images to elucidate topics in image processing, mathematical smoothing and programming. The results are primarily

---

1 A term used for non-linear 'filtering' of image noise. Also see chapter three and four for detailed definitions. In the later chapters a distinction between the terms *smoother* and *filter* will be made.

2 LULU is an acronym for an 'upper-lower' limit class of algorithms. Refer to §3.4 for a first definition.

intended to give technical computer science students with limited engineering and mathematical background insight into graphics programming for scientific and commercial purposes. The tools developed should also prove useful for research in the mathematical verification of filters and smoothers.

It was unavoidable that many interesting topics from image processing had to be excluded from this publication due to the vastness of the subject. A brief overview of an image processing environment is however described in chapter two, while chapter three deals with an overview of one-dimensional smoothing and introduces the **LULU** concepts. The modification of one-dimensional algorithms for the two-dimensional domain is argued in chapter four and the existence of a large number of related new smoothers of the **LULU** class is pointed out. Chapter five summarises the mathematical implications of **LULU** smoothers, while the programming concepts are discussed in chapter six. Chapter seven deals with the practical application of some filters and the newly-developed smoothers of this thesis. In the last chapter, chapter eight, some questions on the future use and viability of **LULU** smoothers in hardware and software are considered and further related research areas are pointed out.

CHAPTER TWO

Digital image processing

2.1 Computer images

During the past few decades, computer images changed from crude ASCII[1] pictures to sophisticated colour images with full photo-realism (see images 2.1.1 and 2.1.2). This reflects the rapid growth in computer hardware and software technology. Sophisticated computer processes can now be observed, often in real-time, with images to assist operators with control procedures [47]. A typical example of such an application is a space shuttle launch. It is unimaginable to think of space exploration without computer images and the related technology [27, 84].

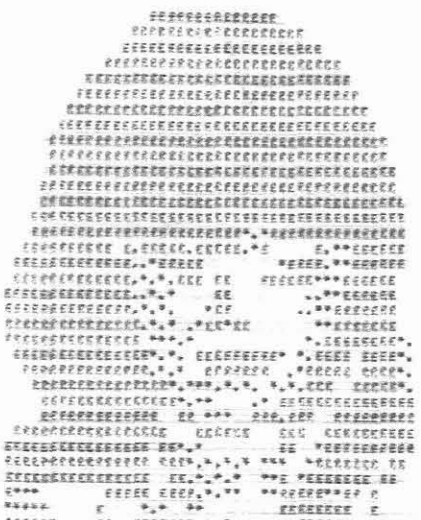


Image 2.1.1

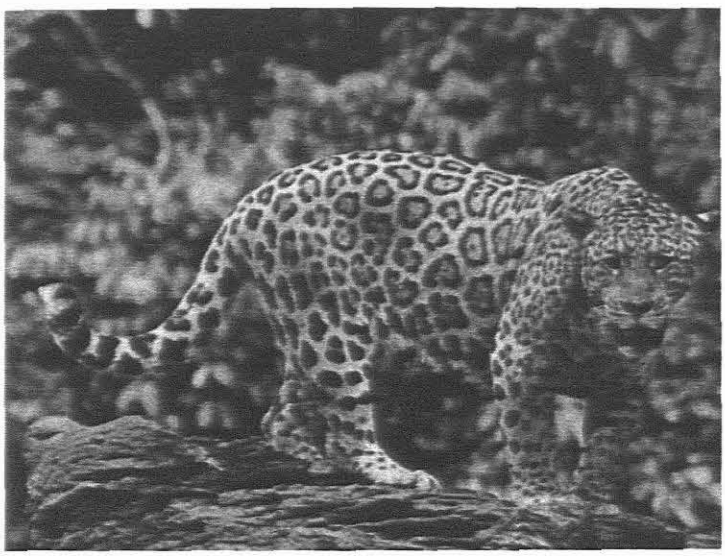


Image 2.1.2

[1] American Standard Code for Information Interchange.

Humans are largely dependant on vision for analysis of information. This is why image processing and graphical representation became so important in software design. Computer aided engineering depends very much on the quality and reliability of 3D graphical and surface representations. Many images, such as those generated from microscopes and telescopes, are only understood by human interaction after extensive computer enhancement has taken place [28]. This can include procedures of illumination, contrast or edge detection to emphasise detail for human understanding. It is important to note that human vision is comparative rather than qualitative and imagery should assist the operator in this respect.

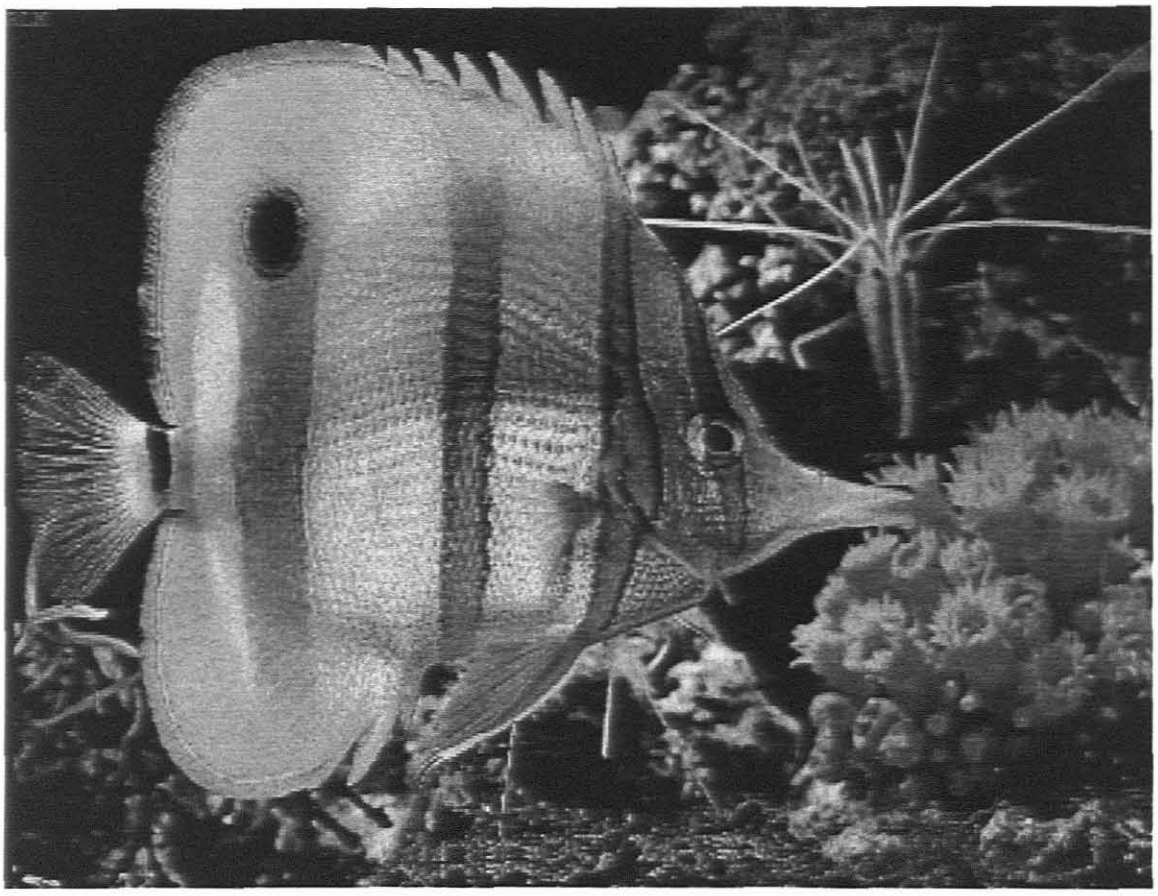
2.2 Acquiring digital images

Digital image processing (DIP) can be loosely defined as a set of computer processes that interact with a digital image to transform the original image into a new one. A digital image is basically a dataset that describes to each pixel in the graphics domain what the position, intensity and colour should be. Colour look-up tables (CLUT's) are imperative for fast image loading. A SVGA[1] CLUT for a PC with 256 colours usually stores 18 bit values: 6 bits for red, 6 bits for green and 6 bits for blue, hence the usual RGB[2] reference to DIP pictures. This results in 262144 colour possibilities. Although the prints in this thesis are all mostly 256 shades of grey, some colour representations are included in the file of images. Screendump 2.2.2 is an example of such a digital picture[3].

Digital images can be formed by either digitising an existing picture (scanning) or by taking a digital photograph. This can be done using a digital camera, where the conventional film is replaced by solid-state sensors which act as photon counters. Depending on the memory size of the camera, a number of digital photographs can be taken and later downloaded to a computer.

---

1 Super video graphics adapter.
2 Red, green and blue.
3 The database of files (appendix A) has the 256 colour version of screendump 2.2.2.



Screendump 2.2.2 256 shades of grey.

Another common procedure is to use a video camera or CCD (charge coupled device) connected to a 'video grabber' device in a computer. Video images can be frozen in real-time and downloaded as images of a specified format. This method is particularly popular due to the relative inexpensive hardware requirements as well as the quality of imagery attainable.

2.3 Neighbourhood ranking

A definite advantage of a digital picture in comparison with an ordinary photograph is the fact that the DIP can be manipulated to suit the operator. This is normally achieved by some convolution technique, whereby a central (nucleus) pixel is transformed to some

other value by taking the neighbouring pixel values into consideration. Figure 2.3.1 indicates some common kernel designs as used in practice. In linear processes all the kernel values as defined will contribute to the new central value of the nuclues of the kernel.

An easy application would be to define a process whereby the an original image[1], **O**, is transformed into a new image, **P**, according to the following set of rules:

- Use a five-point 3X3 kernel (figure 2.3.1).

- Each nuclues[2] from the new matrix is computed as the average of the four surrounding pixel values (NWES[3] values):

$$\eta_{i,j} = (O(north)+O(south)+O(west)+O(east))/4$$

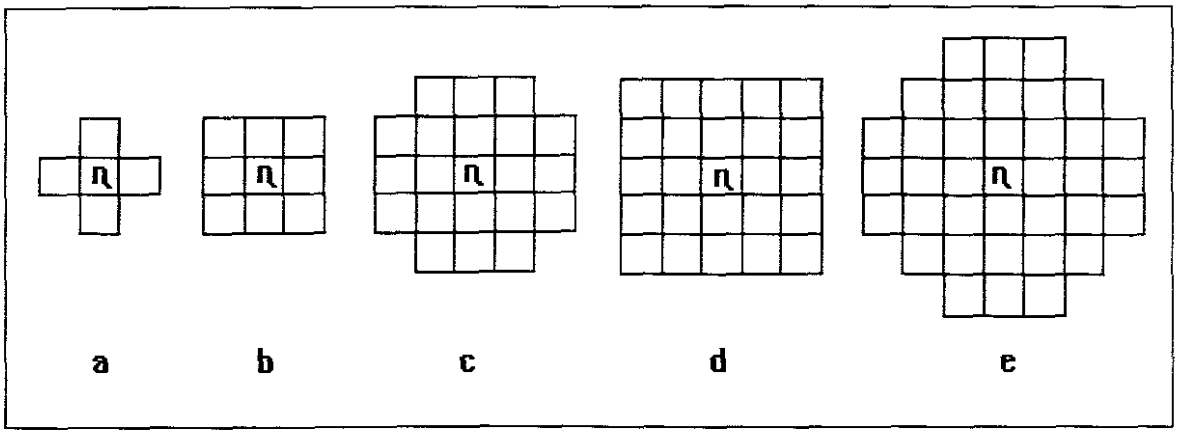


Figure 2.3.1 Some common kernel designs [13].

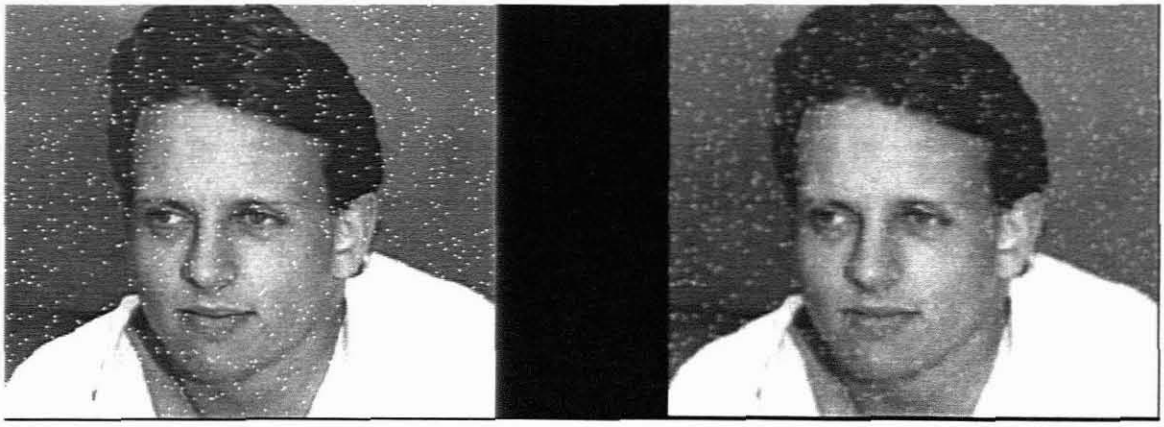
---

1 The following notation will be used throughout this thesis: **O** for *original image* and **P** for final *picture*, i.e. the picture of the matrix, P, after image processing has taken place. Note that in this thesis an image will be capitalised and bold, while the matrix representing the image will be in normal print.

2 The centre pixel of a *sweeping window* is often referred to as the nucleus, η, of the transformation kernel.

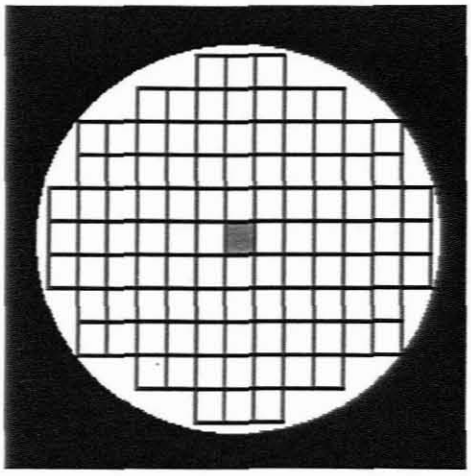
3 Direction of surrounding pixels, north, west, east, south of the image **O** with relation to the nucleus.

In this example the effect of the convolution is to give each pixel a blurred effect, due to the interaction of the neighbouring pixel values. Image 2.3.2 has some white spot noise present. When it is modified with this pixel averaging algorithm the result is clearly visible in image 2.3.3. Instead of getting rid of the noise the resultant image is *'smeared'* and the noise is still clearly visible.



Screendump 2.3.2 'Before' **Screendump 2.3.3** 'After'

- If **O** is the *original* input image of size nxn and the sweeping window is of size 3X3, **P**, the final *picture*, would be of dimension $(n-2)^2$.



Numerous other kernel designs can be applied for special effects. Local equalisation [13,22] can be achieved by using a circular or elliptic area of pixels around the nucleus. Figure 2.3.4 is an example with 74 surrounding pixels having an effect on the replacement of the nucleus [59].

Figure 2.3.4 Circular designs

Depending on what result is required from the transformation, weighing techniques could be employed. This would mean that pixels further from the nucleus do not affect the replacement pixel as much as those nearer to the centre. The effect of this method is to highlight detail in uniform areas of an image and also to show border areas clearer by *pixel spreading* [32].

2.4 Image enhancement techniques

It is not the purpose of this chapter to fully document image processing possibilities. That would be impossible due to the vast nature of the subject. It is however necessary to understand the underlying methods of convolution and transformation of an image before the image processing techniques of later chapters can be explained.

Some of the most common DIP functions are explained and illustrated in the examples to follow.

2.4.1 Image brightness

Look-up tables (LUT's) can be read and manipulated extremely quick with modern computer hardware and software and are used extensively to manipulate computer images. Instead of tuning an image with the brightness button of the computer screen, the same effect can be achieved with software manipulation. When the LUT values are read by the computer, a brightness offset can be added, (figure 2.4.1.1) and the resultant image displayed.

Screendump 2.4.1.2 illustrates the result of this easy routine.

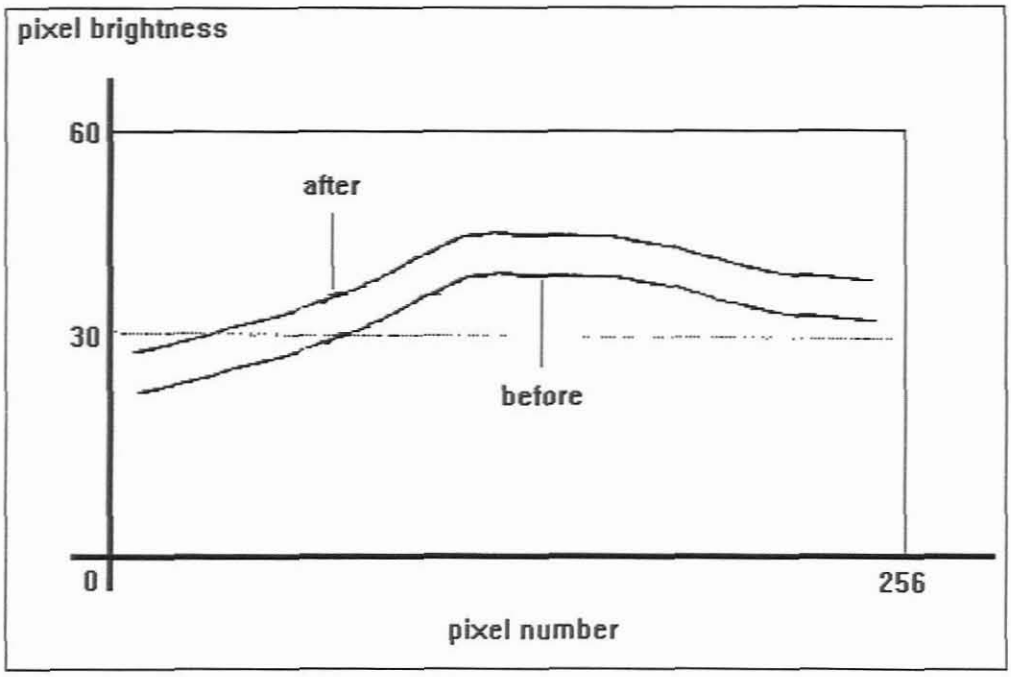


Figure 2.4.1.1 Brightness adjustment

Image 2.4.1.2 Brightness enhancement

The sweeping window of 2.4.1.3 indicates a convolution kernel that would achieve pixel emphasis.

0.0	-1.0	0.0
-1.0	5.0	-1.0
0.0	-1.0	0.0

The basic idea of this transformation is to have kernel weight values symmetrical around the nucleus such that the sums of the weights are greater than zero.

Figure 2.4.1.3

This can be achieved by setting the nucleus weight positive and the values on the rectangular axes negative and symmetrical around the kernel. It is common programming practice to build these enhancement effects into software modules so that the image can be *tuned* interactively with a pointing device.

Image characteristics can be illustrated quite efficiently with a histogram[1] which reflects the brightness levels of pixels. Screendump 2.4.1.4 shows the histogram of a digital picture.

Screendump 2.4.1.4 Image histogram

Early work on local area histogram modification and pixel amplitude re-scaling was done by Ketcham [54], while vast research was published on procedures which involves

[1] Image histograms are sometimes referred to as *luminance* histograms [75].

exponential and hyperbolic shaped histograms by Castleman (1979) and Pratt (1978). For most applications in this research, histogram equalisation is used as a tool for comparing and analysing the effect of noise removal[1].

2.4.2 Contrast enhancement

One of the common defects of electronic images is due to poor contrast being displayed when brightness is reduced. Most of the pixels then have luminance levels lower than the average and therefore the histogram of such an image will be skew toward the darker levels. A similar procedure than the one mentioned in §2.4.1 can be used to set the contrast of an image. Instead of adding a constant to the LUT to effect brightness, the values are now multiplied by a constant. Note that the midpoint of the brightness curve needs to be fixed for scaling purposes [45]. Figure 2.4.2.1 indicates this effect graphically, while figure 2.4.2.2 demonstrates what happens to the resulting histograms.

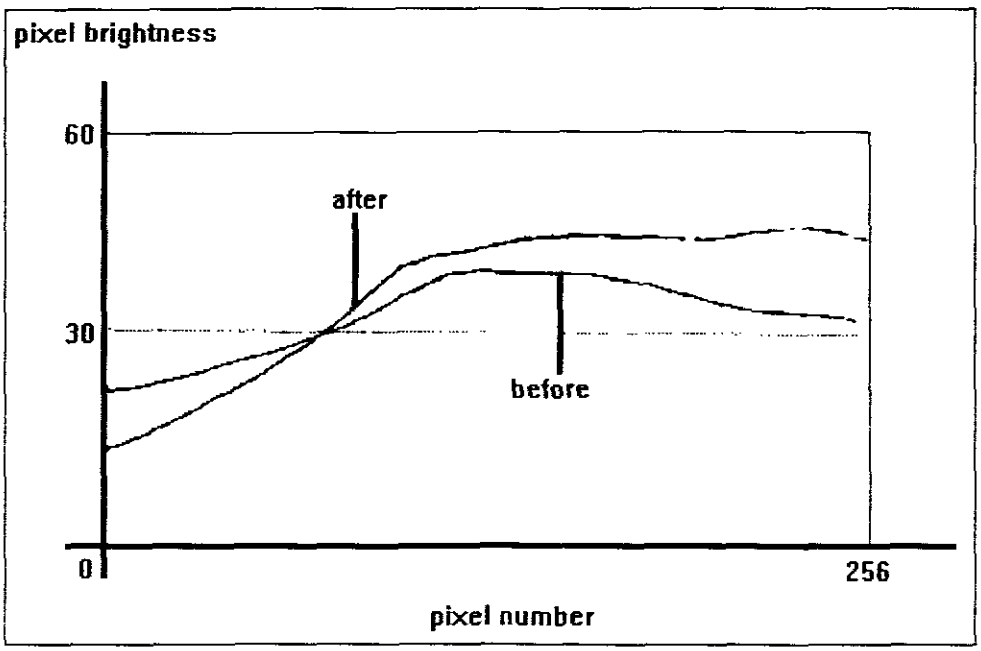
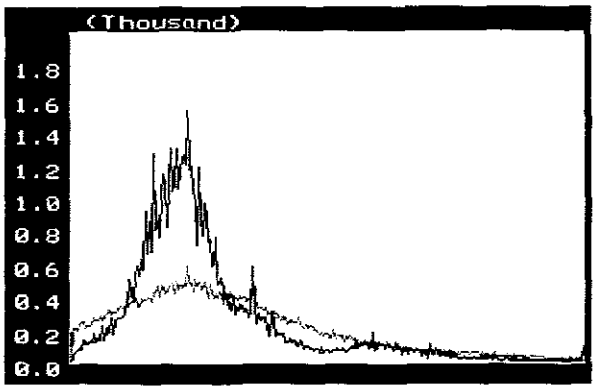


Figure 2.4.2.1 Image contrast, CLUT values

[1] Reduction in the case of images which are over-infected with noise speckles.



Brightness and contrast are the two effects of display units that are most often used, because it allows the image to be 'tuned' to suit individual needs.

Figure 2.4.2.2

2.4.3 Blurring a digital image

There are several techniques available to blur an image, other than the method shown in screendump 2.3.2. A typical example is to apply a *low-pass filter* [50] on the image. Low spatial frequency pixel values will be passed through the filter, while high spatial components will be blocked. This effect is also known as *pixel spreading*.

0.05	0.15	0.05
0.15	0.2	0.15
0.05	0.15	0.05

An example of a 3X3 convolution kernel for pixel spreading is shown in figure 2.4.3.1 Note that the sum of the kernel weights should equal 1.0 and that the kernel weights are again set symmetrical around the nuclei.

Figure 2.4.3.1

Screendump 2.4.3.2 illustrates this effect on an image. The degree of blurring can be set by adjusting the kernel values or by increasing the kernel size.

Screendump 2.4.3.2 Blurring an image

2.4.4 RGB to grey-scale conversion

There are many different methods to convert a colour image to grey-scale and vice versa, especially if dithering is allowed. The most straightforward way is to use the average value of the RGB values for each index for the new LUT.

Although most discussions in this thesis apply to grey-scale images, colour images can be handled in a similar fashion by repeating the averaging function three times, for the RGB values. A more detailed discussion on programming for colour graphics is presented in chapter six.

2.4.5 Edge detection

If the brightness kernel in figure 2.4.1.3 is tuned until the sum of the kernel weights equals zero, then a high-pass filter will come in effect and all low-pass elements of the image will be blocked out. The effect of this convolution is shown in screendump 2.4.5.

Screendump 2.4.5 Edge detection

One of the numerous applications for this image processing feature is satellite photography for GIS[1]. Contours and edges of maps can be scaled and fitted with relative ease, using electronic maps. This can be accomplished by START[2] modules which extracts digital features from linear images [75]. Although such satellite photographs are of high density, they are quite often contaminated with noise that has to be removed by some other procedure. This relates to most of the work reported on in the later chapters of this thesis.

Another example is locally adaptive binarization methods which are used in OCR[3] techniques where low contrast backgrounds and noise are present [78].

2.4.6 Reducing the size of an image

An easy way to reduce the size of an image is to discard some pixels according to some chosen formula and only display the rest. If every tenth pixel is used, a picture, one tenth the size of the original will be displayed. This method will not always give satisfactory

---

1 Geographic information systems.
2 Segment tracing and rotation transformation.
3 Optical character recognition.

results due to raggedness and aliasing side effects the final picture. These side-affects can be overcome to some degree by resampling techniques [71].

If for instance, the image is reduced five times and the resampling is planned row-wise, the average of every five pixels can be used as a new estimate for a new pixel. This way little information is discarded and a *smoothed* reduction is obtained. See screendump 2.4.6 for an example of this technique.

Further enhancement of the reduced image can be obtained if a convolution kernel is used rather than line resampling.

Screendump 2.4.6 Size reduction

2.4.7 Image enlargement

100	250	150
200	300	350
130	400	460

Figure 2.4.7.1

A similar technique to the one discussed in 2.4.6 can be used to increase the size of an image. In this case one obviously needs some replication of pixels.

100	100	100	100	**250**	250	250	250	**150**
100	100	100	100	250	250	250	250	150
100	100	100	100	250	250	250	250	150
100	100	100	100	250	250	250	250	150
200	200	200	200	**300**	300	300	300	**350**
200	200	200	200	300	300	300	300	350
200	200	200	200	300	300	300	300	350
200	200	200	200	300	300	300	300	350
130	130	130	130	**400**	400	400	400	**460**

Figure 2.4.7.1 shows a small patch of pixels that are to be enlarged by a factor four. The result of this process is shown in figure 2.4.7.2.

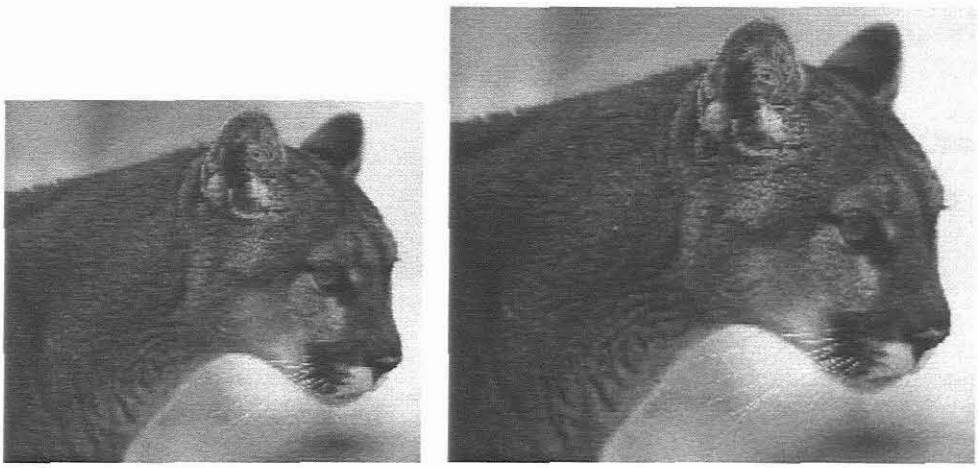
Although this method effectively increases the picture size, the block matrices are clearly visible, especially as the enlargement factor grows.

Figure 2.4.7.2

100	138	175	215	**250**	225	200	175	**150**
125	160	194	230	263	247	232	216	200
150	182	213	245	275	269	263	257	250
175	204	232	260	288	291	294	298	300
200	225	250	275	**300**	313	325	338	**350**
183	219	254	290	325	339	352	365	378
165	212	258	304	350	364	378	392	405
148	205	262	319	375	390	404	419	434
130	198	265	333	**400**	415	430	445	**460**

Figure 2.4.7.3

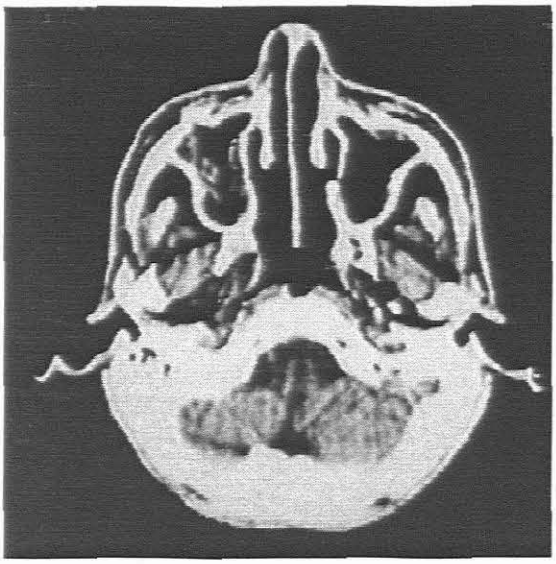
A more acceptable method is to interpolate between pixels to preserve the smoothness of the image. Figure 2.4.7.3 shows the result of such an interpolation process on the current example, while screendump 2.4.7.4 shows the 'smoothed' enlargement.



Screendump 2.4.7.4 Image enlargement

2.5 Three-dimensional images

In many image processing applications volumetric information is required to work with and the display of a single picture is not sufficient [60].



In medical applications, like the well-known CAT[1] scans, it is required to have a vast set of images available to enable the specialist to form a complete picture to diagnose. This is usually accomplished by means of a large number of parallel images, which, when combined, form a three-dimensional image. One such X-ray slice from a brain scan is shown in screendump 2.5.1.

Screendump 2.5.1

A vast field of computer research exist in the medical disciplines for refining imagery, combining data from separate sources into a single image, and evaluating image quality [30]. It is necessary to be able to stretch, squeeze, and rearrange images to compare and correlate damaged tissue with the corresponding normal cases. The role of computer imagery is of cardinal importance for such applications. Image processing may also be used to improve images by comparing constraints based on known features and by eliminating noise. The importance of reliable methods of high integrity is evident, because no significant information should disappear in the process [32, 50].

Figure 2.5.2 illustrates how *voxels*[2] are created to form compound images with multiple picture planes [18].

Another application worth mentioning, is that of voxelization in IFSAR[3] applications. Image registration is needed to extract the correct phase difference between two received

---

1 Computer assisted tomography.

2 Three-dimensional pixels are sometimes referred to as volume pixels, i.e. voxels [82].

3 Interferometric synthetic aperture radar.

signals. Processing parameters must be computed very rapidly, based on the relation between spectral shift and linear stretch of the images involved.

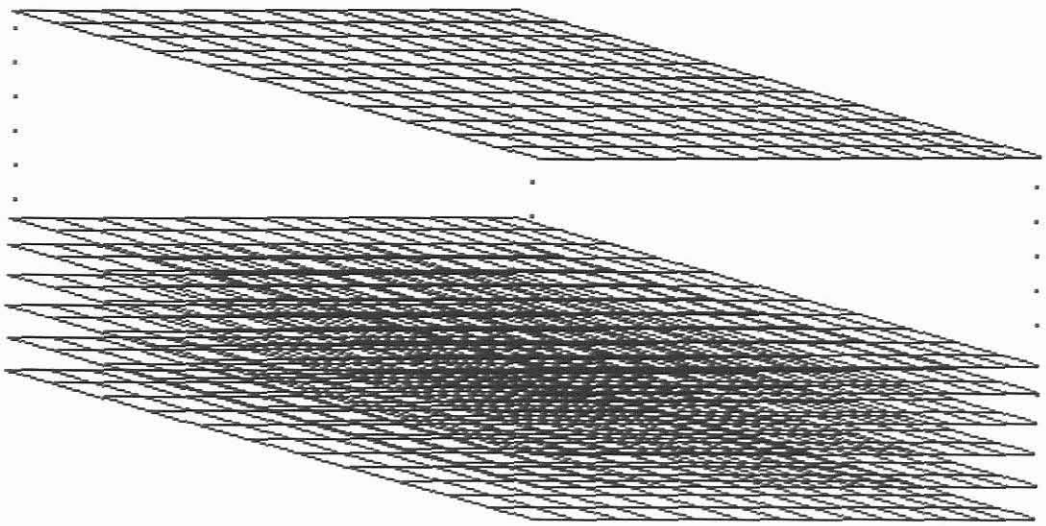


Figure 2.5.2 The construction of volume pixels (voxels)

Due to the vast amount of processing required in these applications, it is an obvious candidate for parallel processing. Each individual *papro*[1] will need basically the same processing power and will operate independently on its slice of data [16,33]. Many other applications of volumetric images can be found in the literature [18,47].

1 Papro is the abbreviation used by some authors for a single processing element in a parallel computer.

CHAPTER THREE

One-dimensional LULU Smoothers

3.1 Experimental data

It is a well-known fact of sampling experimental data that *roughness* or *noise* exists in original data due to equipment errors or other related factors. In acoustic data the difference between a clear signal and a noisy signal can be illustrated when comparing music from compact disc and alternatively an old fashioned needle record player. The CD player uses digital technology to produce a sound which is crisp and clear, while the analogue methods of record players allow noise to seep through due to dust on the record or a blunt pick-up needle.

In the laboratory, audio noise can easily be demonstrated by comparing analogue and digital methods of saving data. If an audio cassette is used to copy music, say twenty times, the deterioration of the final copy can be heard when compared with the original track. The difference in digital and analogue methods can further be illustrated if a sound sampler is available. The graphs of the original track and the final track can be overlaid and the introduced noise will be clearly visible. If the same original digitised wave is copied by computer, it does not matter how many times the copy process is repeated. The wave will remain the same and virtually no noise will be introduced.

As a more sophisticated example, with consequently much more emphasis on accuracy, a missile tracking process could be analysed. Testing a missile involves the capturing of a large amount of data from various instruments. A "Best Estimate Trajectory" (BET) is defined in space by weighing data originating from various data input channels. This process often involves real-time prediction and data filtering putting the emphasis on speed and quality of the algorithms used [53, 64].

If the observation matrix, H, is of dimension nxm, the estimation problem can be described as:

$\mathbf{x} = \mathbf{H}\boldsymbol{\emptyset} + \mathbf{v}$

$\boldsymbol{\emptyset}$ represents the parameter to be estimated at time t

$\mathbf{v}$ represents the random measurement error at time t

$\mathbf{x}$ is the measurement at time t

If typical errors like random noise (statistical) and systematic errors are removed from the data, a solution can be found by minimising the error function [52].

Obtaining solutions can involve standard methods like sampled mean, maximum likelihood or least squares for instance [1]. The speed and efficiency of adaptive filtering [4] demand that all possible *a priori* knowledge of the trajectory should be taken into account. It also can be pointed out that there exists no "unique" solution. Each solution depends on a different set of criteria. It can however happen that some of these distinct solutions may coincide [14, 48].

In our approach to correcting one or two-dimensional experimental data, we shall largely omit stochastic correcting methods and rather concentrate a new class of

deterministic methods to eliminate *pop noise*[1]. Figure 3.1.1 shows noise in the one-dimensional case, while screendump 3.1.2 illustrates a *noisy* computer image. Note that two-dimensional noise appears as discrete isolated pixel variations that are not spatially correlated. Screendump 3.1.3 illustrates this fact with an enlargement of the randomly distributed noise of screendump 3.1.2.

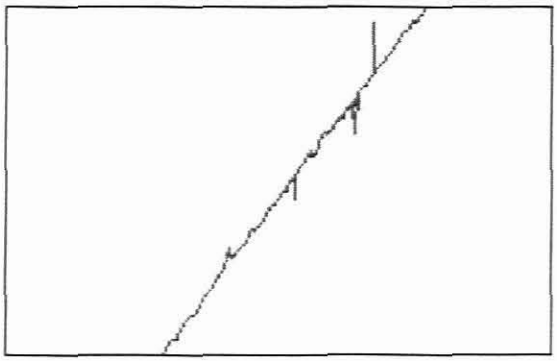
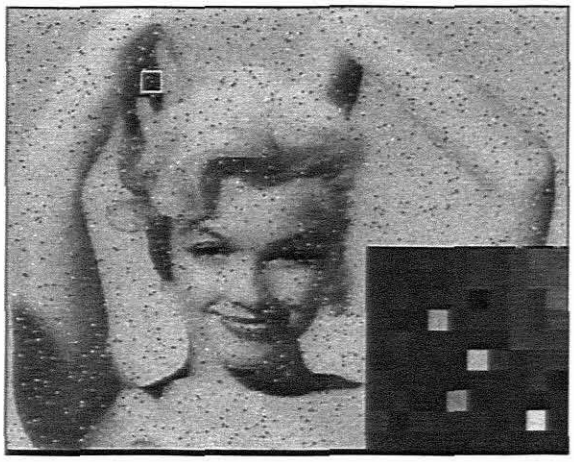


Figure 3.1.1 Noisy 1D data

Screendump 3.1.2 Noisy 2D data



Screendump 3.1.3 Zoom function.

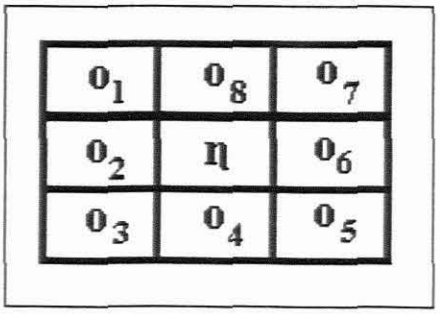


Figure 3.1.4

An automatic check for such data errors can easily be included in software. If a conventional 3X3 window (figure 3.1.4) is used to manipulate and calculate new

[1] Pop noise are pixels in the two-dimensional domain that are markedly different to their immediate neighbours. The similar term commonly used for a one-dimensional high energy peak is *outshooter*.

nucleus values for 2D data, a simple check can be built into software to identify and replace pops according to a predetermined formula. In the simple case below, the nuclei, $\eta_{i,j}$, are replaced with the average value of the surrounding pixels if a pre-defined tolerance is exceeded.

$$\text{if } \left| \eta_{i,j} - 0.125 \sum_{k=1}^{8} o_k \right| > \xi, \text{ then } \eta_{i,j} = \sum_{k=1}^{8} o_k / 8$$

3.2 1D Filters and smoothers

In most signal processing applications it is common practice to have some filter (smoother) built into algorithms to eliminate or reduce the detrimental effects of noise. These smoothers can either be linear or non-linear of nature [44,58]. Linear system theory is a well-developed field used to describe the behaviour of physical phenomena, such as electrical circuits and optical systems. It also provides a firm mathematical background for the study of sampling, spatial resolution and filtering. Linear filters are widely used in engineering and other applications because they obey superposition principles and are shift invariant [43].

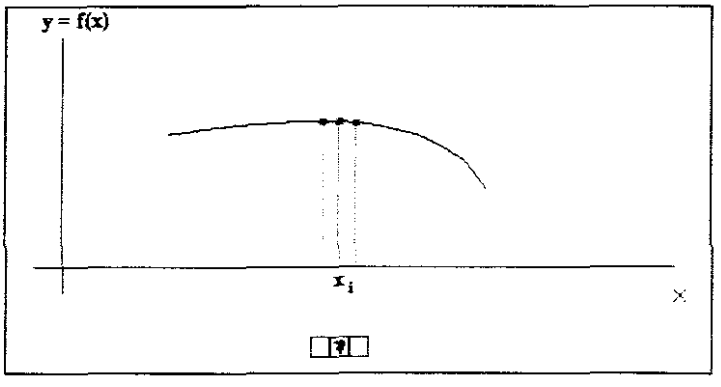


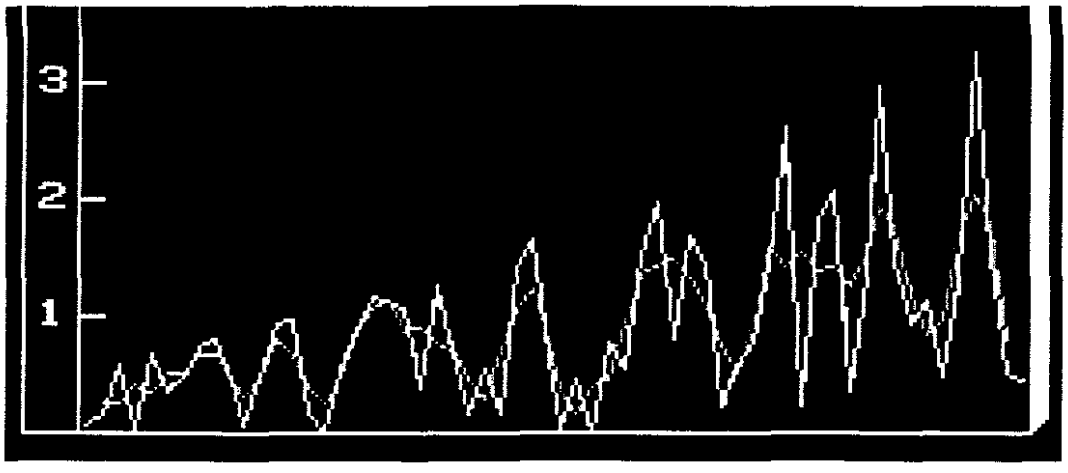
Figure 3.2.1 $f(x_i)$ is replaced by the new nucleus value, η_i

For the sake of uniformity, we shall refer to the linear case as a *filter* and to the non-linear case as a *smoother* [58].

An easy application of a linear filter is where the value x_i in a series, i = 1, 2,..n is replaced with the (weighted) average of its neighbouring data. The two-dimensional sweeping window as proposed in chapter two will now be replaced by a 'tangent' window, w_i of size three. The value of the nucleus, $\eta_i = x_i$, can then be computed from the neighbouring data values. Figure 3.2.1 shows a graphical illustration of such a one-dimensional sweeping window, at point x_i. The nucleus value at every function value $f(x_i)$, i = 1, 2,..n is replaced with the value

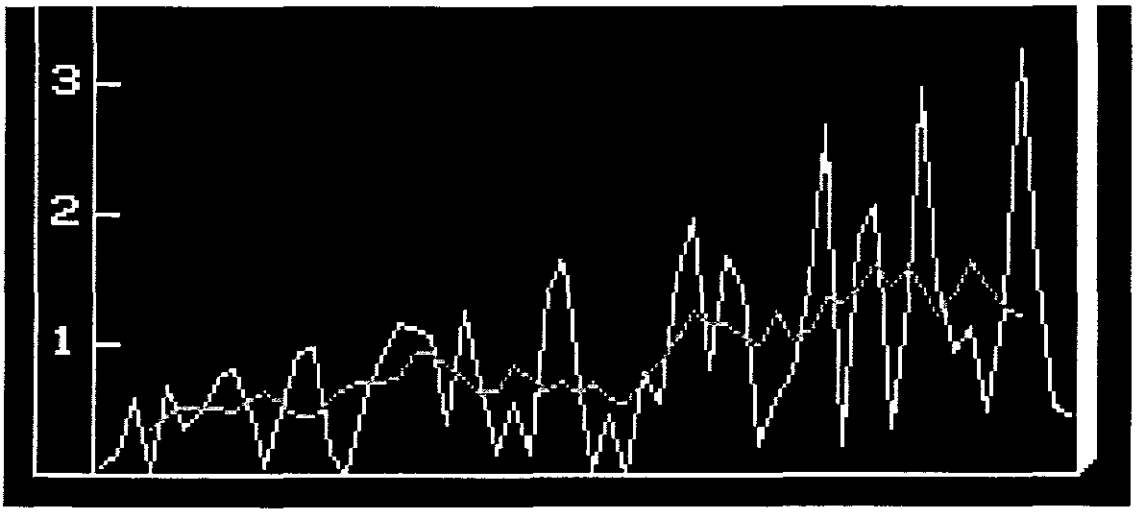
$$\eta_i \leftarrow (f(x_{i-1}) + f(x_i) + f(x_{i+1}))/3.$$

Screendump 3.2.2 shows the effect of a linear filter of window size three on a section of an oscillating function.



Screendump 3.2.2 Average filter, with window size = 3

n screendump 3.2.2 it is obvious that the filter follows the original function quite fficiently as long as there is some degree of stability (monoticity) in the data it operates n. Sudden extremities or directional changes in a function cannot be handled efficiently nd the situation worsens with larger window sizes as shown in screendump 3.2.3.



Screendump 3.2.3 Average filter, with window size = 7

In figure 3.2.3 the window size is changed to seven and the original flow of data is clearly lost.

There are many instances where a linear filter would do more harm than good. A linear filter (or low-pass filter) will normally work quite well on data which has high frequency noise, while the basic function is changing slowly. If the wave was contaminated with erratic noise, i.e. energy surges like the indicated portion of the example in figure 3.1, a non-linear smoother would be a better option to use. The smoother will eliminate noise as far as possible, while the filter would have a *smearing*[1] effect which could drastically effect the quality of the original data.

Smoothers are therefore designed to protect original data, while removing outshooters and still preserving sharp discontinuities in the waveform. It is common practice to first sweep the original data with a smoother before any linear filters are applied. Obviously tests must be built into the algorithms to check for optimality of the data quality, because it is a known fact that repetitive smoothing or filtering may do more harm than good to the original data [55,69].

---

[1] See §4.1 for smearing in images.

3.3 1D Median smoothers

Many authoritative papers were published over the years on the theory of median smoothers. It is worth while to mention the work of Mallows [58] which still forms the basis of many non-linear applications today.

Median smoothers are popular due to their relatively simplistic and efficient operation. Consider a quantified signal of length n, $\mathbf{x} = \{x_1, x_2, ..., x_n\}$[1], and a conventional moving window, $w_i = \{x_{i-k}, x_{i-k+1},....,x_i,....,x_{i+k-1}, x_{i+k}\}$, which is moved over the original x_i values to produce the output sequence, where k is a measure used to adjust window size:

$$y_j = (\mathbf{M}x)_j, \quad j = k+1,k+2,.., n-k.$$

Our notation will refer to the subset:

$$\mathbf{x}(b,e) = \{x_b, x_{b+1}, ..., x_{e-1}, x_e\}$$
$$\mathbf{x}(b,e) \subset \mathbf{x}$$

The median, $\mathbf{M}(i-k,i+k)$, at the point x_i will hence refer to the 2k+1 points in the window, w_i.

If k = 1, for instance, a moving window of size three will determine the new y_i values;

$$\text{each } y_i = (\mathbf{M}x)_i,\ i = 1, 2,..n \text{ and}$$
$$(\mathbf{M}x)_j = \text{med}\{x_{j-1}, x_j, x_{j+1}\},\ j = 2, 3,...,n-2, n-1.$$

---

[1] Note that vectors are indicated as bold, small letters, while operators are presented as bold, upper case letters.

The above data points can also be presented as:

y = M[x] , where y is the output sequence and M is the median smoothing function operating on x.

$\mathbf{y} = \{y_1, y_2, ..., y_n\}$

The following four definitions are often encountered in signal processing and will be used throughout this chapter [31]:

Definition 3.3.1

A *constant neighbourhood* is at least k+1 consecutive identical points such that the constant neighbourhoods and edge together is *monotone*. (See figure 3.3.1 for an example of a monotone increasing function.)

Definition 3.3.2

An *edge* is a monotonic region between two constant neighbourhoods.

Definition 3.3.3

An *impulse* is a constant neighbourhood followed by at least one, but not more than k points which are then followed by another constant neighbourhood having the same value as the first neighbourhood.

Definition 3.3.4

An *oscillation* is a sequence of points which is not part of a constant neighbourhood, an edge, or an impulse.

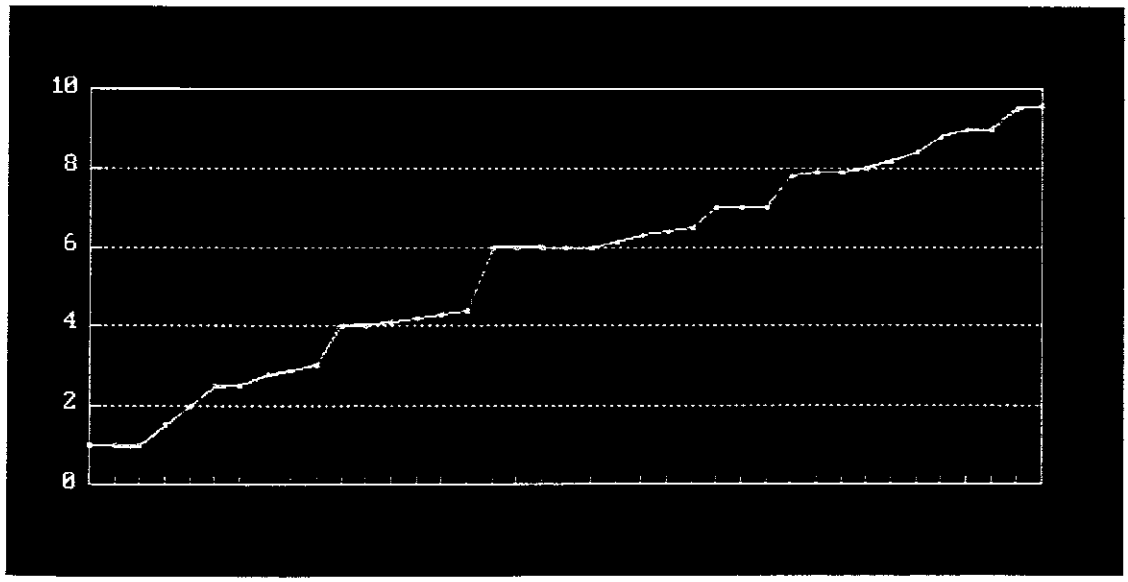


Figure 3.3.1 *A monotone increasing function.*

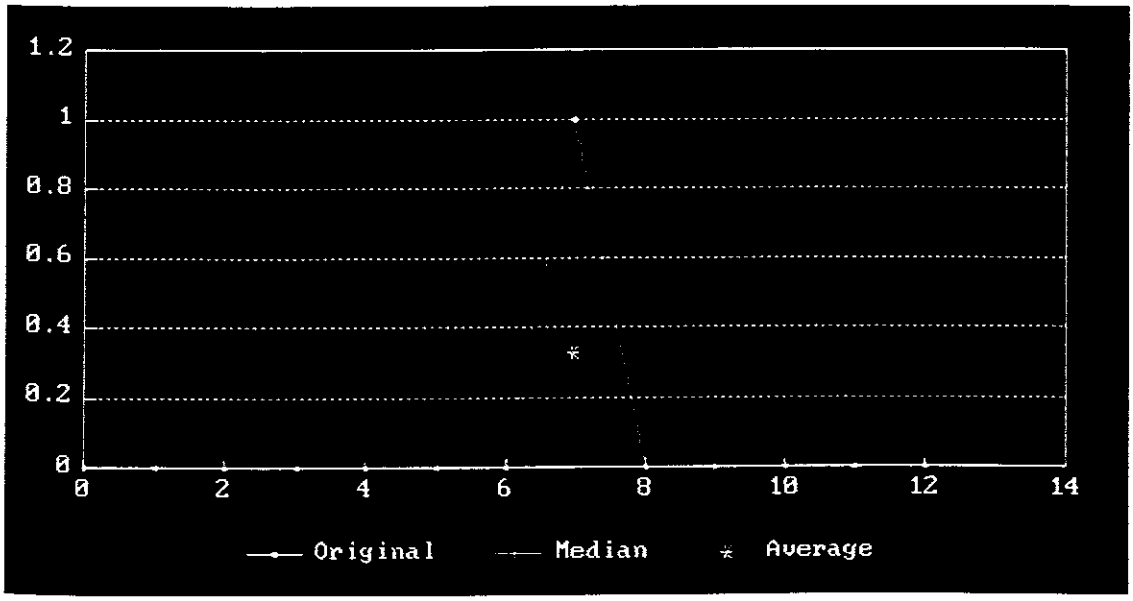


Figure 3.3.2 A single outshooter.

Figure 3.3.2 shows the effect of a median smoother of width three on a single spike acting on a constant function, while figure 3.3.2 shows the effect on a monotonically increasing function.

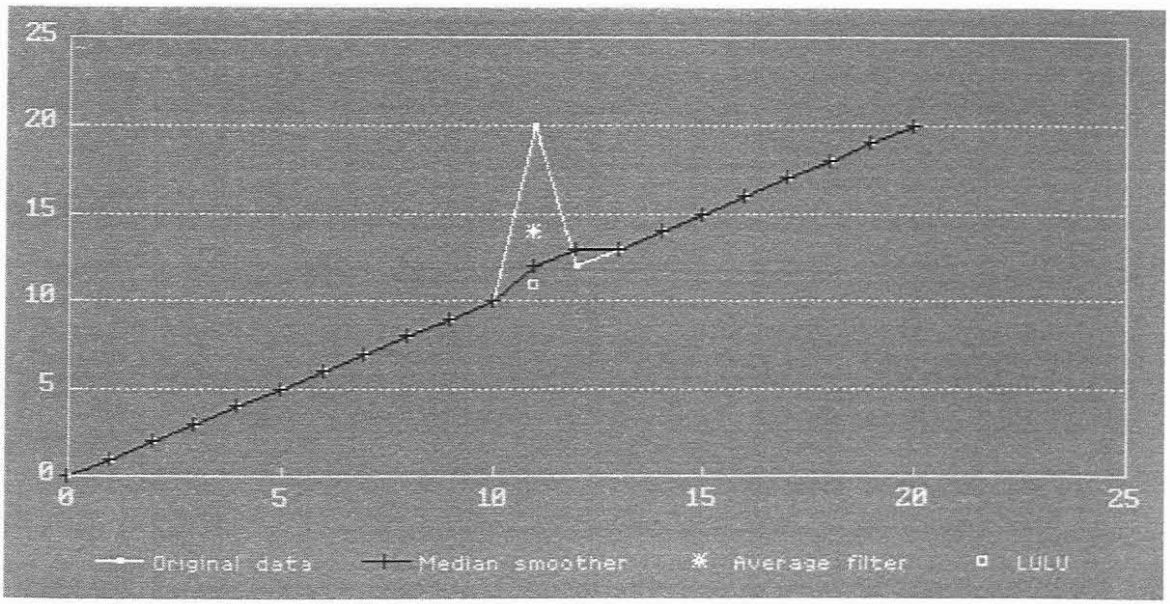


Figure 3.3.3 A single outshooter on a slope.

Note that the median filter of width three does not succeed to fully remove the spike. A **LULU** algorithm, as defined in §3.4, effectively removes the spike and restores a valid point.

Figures 3.3.4 and 3.3.5 indicates the results of median filtering with window sizes of varying width. If the binary data in figure 3.3.4 is sweeped with a median filter with window width three, there will be no change in the output data. If a window width is set to five, the result will be as in figure 3.3.5. Dummy values or 'padding' are added at the beginning and the end of the original sequence to allow for the smoothing of end points. Normally padding has little effect on the output due to the vast stream of points under observation.

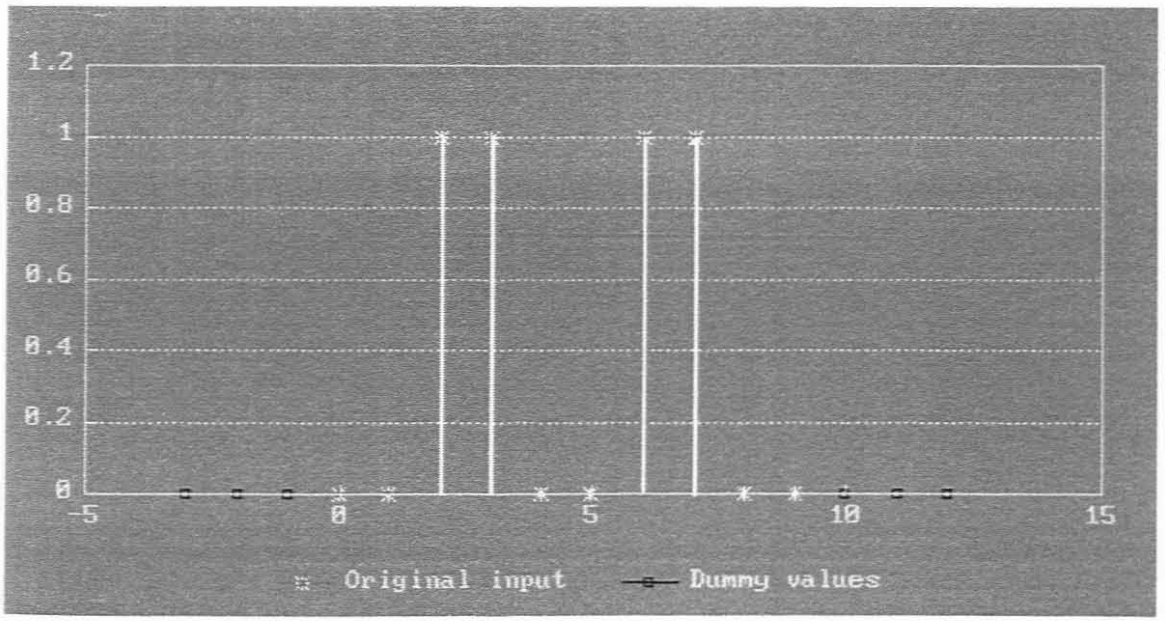


Figure 3.3.4 Original data

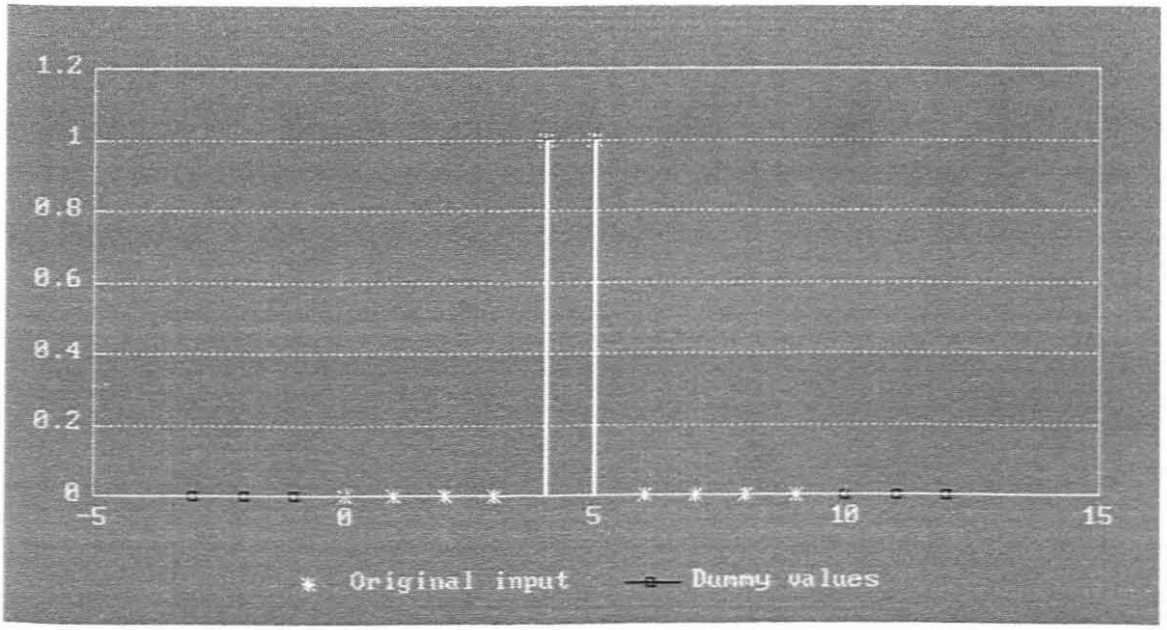


Figure 3.3.5 Median smoother using a window of width five.

The smoothing function **M** is a typical *rank-based selector function*, because the smoothed value at each output point y_i is selected purely on rank order [44] of the values in the window w_i.

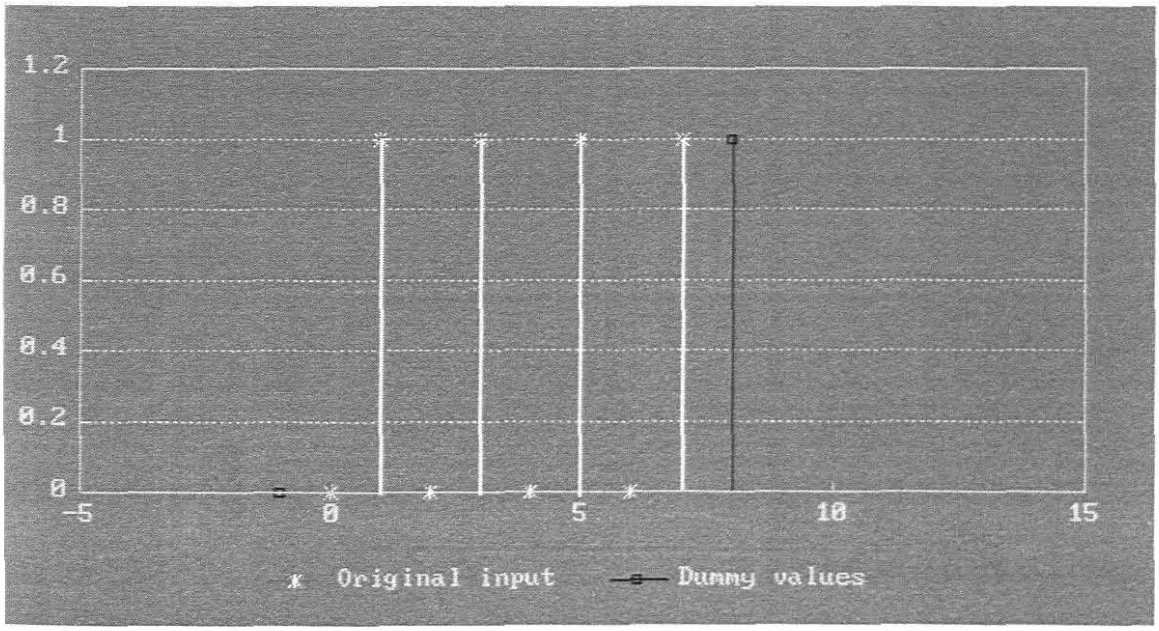


Figure 3.3.6 Original signal

Figures 3.3.6 - 3.3.9 shows the result of repeated median smoothing.

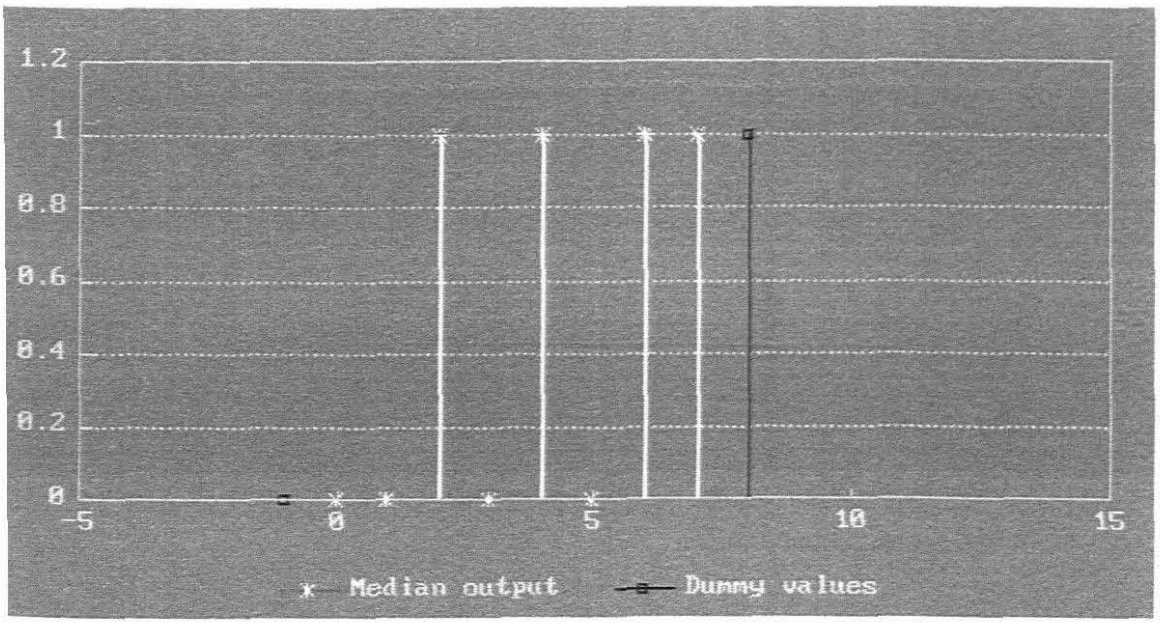


Figure 3.3.7 Signal after one median sweep.

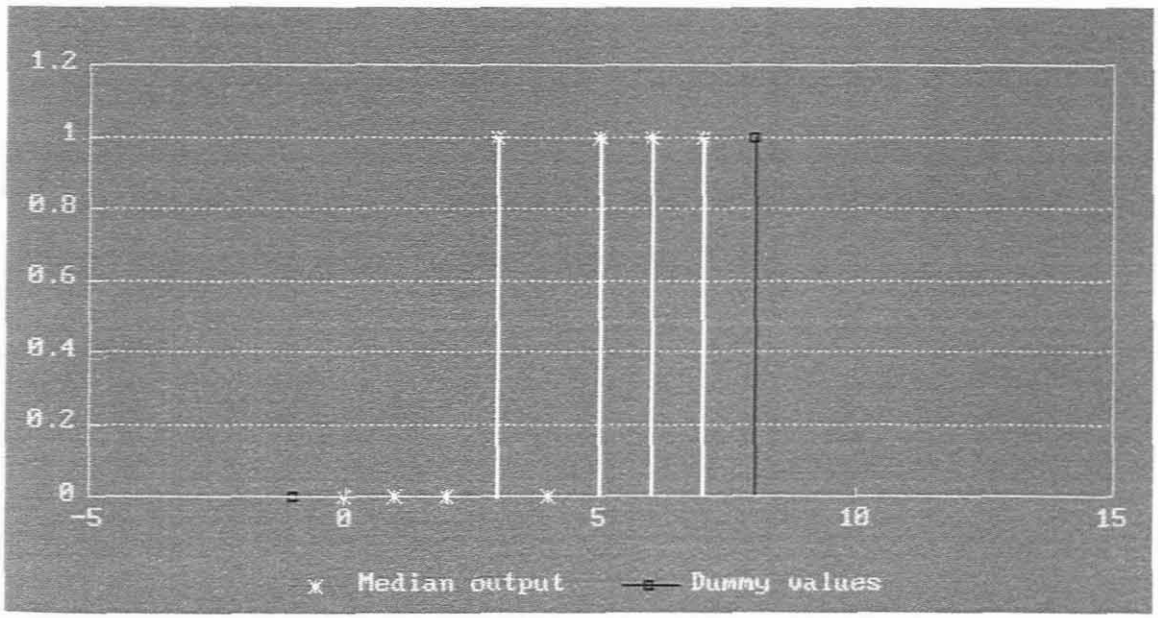


Figure 3.3.8 Signal after two median sweeps.

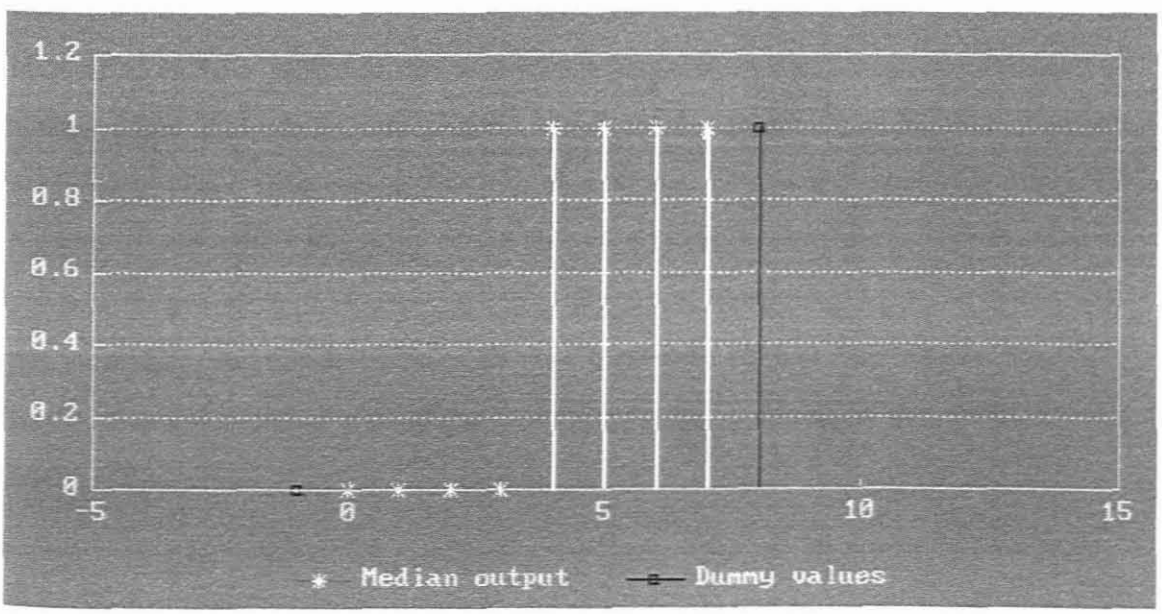


Figure 3.3.9 Final median sweep arrives at *root*.

The idea behind repeated filtering with a running window of a fixed size, is to arrive at the *root* of the sequence. At this stage the signal is *invariant* to successive smoothing, i.e. any filtering after this stage would not change the resultant data.

It is also interesting to note that the roots of **x**, when data is smoothed repeatedly with different window sizes, are not necessarily the same [76,88].

Figure 3.3.10 shows the comparison of a median smoother and a average filter, with a running window of width five.

Although median smoothing is so widely used, it suffers from some deficiencies, as shown in [21] and [82]. The median is an attractive smoother to use due to the ease of its statistical interpretation, but its behaviour becomes complex when repeatedly applied to an image[1].

In addition to these problems, the median tends to converge to suboptimal solutions in certain cases and global convergence need not be achieved, even with repeated application [72]. The reason for this observation is that the algorithm place the impulse at one of the endpoints of the selection interval, and then selects the middle value as the new smoothed value, which might not be the optimal value. It is therefore not surprising that intensive research was conducted in this area which eventually produced a novel set of algorithms based on *partial ordering* of the operators in a lattice [80].

---

[1] Chapter five illustrates some mathematical consequences, while chapter seven shows the deterioration of an image with repetitive median smoothing.

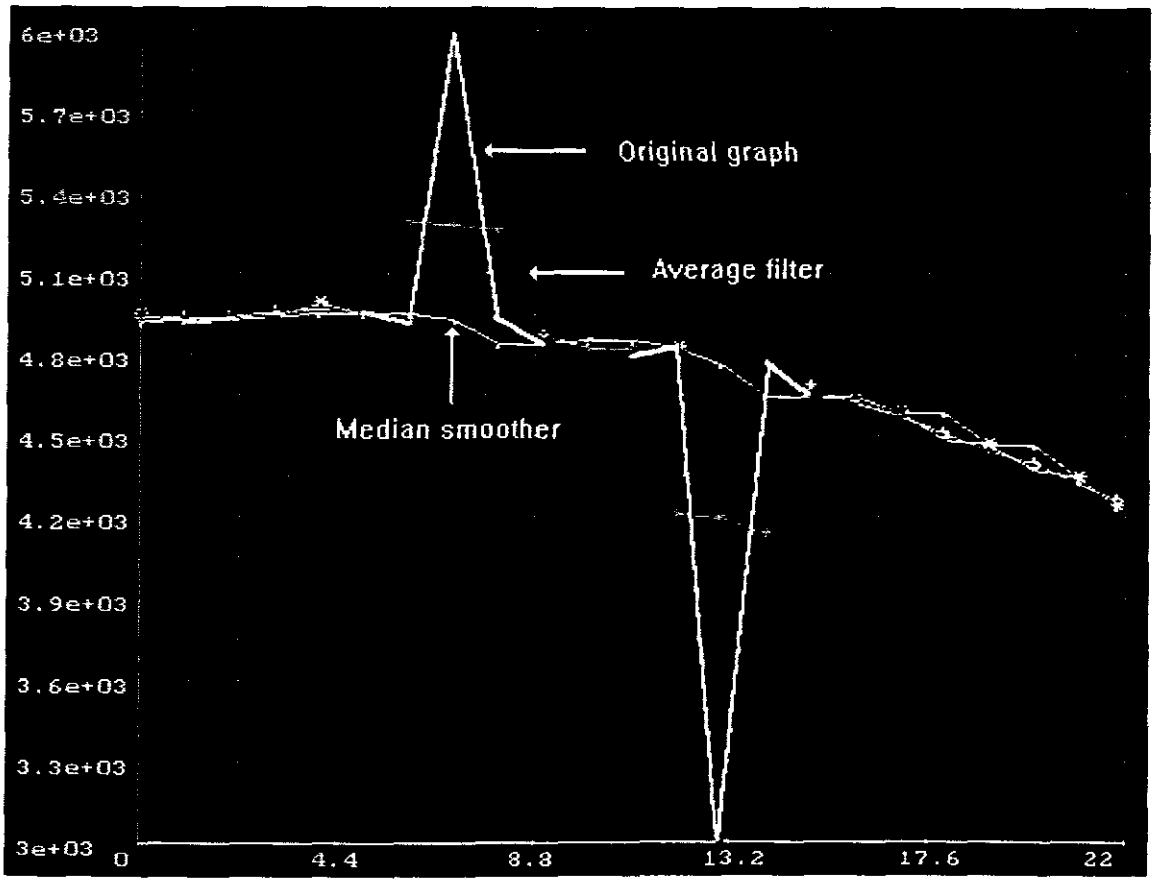


Figure 3.3.10 Comparison of a linear filter and a median smoother.

3.4 1D LULU algorithms

The **LULU** class of non-linear smoothers introduces new variations in non-linear smoothing which can in some instances be shown to improve situations where the median smoother falters [81].

Consider, as before:

y = S(x) , where **x** is the input sequence and **y** is the output sequence and **S** represents the smoothing function.

As an alternative to using median operators in the active window, a *running minimum*[1] can be employed to remove upward pop's in a monotonically increasing function [80,82]. Note that this procedure will widen a downward pulse, as shown in figure 3.4.1.

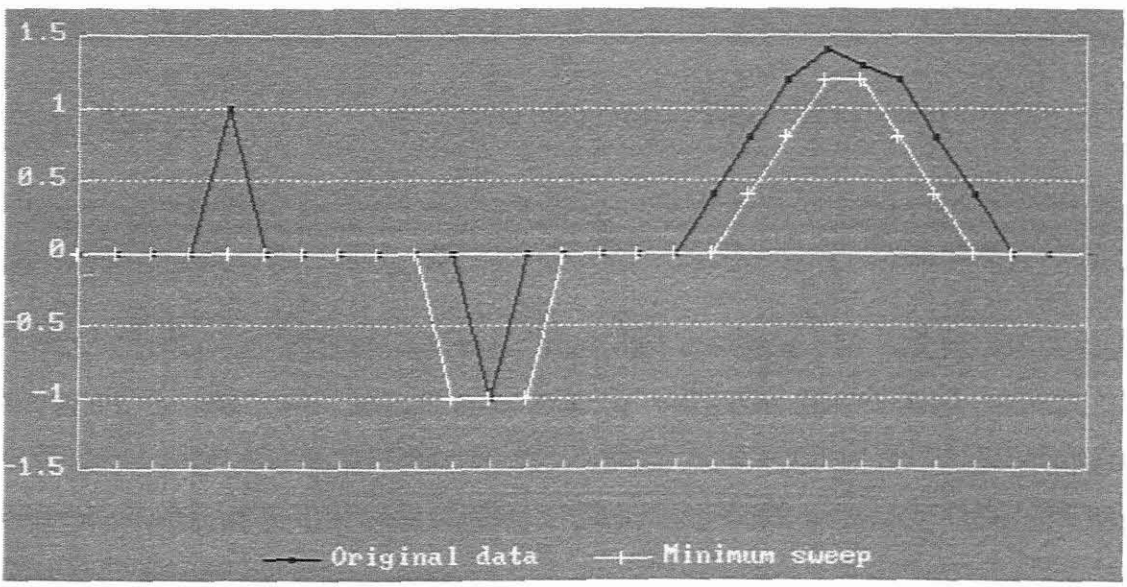


Figure 3.4.1 The effect of a minimum sweep on the original data.

[1] A running minimum operator is defined by a rank-order selector which selects the smallest (lowest ranked) element from the running window.

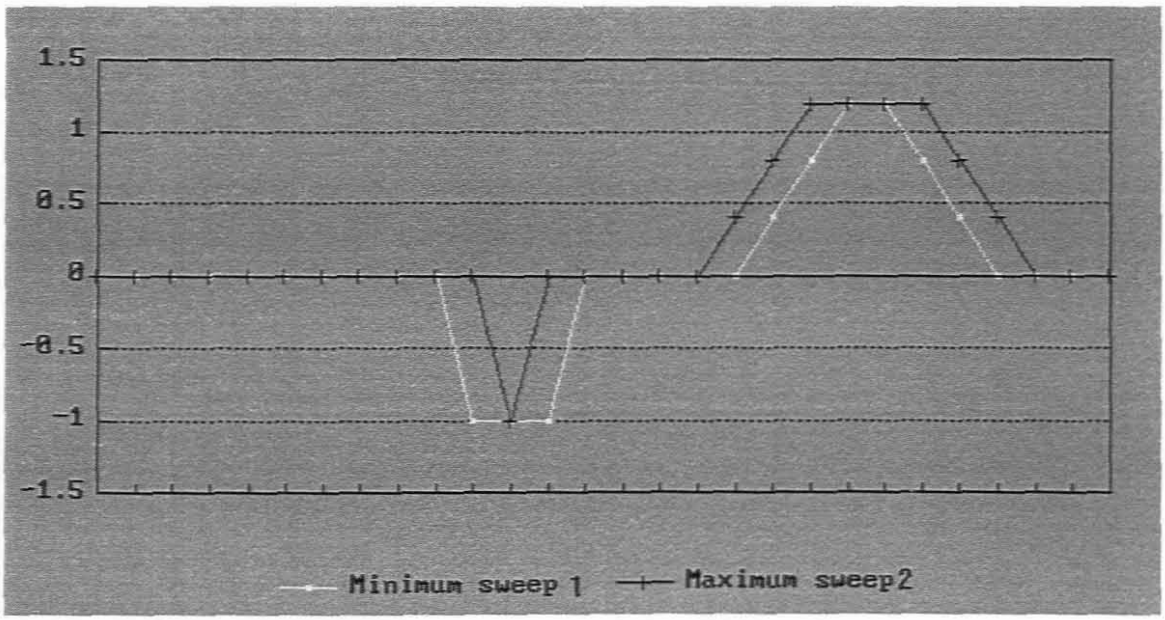


Figure 3.4.2 A running minimum, followed by a running maximum sweep.

It should further be pointed out that a running minimum, followed by a running maximum operator, will restore monotonically increasing (decreasing) parts of a function to its original state. Naturally this depends on the size of the window, w_i, as illustrated in figure 3.4.2. Unless stated differently, a window size of three[1] will be used in the remainder of this chapter.

The basic pair of unsymmetric operators, **U** and **L** are defined as follows [80]:

$(\mathbf{L}x)_i = \{\max\{\min \mathbf{x}(i-k,i), \ldots, \min \mathbf{x}(i,i+k)\}\}$, where $\mathbf{x}(s,t) = \{x_i;\ i \in [s,t]\}$ and $x_i \in \mathbf{x}$.

Similarly,

$(\mathbf{U}x)_i = \{\min\{\max \mathbf{x}(i-k,i), \ldots, \max \mathbf{x}(i,i+k)\}\}$

where x_i is the centre point of the running window, w_i.

---

[1] $k=1$

U will hence have the effect of removing *downward* pulses, leaving upward pulses as is. It will further retain upward/downward trends as shown in figure 3.4.3.

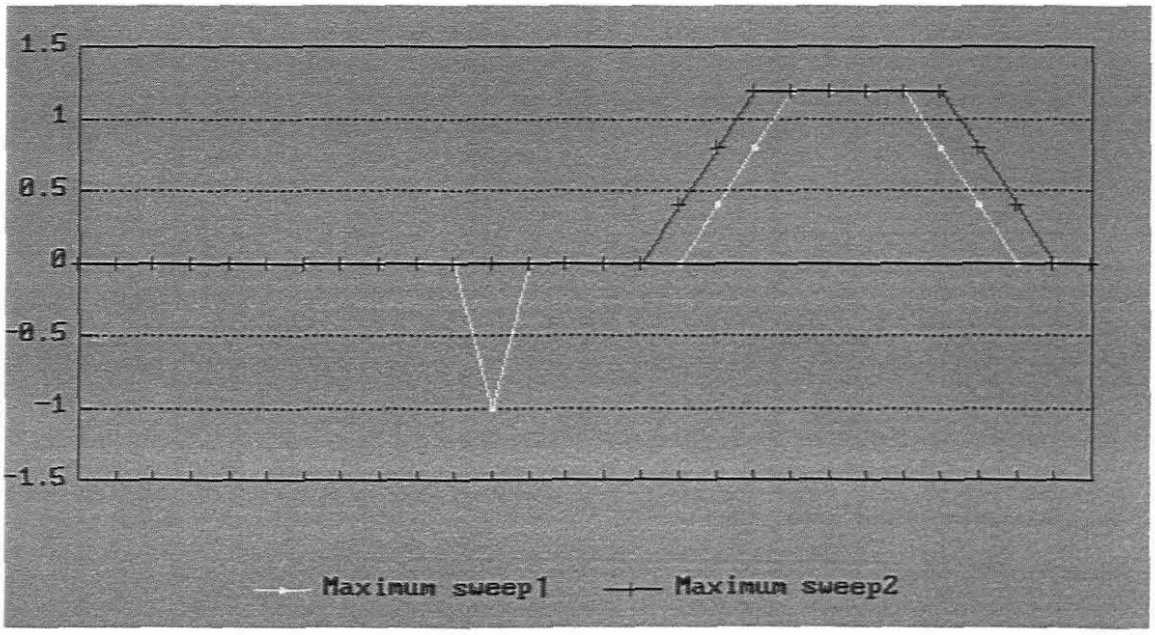


Figure 3.4.3 A minimum, followed by a maximum, followed by a maximum

Note that the *complete*[1] algorithm in effect means a minimum sweep, followed by a maximum, followed by another maximum and finally a minimum sweep. Also note that the process can be made more efficient by replacing the two maximum sweeps of window width n+1 by a single maximum sweep of window size 2n+1 [62].

---

1 The complete algorithm is defined by a sequence of smoothers in a specific order to obtain the desired results.

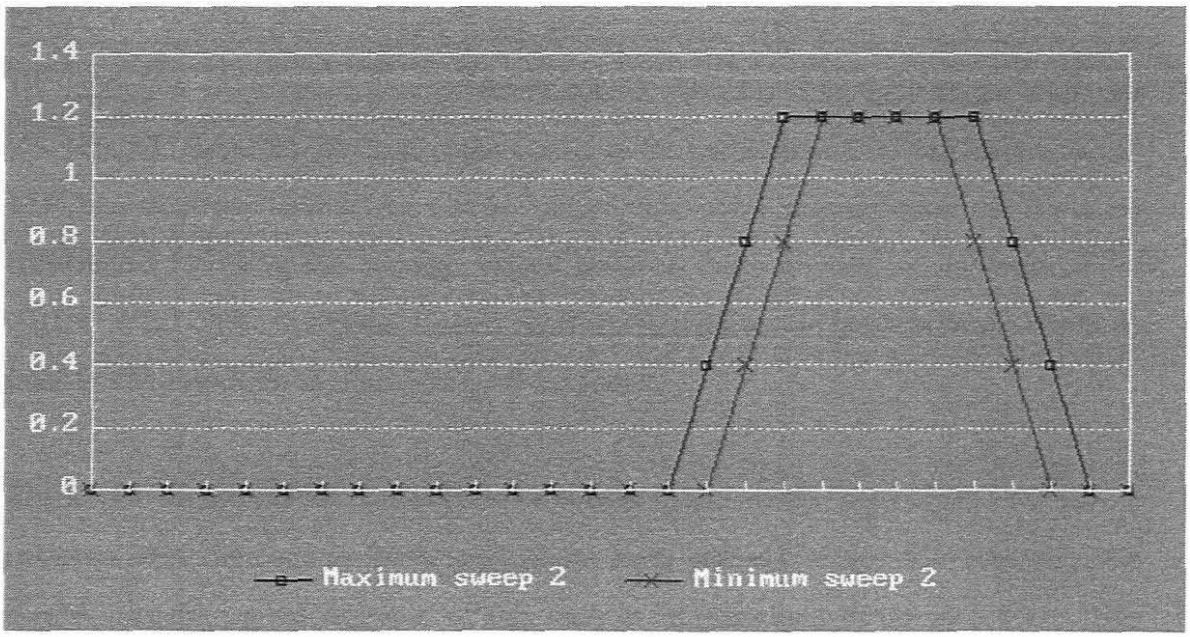


Figure 3.4.4 $\min_2 \max_2 \max_1 \min_1$ sweep (**UL**)

Composition of smoothers was used in §3.3, where the median smoother **M** was repeatedly used on output series of a previous median smoother. A similar operation can be constructed by letting **U** operate on the output of **L** and vice versa. This overcomes the deficiencies of the **U** and **L** operators on their own.

The smoothers **UL** and **LU** can be constructed to remove both upward and downward pop's. **UL** will hence be a running minimum operator, followed by a running maximum operator, followed by a second maximum operator and finally followed by a minimum operator, i.e. $\mathbf{UL} = \mathbf{U}(\mathbf{L}x)_i$. **LU** can be constructed similarly as shown in [80].

Note that an optimal window size can be designed such that k is chosen as *at least* the maximum number of expected outliers [62].

If the correct window size is used, it can be shown that $\mathbf{U}(\mathbf{L}x)_i - \delta < x_i < \mathbf{L}(\mathbf{U}x)_i + \delta$, i.e. valid data is contained in confidence bounds where δ is an appropriate tolerance [61,81]. **UL** therefore gives an upper bound to the reliability of the data series, while **LU** gives the lower bound. An easy and appropriate non-linear smoother can hence be constructed by using the average values between these bounds: x_i is replaced by $(\mathbf{U}(\mathbf{L}x)_i + \mathbf{L}(\mathbf{U}x)_i)/2$ if x_i is not a valid value in the sequence to be smoothed. We shall refer to this algorithm as the **LULU_1D** smoother.

The **LULU** algorithm can be shown to be similar to the median algorithm in the way it removes noisy data and in some applications even perform better. The removal of a single spike is a good example of this fact, as is illustrated in figure 3.3.3. The median algorithm *suppresses* the energy of a flyer, while the **LULU_1D** smoother actually removes the flyer due to the specific order **L** and **U** is used.

It is sufficient to summarise some of the most important characteristics of these operators at this stage[1].

3.4.1 Smoothers can be compared (ordered) by checking the output against the input series. The normal $=, <, >$ relations applies as shown in articles [79] and [80].

It can for instance be shown that $\mathbf{L} \leq \mathbf{M} \leq \mathbf{U}$ [2].

3.4.2 **L** an **U** can shown to be idempotent [81]. This is a very desirable factor, because if the smoother is repeated on its own output, nothing changes: $\mathbf{LL} = \mathbf{L}^2 = \mathbf{L}$ and $\mathbf{UU} = \mathbf{U}^2 = \mathbf{U}$.

---

1 Formal mathematical proofs are presented in chapter five and further practical tests are reported in chapter seven.

2 See the references and chapter five for a summary.

Unlike some other computationally efficient smoothers, where repetitive runs are needed, the **LULU** algorithm will find a consistent solution to the smoothing problem in a predictable number of steps in a single application.

3.4.3 UL and LU can also shown to be idempotent.

3.4.4 LU and UL can be processed in a single array process by noting that LU(X) = -UL(-X).

The idempotency feature[1] and the relatively simple structure of **LULU** algorithms makes it a excellent candidate for various parallel algorithms. This is further investigated in chapter eight. Experimental results achieved with these smoothers are reported in [62].

---

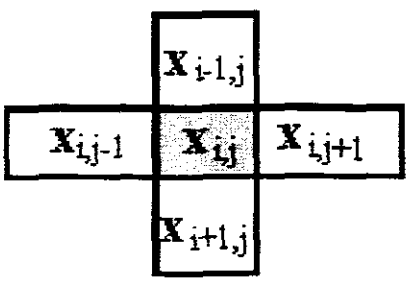
[1] Chapter four has some practical examples of this phenomenon, while chapter five illustrates the mathematical consequences.

CHAPTER FOUR

Two-dimensional LULU smoothers

4.1 Simple two-dimensional linear filters

Two-dimensional linear filters are popular for many applications in signal processing. They can vary from simple neighbourhood convolutions as shown in chapter two, to advanced methods such as Fourier analysis and Kalman Filters [19,44].



Depending on the data, linear smoothers (low-pass filters) can be used when the image is not contaminated with high-frequency energy, such as the noise described in §3.1. This is usually the case when the noise is Gaussian and the Central Limit Theorem is applicable.

Figure 4.1.1 Five-point window

For some applications linear filters will do more harm than good because of the digital nature of the input data. In practice, noisy data can clearly be seen on a television image

as *spots*[1] when an electrical appliance starts or when a car with faulty ignition suppression passes nearby.

If a conventional two-dimensional five-point sweeping window is chosen, as illustrated in figure 4.1.1, only the pixels nearest to the central nucleus, $\eta = x_{i,j}$, are used. To illustrate how two very basic linear filters can be formulated, each nucleus pixel, $x_{i,j}$, is replaced with the values of its surrounding pixels according to one of the following formulae:

4.1.2 $x_{i,j} \leftarrow 0.25*(x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1})$

4.1.3 $x_{i,j} \leftarrow 0.2*(x_{i-1,j} + x_{i+1,j} + x_{i,j} + x_{i,j-1} + x_{i,j+1})$, with $0 \leq i \leq n, 0 \leq j \leq m$

Let $x_{i,j}$ refer to a pixel at the position, row i, column j, of the two-dimensional nxm matrix image, **X**[2]. In the first example, the nucleus pixel is replaced with the average of the surrounding four pixels from the five-point window (directions north, east, south and west), ignoring the nucleus[3]. The advantage of the averaging method 4.1.2 above 4.1.3 is that noise energy in the nucleus is avoided at the [i,j] position. The surrounding pixels will however still show the effect of the outshooter. The set of **LULU** programs, as demonstrated in chapter six, has formula 4.1.3 included as an example to show what happens when high energy noise is filtered with a method which is not suitable. Screendump 4.1.4 has an image on the left (**X** indicating the original input image to be filtered) which was speckled with random black dots. The right-hand side of the screendump, **P**, shows the result of the linear process, Φ, as formulated in 4.1.3. Screendump 4.1.5 shows an enlarged section of the original image, clearly indicating two

---

1 Another common noise type often seen on television screens, is a single noise line or cluster of lines, displayed horizontal or diagonally.

2 Note that the image, **X**, presented by a matrix, X, is always printed in capitalized bold letters, while the matrix self is presented in ordinary print.

3 If the linear process is Φ, the input image is **X** and the output image is **Y**, the filtering operation can be described as: $\mathbf{Y}_\Phi = \Phi(\mathbf{X})$. In the non-linear case, the original image, **O**, will be smoothed to a final picture, **P** : $\mathbf{P}_S = \mathbf{S}(\mathbf{O})$. Note that O, X and Y are the matrices from which the images **O**, **X** and **Y** are generated.

pops[1] and the corresponding result on the right-hand side after the linear algorithm was applied. It is clear that this linear process suppresses the noise, but actually *smears* the resultant picture. A high energy pop will be averaged to itself and its four surrounding neighbours. This result is also clearly visible when the Windows Paintbrush® ***VIEW*** function is applied on such a filtered image.

Screendump 4.1.4 A five-point average filter applied on a very famous woman.

Screendump 4.1.5 Zoom results **X** $\mathbf{P_\Phi = \Phi(X)}$

A linear filter is therefore not a good candidate for the removal of spot noise and non-linear methods will have to be investigated for this purpose. To overcome the problems experienced with linear methods, it is imperative to employ non-linear pre-filters to remove spot noise before any other enhancement is done to the image.

---

[1] 'Pops' refers to noise speckles in 2D data.

Once the robustness and usefulness of the **LULU_1D** algorithm was properly proven and tested, it was necessary to extend and investigate this class of smoothers for two-dimensional applications with the aim of eliminating noise with minimal damage to the original image.

The following sections summarise the evolution of a new class of non-linear two-dimensional smoothers which were developed. As an initial point of departure the **LULU_1D** algorithm, as described in chapter three, was investigated for two dimensional application. The fundamental question was to prove that horizontal **LULU_1D** sweeps, followed by vertical **LULU_1D** sweeps give similar results as sweeping vertically first and then horizontally. Intuitively success is guaranteed because an image can be viewed as a set of horizontal (or vertical) scan-lines. Unfortunately, as with many scientific problems (and Nature itself), experimentation soon showed that although these algorithms work well for many practical examples, they are not computationally and mathematically as consistent as algorithms that were designed from the outset for two dimensions. These algorithms are however included in the set of **LULU_2D** programs and have shown to be an improvement to even the median applications for some specific applications[1].

A more promising method to pursue was to look at neighbourhood regions, as described in chapter two. This led to the discovery of new classes of two-dimensional **LULU** algorithms [80].

4.2 Morphological systems

Mathemathical morphology is a set-theoretically based methodology for geometrical analysis of image processing. The class of non-linear morphological systems is widely

---

[1] See chapter seven for practical examples.

used in many computer vision applications, particularly to represent and extract shape in multidimensional systems. Although some of the first work on morphology was done in the sixties, Nakagawa and Rosenfield [69] presented the first studies where max-min operators were used on gray-level images for noise reduction.

The basic idea is to use a disk-like window to sweep a binary image, replacing each pixel with the logical *AND* (shrinking) of its immediate neighbours [59, 69]. The opposite of this process involves the *OR* operand and is referred to as the 'expanding operation' in the literature. If these basic operators are replaced with maximum or minimum footprints of a sweeping window on a gray-scale image, the so-called '*erosion*' or '*dilation*' effects are achieved [59].

If the one-dimensional **LULU** structures are to be investigated for two-dimensional application, it would naturally require extensive refinement of the morphological systems as described in the literature. A novel way of presenting a two-dimensional **LULU** smoothing method is to correlate the theory of **LULU_1D** with the ideas presented in §4.1.

4.3 LULU 2D_5W design principles

Let **O** be an nxn input matrix (*image*) to be smoothed. We can define a 3x3 sweeping window similar to the one used in 4.1.2:

4.3.1 $w_{i,j} = \{o_{i,j}, o_{i,j-1}, o_{i,j+1}, o_{i-1,j}, o_{i+1,j}\}$ with $0 \le i,j \le n$

where $o_{i,j}$ is the nucleus to be processed within the sweeping window, $w_{i,j}$, at the matrix position [i,j].

A *floor* **LULU_2D** operator can now be defined as:

4.3.2 $$(Fo)_{i,j} = \max(\min(o_{i,j}, o_{i,j-1}), \min(o_{i,j}, o_{i,j+1}), \min(o_{i,j}, o_{i-1,j}), \min(o_{i,j}, o_{i+1,j}))$$

A *ceiling* **LULU_2D** operator can be defined similarly as:

4.3.3 $$(Co)_{i,j} = \min(\max(o_{i,j}, o_{i,j-1}), \max(o_{i,j}, o_{i,j+1}), \max(o_{i,j}, o_{i-1,j}), \max(o_{i,j}, o_{i+1,j}))$$

The image can now be smoothed, in succesive horizontal[1] scanline sweeps as indicated in the program listings §6.6.6. The notation used for a floor sweep is **F**. A similar notation is used for the ceiling sweeps, **C**. The effect of **F** is to remove upward impulses[2], whilst **C** does the opposite. It therefore seems logical that a *complete* smoother must therefore include both the *floor* and *ceiling* processes to remove noise as effective as possible.

Rohwer [81] describes the mathematical implications of this new approach for two-dimensional smoothing in his article, ***'LULU-operators for two-dimensional data'***. Idempotence of the **F** and **C** operators can be proven (§5.2) and a *true* **LULU** structure has been defined.

A complete two-dimensional filtering process will hence require a total of n-2 horizontal **F** sweeps of the image followed by the same number of **C** sweeps before the picture is cleaned of spot noise.

At this stage it is interesting to thoroughly test the effect of idempotency for two-dimensional smoothing. In practice this implies the smoothing of a vast number of images with basic geometrical features, all contaminated with erratic noise. A **LULU_2D/5W** algorithm[3] can be defined according to the formulae 4.3.4 or 4.3.5:

---

1 Because the operators **F** and **C** have a rotational invariance of 90°, it does not matter whether the sweeps are performed horizontally or vertically for all practical reasons. Also note that the term *sweep* is often used instead of *scanline sweep*.

2 Assume the sweeping window to move on the plane of the two-dimensional function.

3 **LULU_2D/5W** indicates a **LULU** algorithm based on a five-point two-dimensional sweeping window as illustrated in fig.4.1.1.

4.3.4 P = FC(O) and

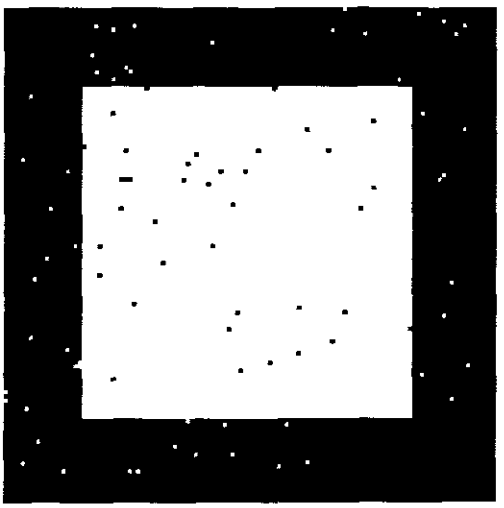
4.3.5 P = CF(O), where the resulting dimension of P is $(n-2)^2$

As **O** was already defined as the nxn input matrix, contaminated with noise, **P** will always refer to the final picture after the smoothing process. 4.3.4 defines an algorithm where the image is first smoothed with the ceiling smoother, **C** (as defined in 4.3.3), and the resultant (intermediate) picture, $\mathbf{P}_c = \mathbf{C}(\mathbf{O})$, is then smoothed with algorithm 4.3.2 to achieve the final output $\mathbf{P} = \mathbf{F}(\mathbf{P}_c) = \mathbf{FC}(\mathbf{O})$. It can be shown that $\mathbf{P} = \mathbf{CF}$ and $\mathbf{P} = \mathbf{FC}$ produces a similar result if the noise levels are within reasonable bounds for *erratic noise* as described in §3.1.

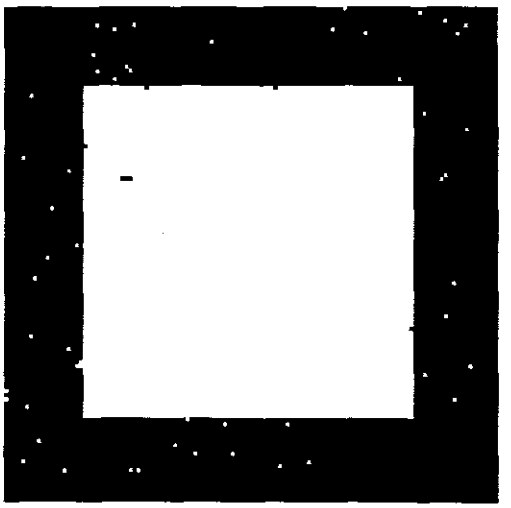
As an example to show this process, a binary image, **O**, was constructed as a binary image with the *bottom* (black) part being zero and the rectangular *top* part set to one (screendump 4.3.6). The image was then sprinkled with random noise ('*salt and pepper noise*'). Note that the flyers are of opposite magnitude to the area where it was positioned. The rectangular section is thus constructed from pixels of value one with superimposed noise, shown as the black spots. Screendump 4.3.7 shows the result after a **C** sweep with a five point window was performed on the full matrix. The original features of the image remains unaltered, but most of the impulsive noise in the rectangular *(ceiling)* section is eliminated. Obviously noise will remain, depending on the noise intensity, or where noise pixels are next to each other and the window size is not adequate. This fact is observed in figure 4.3.7. Finally $\mathbf{P}_c$ is smoothed with a *floor* sweep to eliminate the noise in the remaining part of this intermediate picture and the result is illustrated in screendump 4.3.8.

Note that, as with other non-linear smoothers, the boundary between *floor* and *ceiling* involves the ambigious nature of the smoothing process [31]. The positive feature of

LULU smoothers is the fact that the boundary is left unaltered, no *edge shift* [36,74] occurs, thus preserving the original image.



Screendump 4.3.6 **O**

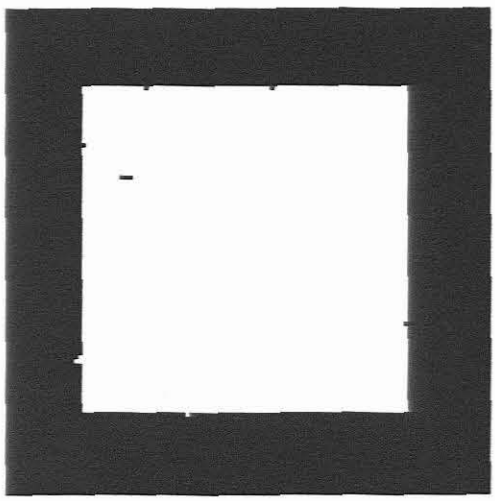


Screendump 4.3.7 $\mathbf{P_c = C(O)}$

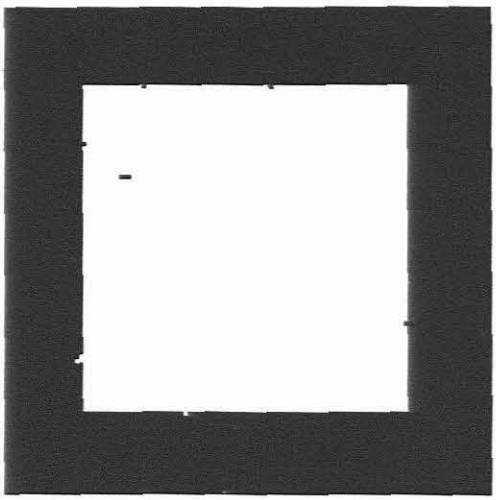
The effectiveness of any smoother naturally depends on the density of the noise spots, as well as nature and distribution of noise present in **O**. The noise remaining after a pre-smoother has been applied can be treated with some applicable process, such that the image is restored as accurately as possible[1].

Screendumps 4.3.8 and 4.3.9 visually displays the results of algorithms 4.3.4 and 4.3.5.

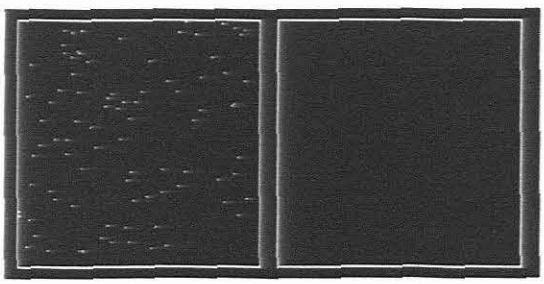
[1] See chapter seven for practical illustrations.



Screendump 4.3.8 **P = FC(O)**



Screendump 4.3.9 **P = CF(O)**



Screendump 4.3.10 **$P_f = F(O)$**

Screendump 4.3.10 shows a blank image scattered with random noise. The interesting fact is that this image only needs an **F** sweep, because there is no binary clustered feature necessitating a ceiling sweep.

A **C** sweep will therefore not change the image and $P_c = C(O) = O$.

Screendump 4.3.11 Original image, **O** **Screendump 4.3.12** **$P_c = C(O)$**

As a final example, screendump 4.3.11 has the same original image as 4.1.4, but it is contaminated with random noise scattered in all 256 grey levels of the image. Note that **C** again removed the *downward* spikes from the position of the sweeping window in the hyperplane. This explains the fact that only the whiter shade outshooters remain in screendump 4.3.12.

Screendump 4.3.12 $\mathbf{P_c = C(O)}$ **Screendump 4.3.13** $\mathbf{P = FC(O)}$

If image 4.3.11 is smoothed again with **F**, the final picture $\mathbf{P = F(P_c) = FC(O)}$, yields the *completely* smoothed picture in screendump 4.3.13.

Screendump 4.3.14 The results of **LULU_2D/5W** with oversaturated noise.

The usefulness of the **LULU_2D/5W** algorithm has been illustrated for practical application and it is quite clear by analysing the resultant picture that the quality of the original image has been retained.

Screendump 4.3.14 shows the effect of this smoother when too many noise pixels were present in the original image. Remaining noise will thus always be in the direction of the NWES[1] axes, relative to the nucleus pixels.

4.4 LULU_2D/9W design

In this case the nine-point sweeping window is the set of bordering pixels:

4.4.1 $$w_{i,j} = \{ o_{i,j}, o_{i,j-1}, o_{i,j+1}, o_{i-1,j}, o_{i+1,j}, o_{i-1,j-1}, o_{i-1,j+1}, o_{i+1,j-1}, o_{i+1,j+1} \},$$
$$0 \le i,j \le n$$

$o_{i-1,j-1}$	$o_{i-1,j}$	$o_{i-1,j+1}$
$o_{i,j-1}$	$o_{i,j}$	$o_{i,j+1}$
$o_{i+1,j-1}$	$o_{i+1,j}$	$o_{i+1,j+1}$

Graphically this window is similar to the one in §4.4, but with the four diagonal pixels are included as shown in figure 4.4.2. The eight pixels bordering the nucleus, will now have the 45° rotational axes included.

Figure 4.4.2

LULU_2D/9W operators can now be defined as:

4.4.3 $$(Fo)_{i,j} = \max(\min(o_{i,j}, o_{i,j+1}, o_{i-1,j+1}, o_{i-1,j}), \min(o_{i,j}, o_{i,j-1}, o_{i-1,j-1}, o_{i,j-1}),$$
$$\min(o_{i,j}, o_{i,j-1}, o_{i+1,j-1}, o_{i+1,j}), \min(o_{i,j}, o_{i,j+1}, o_{i+1,j+1}, o_{i+1,j}))$$

---

[1] In the same sense as north, west, east and south pixels, as defined in chapter three.

4.4.4 $$(Co)_{i,j} = \min(\max(o_{i,j}, o_{i,j+1}, o_{i-1,j+1}, o_{i-1,j}), \max(o_{i,j}, o_{i,j-1}, o_{i-1,j-1}, o_{i,j-1}),$$
$$\max(o_{i,j}, o_{i,j-1}, o_{i+1,j-1}, o_{i+1,j}), \max(o_{i,j}, o_{i,j+1}, o_{i+1,j+1}, o_{i+1,j}))$$

The characterisation of the nine point **LULU** operator is similar to the five point operator in the way which it eliminates spot noise. The fact that the operators now does a selection on four sub-matrices, rather than on four pixel values, has important significance, as can be seen in screendump 4.4.5. Due to the construction of this filter, the nucleus value is now compared with three pixels in each of the four surrounding surface elements. If the same image, with the same noise distribution in as §3.4 is used, this smoother is obviously superior to the five-point algorithm as far as its noise-cleaning power is concerned.

Screendump 4.4.5 The results of **LULU_2D/9W**

Screendump 4.4.5 illustrates an original image with grey scale noise distributed randomly in all 256 grey shades, cleaned by the **LULU_2D/9W** algorithm. Screendump 4.4.6 shows the resultant noise which can be expected to remain after this algorithm was used in a picture where the original image was oversaturated with noise.

Screendump 4.4.6 The results of **LULU_2D/9W** with oversaturated noise.

4.5 Other LULU_2D structures

The fact that an image seems to be somewhat *embossed*[1] after being smoothed with **LULU_2D/9W**, opens the following questions:

- do there exist any other useful **LULU** structures?
- can *hybrid*[2] **LULU** methods be constructed?
- if so, can they be proven to be idempotent?
- can a 'best' smoother be found for a particular noise distribution?

Although some of the answers to these questions are given in chapters five to eight, it is appropriate to investigate some other interesting algorithms at this stage. The emphasis will be on practical algorithms which can contribute in some pre-defined manner to the problem of noise-reduction.

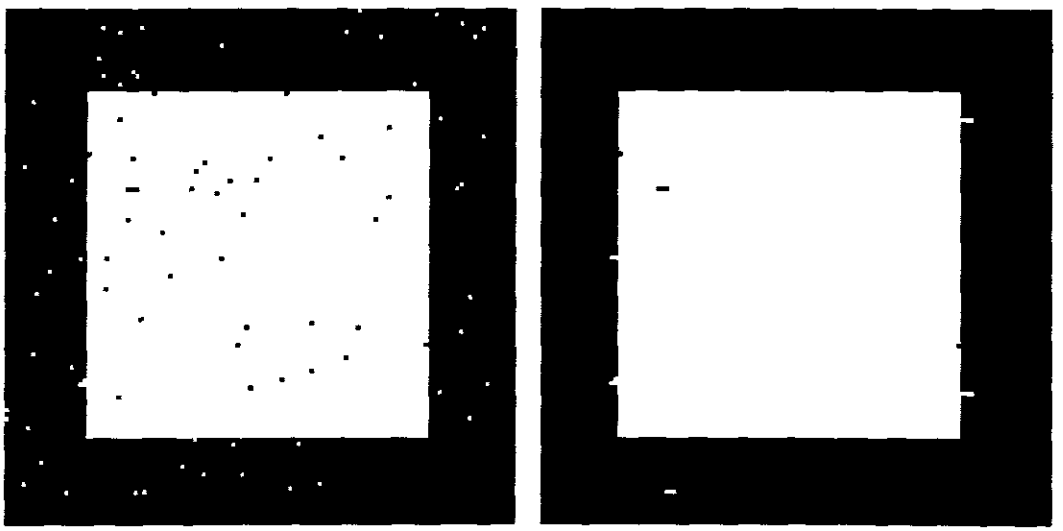
---

1 Larger window sizes tends to replace neighbouring nuclei with the same digital value, reducing the *crispness* of an image to a certain extent.

2 A *hybrid* smoothing algorithm is constructed when a smoother is preceded or followed by another filter/smoother.

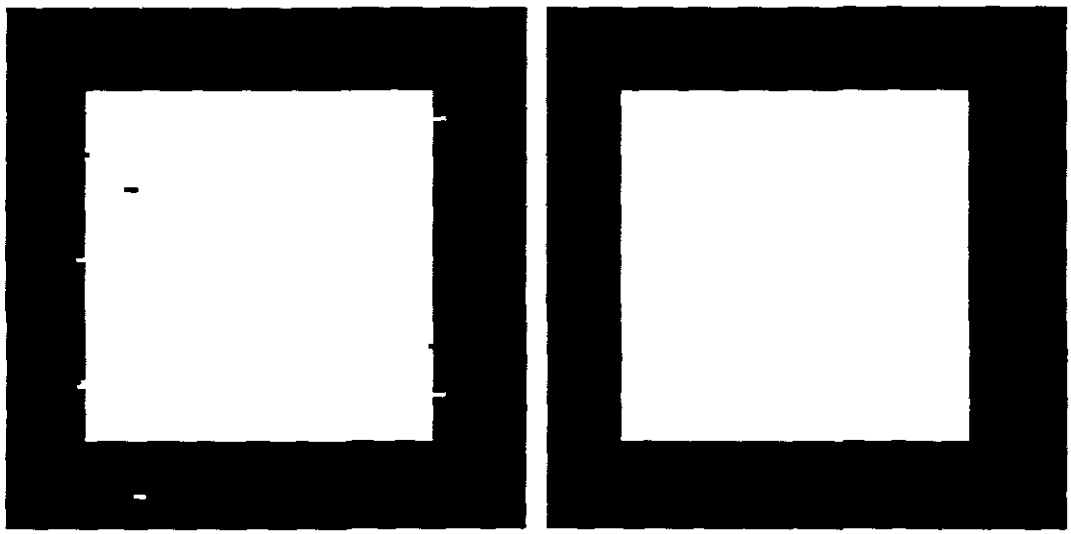
4.5.1 LULU_2D/HV

Paragraph 4.1 introduced the idea of sweeping the two-dimensional image with **LULU_1D** scan lines in horizontal and vertical directions. This idea will now be investigated further by



Screendump 4.5.1.1 Original image, **O** Resultant picture, $\mathbf{P} = \mathbf{P}_h$

applying **LULU_1D** sweeps to **O**. Using the convention as defined before, **LULU_2D/VH** defines an algorithm where the image is firstly sweeped with horizontal **LULU_1D/H** scan lines and the resultant picture, $\mathbf{P}_h$, is then sweeped vertically with the similar **LULU_1D/V** algorithm to yield the final picture, $\mathbf{P} = \mathbf{P}_{vh}$. Screendump 4.5.1.1 shows the original image, as used before, on the left, while the right-hand side shows the resultant picture, $\mathbf{P}_h$, the image after **LULU_1D/H** was applied. It is clear to see that the previous problem concerning the edges of the image is solved with this method. A final vertical sweep concludes the process and the original image is restored, as shown in screendump 4.5.1.2.



Screendump 4.5.1.2 Intermediate $\mathbf{P}_h$ Final smoothed picture, $\mathbf{P} = \mathbf{P}_{vh}$

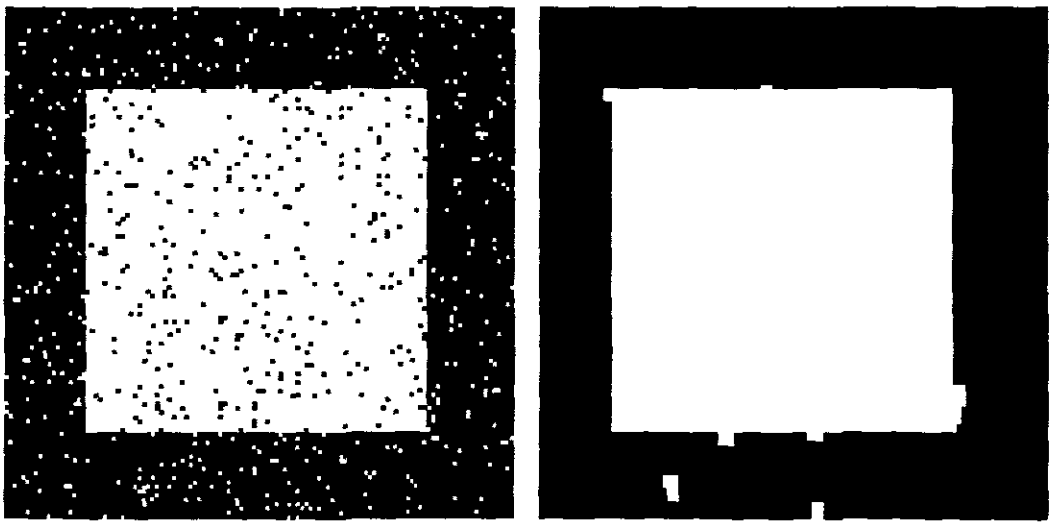
Note that this smoother seems to be considerably more powerful than **LULU_2D/5W** because the same image, (figure 4.5.6), was completely restored in this case. The power of this smoother comes at a cost, because it is numerically more complex. Not only does the algorithm involve more floating point operations, but it is actually a hybrid because it involves an average calculation between the upper and lower bounds. Comparisons between the different algorithms are left for discussion and analysis in chapter five.

4.5.2 LULU_2D/VH

It can be verified that a similar result as in §4.5.1 would have been attained if the **LULU** sweeps were switched around. (See screendump 4.5.1.2(a) in the appendices). From the results shown in this chapter is seems as if **LULU** sweeps can be arbitrarily switched around[1] and still give a similar end result. This is not necessarily true and the opposite can easily be shown by using a different example. Screendumps 4.5.2.1 and 4.5.2.2 illustrates that **LULU_2D/VH** does not yield the same results as **LULU_2D/HV** for the same original image, **O**, which is oversaturated with random noise.

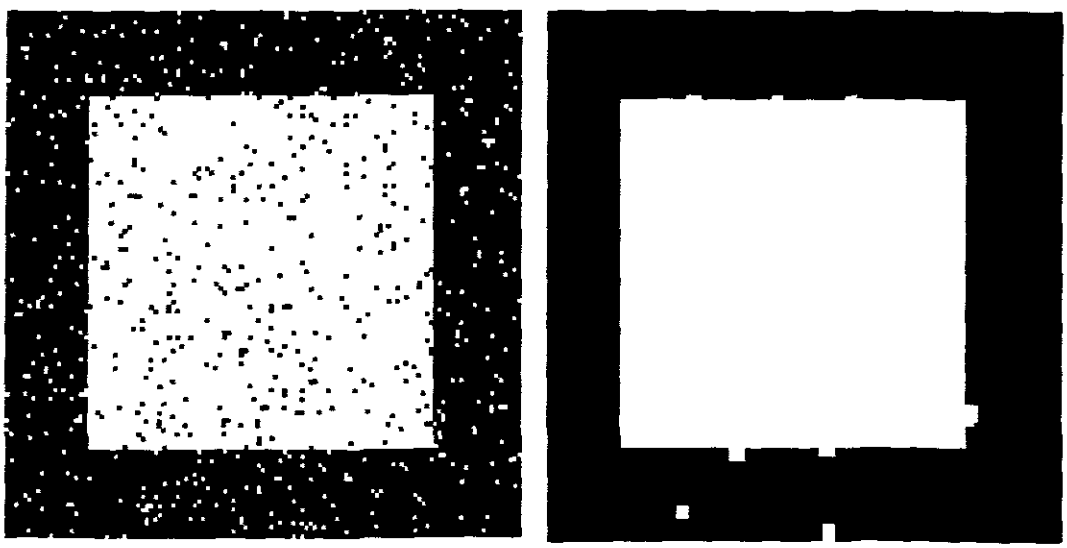
---

[1] Also for **F** and **C**.



Screendump 4.5.2.1 Original image, **O** Resultant picture, $\mathbf{P} = \mathbf{P}_{vh}$

The mathematical and practical implications of this phenomenon is explained in detail in chapter five.



Screendump 4.5.2.2 Original image, **O** Resultant picture, $\mathbf{P} = \mathbf{P}_{hv}$

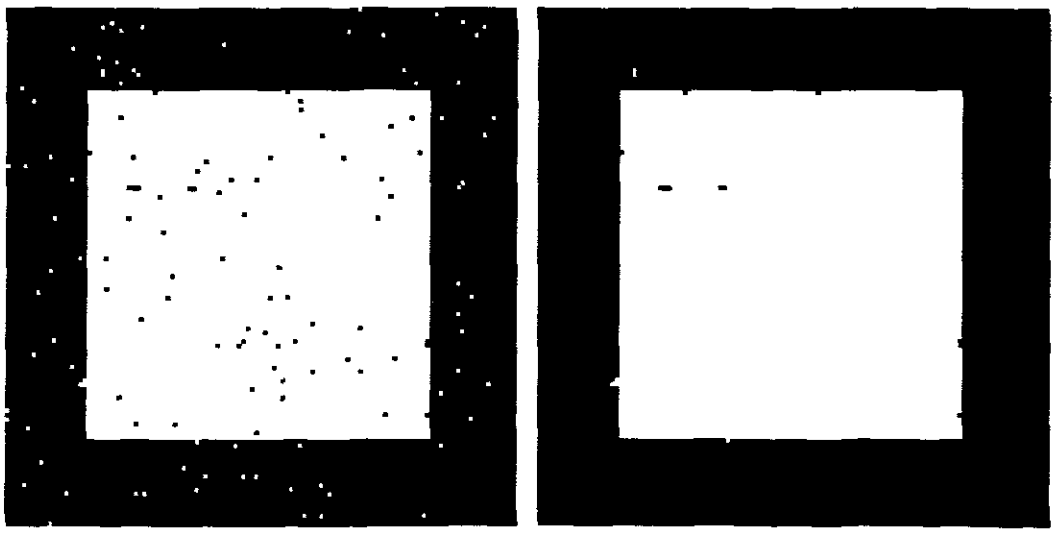
The different two-dimensional methods, as developed in this chapter, can now be summarised in notational format.

4.5.3 2D LULU notation

- **LULU_2D/5W** An image, **O**, is converted to a final picture, **P**, after five-point *ceiling* and *floor* sweeps have been performed on it. We can refer to the resultant picture as $\mathbf{P} = \mathbf{P}_{L5}$.
- **LULU_2D/9W** An image, **O**, is converted to a final picture, **P**, after nine-point **LULU** *ceiling* and *floor* sweeps have been performed on it. We can refer to the resultant picture as $\mathbf{P} = \mathbf{P}_{L9}$.
- **MEDIAN_2D/5W** An image, **O**, is converted to a final picture, **P**, after five-point median sweeps have been performed on it. We can refer to the resultant picture as $\mathbf{P} = \mathbf{P}_{M5}$.
- **MEDIAN_2D/9W** An image, **O**, is converted to a final picture, **P**, after nine-point median sweeps have been performed on it. We can refer to the resultant picture as $\mathbf{P} = \mathbf{P}_{M9}$.
- **LULU_2D/HV** An image, **O**, is converted to a final picture, **P**, after one-dimensional **LULU** sweeps have been performed on it in the order, ***HV***. We can refer to the resultant picture as $\mathbf{P} = \mathbf{P}_{hv}$.
- **LULU_2D/VH** An image, **O**, is converted to a final picture, **P**, after one-dimensional **LULU** sweeps have been performed on it in the order, ***VH***. We can refer to the resultant picture as $\mathbf{P} = \mathbf{P}_{vh}$.
- **LULU_2D/V** This algorithm refers to **LULU_1D** sweeps, applied only vertically on a two-dimensional image. We can refer to the resultant picture as $\mathbf{P} = \mathbf{P}_{v}$.
- **LULU_2D/H** This algorithm refers to **LULU_1D** sweeps, applied only horizontally on a two-dimensional image. We can refer to the resultant picture as $\mathbf{P} = \mathbf{P}_{h}$.

4.5.4 Hybrid algorithms

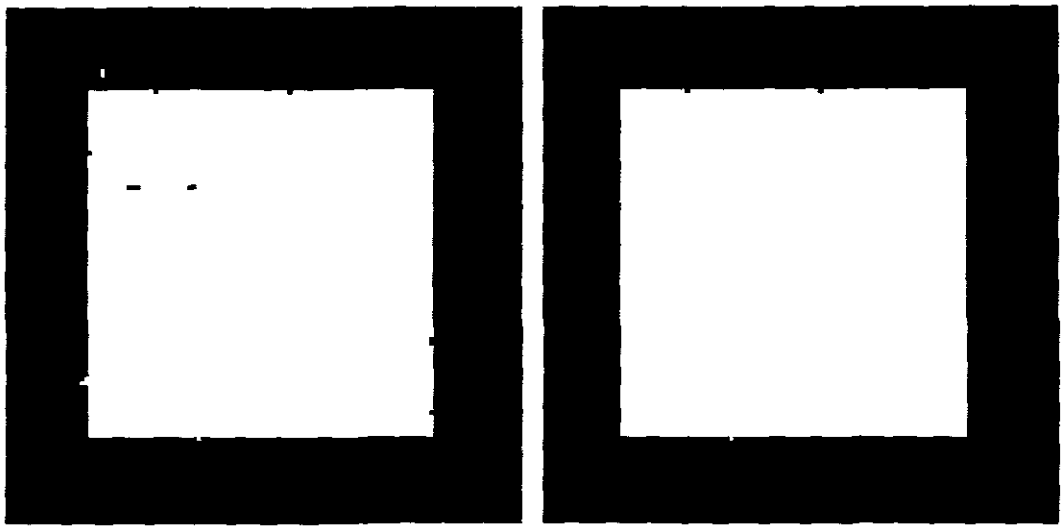
It is possible to combine algorithms such as those mentioned in this chapter with linear or non-linear methods to form *hybrid* methods.



Screendump 4.5.4.1 **O** $\mathbf{P}_{L5}$

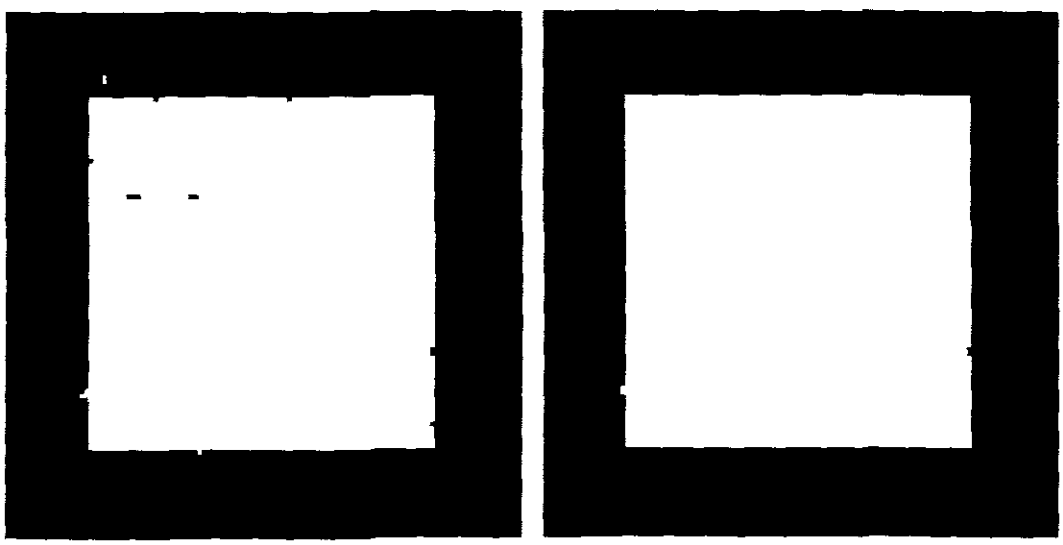
As mentioned already, the **LULU_2D/HV** and **LULU_2D/VH** algorithms are implicit examples of hybrid algorithms. Numerous other hybrids can be formed by using smoothers, filters and other enhancement techniques in combination. **LULU** smoothers are designed primarily to serve as pre-filters and therefore most of the investigation in this project did not go beyond the testing and verification up to the stage where post-filtering should start.

If the example in screendump 4.5.4.1 is smoothed using **LULU_2D/V**, the right-hand side of screendump 4.5.4.2 shows the result, $\mathbf{P}_{vL5}$.



Screendump 4.5.4.2 P_{L5} P_{vL5}

It is clear that if **LULU_1D** is now applied horizontally, the image will be completely corrected. In this example it is immaterial in which order the **LULU_1D** algorithm was applied, because the remaining noise intensity is low. This application clearly indicates the superiority of **LULU** algorithms to the median, especially in steep border regions. Screendump 4.5.4.3 shows that a five-point median algorithm operating on the five-point **LULU** output could not clean the image completely of noise.



Screendump 4.5.4.3 P_{L5} P_{M5L5}

In the search for optimal correction of a noisy image many options are available to measure images for quality. Numerous image processing software packages for enhancement are commercially available and a great number of algorithms exist in the literature. If the quality of a single picture is to be improved, an operator will usually attempt to clean noise first by some pre-filter or -smoother of his choice and then try to enhance the image with further processes. Many tools to compare the quality of the input image to that of the final picture exist, but it is normally the human eye that makes a conclusive decision due to the subjectivity of visual display.

When streams of images have to be improved, normally at real-time, human intervention is out of the question and automatic procedures have to be devised [5,24,35]. Depending on available *a priori* information the system will usually determine whether pre-filtering with some smoother is necessary. If the noise type is not known, or is of alternating nature, the smoothing process can become extremely complicated. The speed at which these enhancements must occur is also a limiting factor. Smoothers and filters for digital sound application need to operate in the order of 11kB/s. A 1024X768 image in 24-bit colour, used in digital video at 25 frames per second, requires 2,4Mb of data to be processed every second [8].

The fact that digital video display is barely possible on current hardware and the quality is not yet comparable to that of the conventional analogue methods, is an indication that new DIP methods must be developed, most probably for more advanced hardware. Some of these methods for parallel structures are discussed in §6.7.2.

CHAPTER FIVE

Mathematical verification of 2D LULU smoothers.

5.1 Mathematics for image processing and graphics

Computer graphics makes use of a wide range of mathematical concepts to achieve the aims of various programming tasks. For an image processing assignment, such as the objective of this project, the first task is to set a reference frame for pixel identification. Although the image processing for **LULU** is mapped against a normal Cartesian co-ordinate system, other systems, such as spherical, cylindrical or polar co-ordinates can be useful for specific programming environments. A solid knowledge of analytic geometry, linear algebra, vector analysis, tensor analysis, complex numbers and numerous other areas from numerical analysis is advantageous [32,83] for image processing students.

The well-known Fourier transform and variants thereof provide a spectral decomposition of an image into components that isolate and enhance image features. These methods are extremely useful for linear image processing applications and are excellent tools for noise reduction when low-frequency noise is present. If $\Im(u,v)$ is the discrete two-dimensional transform of a sampled image f(j,k), the transform pairs can be presented as [76,83]:

5.1.1 If $$\Im(u,v) = \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} f(j,k)\, A(j,k;u,v)$$

5.1.2 and $$f(j,k) = \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} \Im(u,v)\, B(j,k;u,v)$$

where A(j,k;u,v) is the forward transform kernel and B(j,k;u,v) is the backward kernel.

For non-linear applications, like the **LULU** smoothing theory, results are attained through discrete matrix transforms using a sweeping window as the device to effect convolution. Naturally other effects can be implemented by elementary matrix operations, such as matrix addition, subtraction or multiplication [41, 50].

5.2 Mathematical observations considering LULU 1D formulae

The concept of grouping local maximum and minimum operators into *shrinking* or *expanding* routines for thresholding in binary images is not new. Nakagawa and Rosenfield published a paper in 1978 reporting on the noise suppression capabilities of such max-min operators for two-dimensional application [69]. A new approach developed by analysing max-min operators for one-dimensional theory and thereafter applying this knowledge to design two-dimensional algorithms. Dr. C.H. Rohwer from the Department of Mathematics, Stellenbosch University, has been the major force behind the development and analysis of **LULU** structures in general. This chapter is a short overview of some of his work. The serious mathematically minded reader is advised to refer to the literature for more detailed information. It must also be pointed out that non-linear smoothers are deceivingly intricate in nature and not all crucial proofs are necessarily proved by algebraic means. The purpose of this chapter is therefore more explanatory and indicates the logical basis of the physical phenomena that has been recorded in chapters three and four.

The one-dimensional **LULU** smoothing concept was described in chapter 3, while the application of the **LULU_1D** algorithm in two dimensions was introduced in §4.4. Although the **LULU_HV** and **LULU_VH** algorithms proved to be quite powerful and useful for certain noise patterns, it lacked idempotency and are therefore not formally investigated. Another option for the formulation of the one-dimensional operators, is:

5.2.1 $(Lx)_i = \max(\min(x_{i-1}, x_i), \min(x_i, x_{i+1}))$,

5.2.2 $(Ux)_i = \min(\max(x_{i-1}, x_i), \max(x_i, x_{i+1}))$,

for real sequences.

5.2.1 and 5.2.2 provided the idea to construct two-dimensional **LULU** operators similarly.

In order to fully understand the two-dimensional **LULU** structure, it might be worthwhile to give some further consideration to the construction of a multiplication table for the one-dimensional semi-group [79]:

5.2.3 $\mathbf{L} \leq \mathbf{I} \leq \mathbf{U}$, where **I** is the identity operator,

5.2.4 $\mathbf{LL} = \mathbf{L}^2 = \mathbf{L}$ and similarly $\mathbf{UU} = \mathbf{U}$, (idempotency of the basic operators),

5.2.5 $\mathbf{LUL} = \mathbf{UL} \leq \mathbf{LU} = \mathbf{ULU}$.

	I	L	U	LU	UL
I	I	L	U	LU	UL
L	L	L	LU	LU	UL
U	U	UL	U	LU	UL
LU	LU	UL	LU	LU	UL
UL	UL	UL	LU	LU	UL

Table 5.2.6

From 5.2.3 - 5.2.5, the multiplication table 5.2.6 can be constructed. Idempotance of **UL** and **LU** can now be proven from 5.2.5 using associativity, idempotence and syntoneness of **L** and **U** [79,80,81].

Note that it is possible to extend this table for combinations like **LUL** and **ULU**, but is this not of significance for the comparison between one-and two-dimensional smoothers at this stage. The **UL** and **LU** combinations are sufficient for the construction of efficient one-dimensional smoothing algorithms and the intention is to try to prove a similar structure for the two-dimensional domain.

5.3 Local selectors for a five-point support window in two dimensions

If we assume that sample data is stored in a two-dimensional grid of equidistant intervals where each pixel is reflected as a sampled measurement x_{ij}, the image can be defined as: $\mathbf{x} = \{ x_{i,j}: (i,j) \in D = [1,N] \subset [1,M] \}$, where D^1 denotes the set of double indexed integers.

It is acceptable to initially consider the most straight-forward two-dimensional five-point sweeping window (as defined in §4.1.1) and formulate the accompanying two-dimensional **LULU** operators as:

5.3.1 $(\mathbf{F}_5 x)_{i,j} = \max(\min(x_{i,j}, x_{i,j-1}), \min(x_{i,j}, x_{i,j+1}), \min(x_{i,j}, x_{i-1,j}), \min(x_{i,j}, x_{i+1,j}))$

5.3.2 $(\mathbf{C}_5 x)_{i,j} = \min(\max(x_{i,j}, x_{i,j-1}), \max(x_{i,j}, x_{i,j+1}), \max(x_{i,j}, x_{i-1,j}), \max(x_{i,j}, x_{i+1,j}))$

Let $\mathbf{F} = \mathbf{F}_5$ indicate a five-point *floor* operator acting on the entire picture data and $\mathbf{C} = \mathbf{C}_5$ the corresponding five-point *ceiling* operator for the rest of this section, unless stated differently.

It can easily be verified that **F** removes a single upward impulse and that **C** removes a downward impulse due to the comparisons between the nucleus, $x_{i,j}$, and its four nearest neighbours. The following theorem can thus be derived [80]:

Theorem 5.3.3 $\mathbf{F} \leq \mathbf{I} \leq \mathbf{C}$, where I is the identity operator.

Proof: Removing an element from a set cannot increase the maximum value of the set. Therefore,

$$(Cx)_{i,j} = \min(\max(x_{i,j}, x_{i,j-1}), \max(x_{i,j}, x_{i,j+1}), \max(x_{i,j}, x_{i-1,j}), \max(x_{i,j}, x_{i+1,j}))$$
$$\geq \min(\max(x_{i,j}), \max(x_{i,j}), \max(x_{i,j}), \max(x_{i,j})) = x_{i,j}$$

similarly,

---

[1] Note that vector and array data are presented as small letters in bold, while matrices are capitalised and not bold. When a matrix is used to represent an image it will be shown in bold, the same as for operators and selectors.

$$(Fx)_{i,j} = \max(\min(x_{i,j}, x_{i,j-1}), \min(x_{i,j}, x_{i,j+1}), \min(x_{i,j}, x_{i-1,j}), \min(x_{i,j}, x_{i+1,j})),$$
$$\leq \max(\min(x_{i,j}), \min(x_{i,j}), \min(x_{i,j}), \min(x_{i,j})) = x_{i,j}, \qquad \forall\ x_{i,j}$$

This basic, but very important identity can be checked visually by computing examples of the following nature:

(a) Binary example of a single pop on the null matrix:

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 0 0.	0. 0 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. 0. 0.	0. 0. 0. 0. 0.
Original matrix	After ceiling	After floor
O	**C**(O)	**F**(O)

Note that *'dummy'* zero's (0.) are used to pad the edges of the 3X3 original sub-matrix, (O).

As predicted, the pop is removed by the floor algorithm, while the ceiling algorithm leaves O unaltered.

(b) A 'downwards' pop:

0. 0. 0. 0. 0.	0. 0. 0. 0. 0.	0. 0. 0. 0. 0.
0. 1 1 1 0.	0. 1 1 1 0.	0. 1 1 1 0.
0. 1 0 1 0.	0. 1 1 1 0.	0. 1 0 1 0.
0. 1 1 1 0.	0. 1 1 1 0.	0. 1 1 1 0.
0. 0. 0. 0. 0.	0. 0. 0. 0. 0.	0. 0. 0. 0. 0.
Original matrix	After ceiling	After floor
O	**C**(O)	**F**(O)

If another binary example is constructed where a pop is in a downward direction, the ceiling algorithm removes the downward pop, while the floor algorithm leaves O unaltered and the relation 5.3.3 is verified. The behaviour of these operators is also clearly illustrated in screendumps 4.2.6 , 4.2.7 and 4.2.10.

The next important question to answer, is what happens when a two-dimensional **LULU** operator acts on itself. This question was intuitively answered by checking pixel counts from test runs as shown in chapter four. In order to prove idempotency of a complete smoother, we have to consider syntoneness [80] of an operator, Γ

Definition 5.3.4 An operator Γ is syntone if $\mathbf{x} \geq \mathbf{w}$ implies that $\Gamma\mathbf{x} \geq \Gamma\mathbf{w}$.

It can be verified if Γ A and B are all syntone and $\Gamma \geq A$,
the relations $\Gamma B \geq \Gamma B$ and $B\Gamma \geq \Gamma A$ are true.

The following theorem can thus be proven:

Theorem 5.3.5 $\mathbf{F^2 = F}$ and $\mathbf{C^2 = C}$ (Idempotency)

Proof: $\mathbf{C \geq I}$, therefore $\mathbf{C^2 \geq C}$ due to syntoneness.

If $v_{i,j} = C(x_{i,j})$, assume $C^2(x_{i,j}) > C(x_{i,j})$and thus

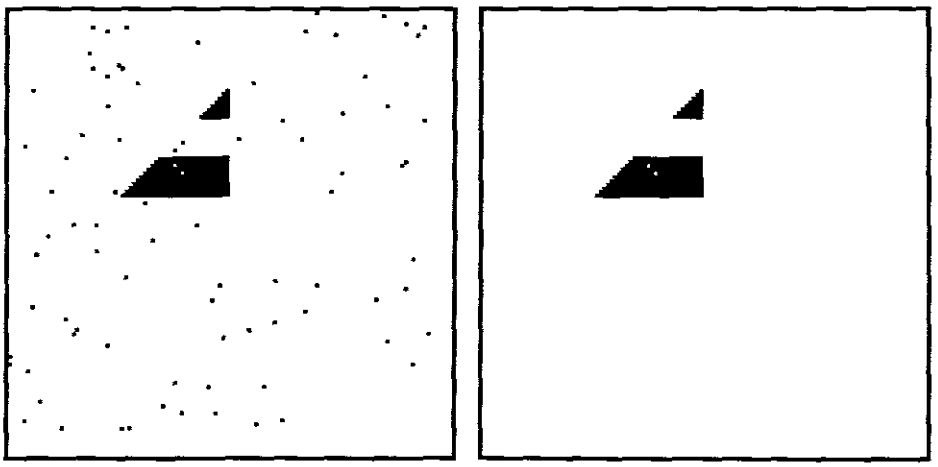
$\max(v_{i,j}, v_{i,j+1}) > v_{i,j}$, $\max(v_{i,j}, v_{i,j-1}) > v_{i,j}$,
$\max(v_{i,j}, v_{i-1,j}) > v_{i,j}$, $\max(v_{i,j}, v_{i+1,j}) > v_{i,j}$.

If this is true, each set $S = \{ v_{i,j+1}, v_{i,j-1}, v_{i+1,j}, v_{i-1,j} \}$ has a maximum larger than $v_{i,j}$.

Take $v_{i-1,j}$ as an example. Since $v_{i,j} = C(x_{i,j})$, either $x_{i-1,j} > v_{i,j}$ or each of $\{ x_{i,j}, x_{i-2,j}, x_{i-1,j-1}, x_{i-1,j+1} \}$ is larger than $v_{i,j}$. If this argument is repeated for each set S, then either or all of $\{ x_{i,j+1}, x_{i,j-1}, x_{i+1,j}, x_{i-1,j} \}$ is larger than $v_{i,j}$. This leads to a contradiction, because $C(x_{i,j}) > v_{i,j}$.
Therefore $\mathbf{C^2 = C}$ and $\mathbf{F^2 = F}$ can be proved similarly.

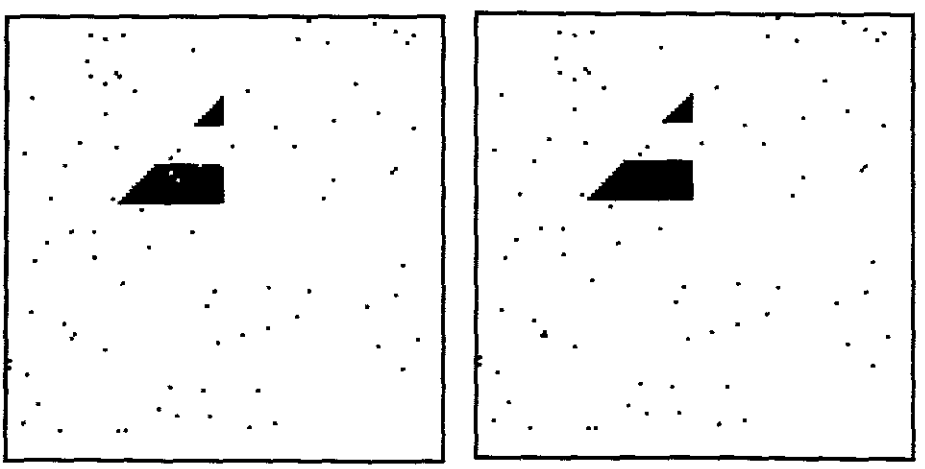
This result proves that idempotency holds for the basic two-dimensional smoothing operators and that the theory has the desired correspondence with one-dimensional

LULU theory. The practical implication of idempotency of the basic five-point operators can easily be illustrated with a binary example (120X120) in screendump 5.3.5.



Screendump 5.3.6 **O** After a floor sweep : **F(O)**

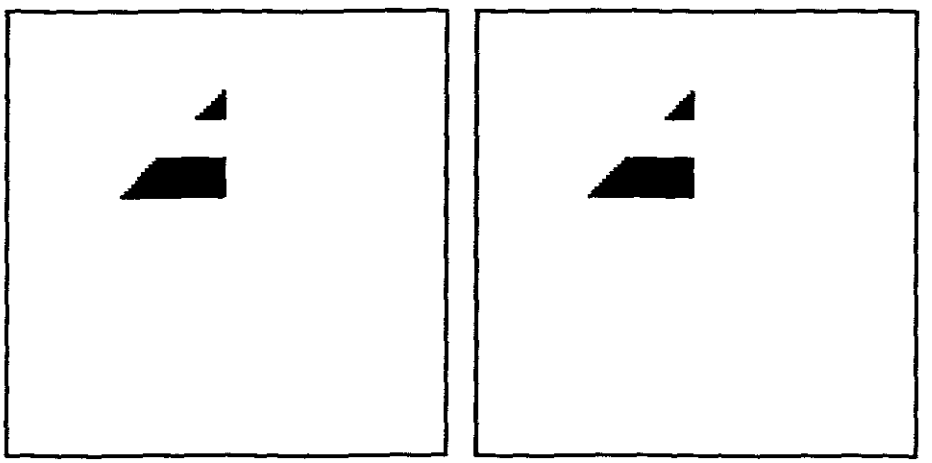
The floor sweep, **F(O)**, removes upward noise in the original image, **O**, as predicted in the theory and leaves the rest of the image unaltered. $\mathbf{FF(O) = F^2(O) = F(O)}$, and therefore no further floor sweep would affect the resultant picture, **P = F(O)**.



Screendump 5.3.7 **O** After a ceiling sweep : **C(O)**

The ceiling sweep removes the noise pixels[1] in upper geometric structure and leaves the rest of the original image intact. Note that the triangular section, which had no noise in **O**, was unaffected by **C**, thus preserving original features. Repeated application of **C** will again have no further effect on the resultant picture as proved in 5.3.5.

In chapter four the usefulness of a ceiling sweep, followed by a floor sweep is illustrated as a *complete algorithm*[2] to remove ceiling and floor noise in a binary image. Screendump 5.3.8 show the results of the complete smoothers.



Screendump 5.3.8 $\mathbf{P}_{cf}$ $\mathbf{P}_{fc}= \mathbf{F(C(O))}$

It is interesting to note that in this particular example, the original matrix is not oversaturated with random noise and both $\mathbf{P}_{cf}$ and $\mathbf{P}_{fc}$ are effectively cleaned of impulsive noise. Although the image is successfully smoothed by the smoothers **FC** and **CF** there is no guarantee that **FC** will be the similar as **CF** when the image noise is not in acceptable bounds (also see §3.1). It seems, at this stage anyhow, that the smoother pairs in screendump 5.3.8, are successful in removing impulsive noise from a binary picture. The question should now be asked: does idempotency features of these composite pairs exist? It seems a reasonable question to ask, from the knowledge gained in the one-dimensional

---

1 'Down' pops.

2 A *complete smoother* refers to a **LULU** algorithm which does not need any follow-up operation to remove noise.

case and because we have already shown that there will be no change in the resultant picture if any of the smoothers, **F** or **C**, are used repetitively.

Theorem 5.3.9 **FC and CF are idempotent**

Proof: $(FC)^2 = FCFC \geq FFFC = FC$, (from $C \geq F$), and similarly,

$(FC)^2 = FCFC \leq FCCC = FC$,

Thus $(FC)^2 = FC$ and idempotence is proved.

It similarly follows that CF is also idempotent.

From the multiplication table, 5.3.13, the following interesting extensions of combined operations can also be proved:

The effectiveness of these combined five-point smoothers for high-resolution grey-scale smoothing are illustrated in screendumps 4.2.11 - 4.2.14.

Definition 5.3.10

A matrix is 1-monotone if it is 1-monotone in both indexes at (i,j) [80], iff. the sequences $x^j = \{ x_I^j; x_I^j = z_{i,j}, I \in \mathbf{Z} \}$.

Definition 5.3.11

The matrix with data $z = [z_{i,j}]$ is called weakly 1-monotone at (i,j) if the set $A = \{ z_{i,j+1}, z_{i,j}, z_{i,j-1} \}$ is monotone in the index j; or $B = \{ z_{i-1,j}, z_{i,j}, z_{i+1,j} \}$ is monotone in the index I; or the sets A and B are both not convex; or both not concave in the indexes j and i respectively. A matrix is 1-monotone if it is 1-monotone everywhere.

Theorem 5.3.12 **FCF ≤ CFC.**

Proof: $(FCF)^2 = FCFFCF \leq ICFFCI = CFC$, (from $I \geq F$).

The relation between **CF** and **FC** can only be proven after a thorough investigation of the monoticity of the combined structures. Rohwer [80], indicates that **FCF(x)** is weakly 1-monotone, so that **C** maps on itself and **FC** = **CFC** is proved.

Theorem 5.3.13 **CF ≤ FC.**

Proof: $CF = FCF \leq CFC = FC$.

Theorem 5.3.14 **FCF** and **CFC** are idempotent

Proof: $(FCF)^2 = FCFFCF \geq FFFFCF = FCF$, (from $C \geq F$), and similarly,
$(FCF)^2 = FCFFCF \leq FCCCCF = FCF$, therefore
$(FCF)^2 = FCF$ and idempotency is once again proved.

	F	C	FC	CF
F	F	FC	FC	CF
C	CF	C	FC	CF
FC	CF	FC	FC	CF
CF	CF	FC	FC	CF

Table 5.3.15

This last proof completes the missing link in order to compose a similar multiplication table, table 5.3.15, for a semi-group of the five-point **C** and **F** operators.

Note that the *identity operator*[1] is not included as was the case with the one-dimensional table. The reason for this is that **I** is not comparable with the smoothers **FC** or **CF**.

5.4 Local selectors for a nine-point support window in two dimensions

It has been pointed out in the previous chapters of this thesis that numerous other **LULU** operators, other than the five-point structure can be formulated. **LULU_2D/HV** and **LULU_2D/9W_1**[2] are typical examples of smoothers which are presented and tested in practice, but due to their *weaker*[3] structure are not formally examined mathematically.

Rohwer [80] further defines a class of **LULU** operators, based on the five-point window:

5.4.1 $$(F^*x)_{i,j} = \max(\min(x_{i,j}, x_{i,j+1}, x_{i+1,j}), \min(x_{i,j}, x_{i,j+1}, x_{i-1,j}), \min(x_{i,j}, x_{i,j-1}, x_{i-1,j}), \min(x_{i,j}, x_{i,j-1}, x_{i+1,j}))$$

5.4.2 $$(C^*x)_{i,j} = \min(\max(x_{i,j}, x_{i,j+1}, x_{i+1,j}), \max(x_{i,j}, x_{i,j+1}, x_{i-1,j}), \max(x_{i,j}, x_{i,j-1}, x_{i-1,j}), \max(x_{i,j}, x_{i,j-1}, x_{i+1,j}))$$

Intuitively one senses that this structure fits in somewhere between the five-point structure of §5.4 and nine-point **LULU** structure as defined in §4.3. This fact is illustrated by the next theorem.

---

1 The identity operator **I** can be defined as the operator which will leave a picture unaltered if applied: $\mathbf{P} = \mathbf{I}(\mathbf{P})$

2 Refer to §4.4.1, §4.4.2 and §4.4.3.

3 FCF ≠ CF. Refer to [80] for further information on the mathematical relation between operators (*strength* and *weakness*).

Theorem 5.4.3 $\mathbf{F^* \le F \le C \le C^*}$

Proof: If one element is removed from each of the four element sets of $\mathbf{F^*}$, we get the definition of $\mathbf{F}_5$. The minima of $\mathbf{F^*}$ could therefore not have increased and subsequently the maxima of these minimums could also not have increased, thus:

$\mathbf{F^* \le F_5}$. A similar argument holds for $\mathbf{C^* \ge C_5}$.

An even *tighter*[1] **LULU** structure can now be assembled from a nine-point sweeping window:

5.4.4 $$(F_9x)_{i,j} = \max(\min(x_{i,j}, x_{i,j+1}, x_{i-1,j+1}, x_{i-1,j}), \min(x_{i,j}, x_{i,j-1}, x_{i-1,j-1}, x_{i,j-1}),$$
$$\min(x_{i,j}, x_{i,j-1}, x_{i+1,j-1}, x_{i+1,j}), \min(x_{i,j}, x_{i,j+1}, x_{i+1,j+1}, x_{i+1,j})$$

5.4.5 $$(C_9x)_{i,j} = \min(\max(x_{i,j}, x_{i,j+1}, x_{i-1,j+1}, x_{i-1,j}), \max(x_{i,j}, x_{i,j-1}, x_{i-1,j-1}, x_{i,j-1}),$$
$$\max(x_{i,j}, x_{i,j-1}, x_{i+1,j-1}, x_{i+1,j}), \max(x_{i,j}, x_{i,j+1}, x_{i+1,j+1}, x_{i+1,j})$$

The following theorems regarding the **LULU_2D/9W** can be derived similarly to the **LULU_2D/5W** identities, as proven in §5.4:

Theorem 5.4.6

(a) $\mathbf{F_9 \le I \le C_9}$

(b) $\mathbf{(F_9)^2 = F_9}$ and $\mathbf{(C_9)^2 = C_9}$

(c) $\mathbf{F_9C_9F_9 \le C_9F_9C_9}$

(d) $\mathbf{F_9C_9}$ and $\mathbf{C_9F_9}$ are idempotent.

(e) $\mathbf{F_9C_9F_9}$ and $\mathbf{C_9F_9C_9}$ are idempotent.

Due to syntoneness and idempotence it follows that $\mathbf{F_9 \le F_9C_9 \le C_9}$ and $\mathbf{F_9 \le C_9F_9 \le C_9}$. The nine-point structure is still an *incomplete structure* because the relation between $\mathbf{F_9C_9}$ and $\mathbf{C_9F_9}$ can not yet be proven formally and the multiplication table is thus incomplete.

---

[1] In the same sense as *stronger*. In practical terms it means more noise will be removed than with a *weaker* operator.

5.5 The relation of the median to LULU operators

It can be shown that any outshooter in the one-dimensional case which is removed by a median smoother, will also be removed by a similar **LULU_1D** smoother. To be more precise, $\mathbf{L} \leq \mathbf{M} \leq \mathbf{U}$, and the behaviour of the median is thus restricted to the upper and lower bounds of **LULU_1D** [79].

As before, the logical question arises about the comparison between the median operator in two-dimensional space, **M**, and the **LULU_2D** operators **F** and **C**. Due to the significant destructiveness of the nine-point median smoother, only the **MEDIAN_2D/5W** algorithm will be compared with $\mathbf{F}_9$ and $\mathbf{C}_9$.

Theorem 5.5.1 $\mathbf{F}_9 \leq \mathbf{M}_5 \leq \mathbf{C}_9$, where $\mathbf{M}_5 = \text{median}\{ x_{i,j}, x_{i,j-1}, x_{i,j+1}, x_{i-1,j}, x_{i+1,j})$

Proof: $(C_9x)_{i,j} = \min(\max(x_{i,j}, x_{i,j+1}, x_{i-1,j+1}, x_{i-1,j}), \max(x_{i,j}, x_{i,j-1}, x_{i-1,j-1}, x_{i,j-1}),$
$\max(x_{i,j}, x_{i,j-1}, x_{i+1,j-1}, x_{i+1,j}), \max(x_{i,j}, x_{i,j+1}, x_{i+1,j+1}, x_{i+1,j}))$
$\geq \text{median}(x_{i,j}, x_{i,j-1}, x_{i,j+1}, x_{i-1,j}, x_{i+1,j})$, since the median is the minimum of the maxima of all four-element subsets of the five-point window. A similar argument holds for the relation $\mathbf{F}_9 \leq \mathbf{M}_5$.

Although median smoothers have received a lot of attention in the literature, it is also known for some defects and inconsistencies. This is more noticeable for larger window sizes, such as the **MEDIAN_2D/9W** algorithm [21,78]. The practical implications of these deficiencies are illustrated with examples in chapter seven.

CHAPTER SIX

Programming considerations

6.1 Programming LULU algorithms in C and C++

At the start of this project it was decided to program the routines on easily accessible computers in a language, such as C or C++ [45, 51], to make the programs reasonably transportable to other architectures. With yearly improvements and obtainability of computer hardware the programs were tested on a range of desktop machines. The initial binary testing was done on a basic 386 PC and the image processing later on a Pentium computer. Obviously programming on 386 machines, especially in student laboratories, had some memory restrictions due to the 32 bit nature of the programs. 2Mb of RAM was necessary to run the programs, but loading and processing of large images needed more RAM. The program supports up to 32 Mb of memory. If a program runs out of memory, a temporary swap file will be created. This is used to simulate RAM, using virtual memory options.

The initial programs were written in ANSI C and later refined and written entirely in C++. The Watcom 32 bit C/C++ optimising compiler was chosen for the final programming, using the 32 bit DOS4G/W extender from Rational Systems. The WATCOM graphics library is used which supports VESA and most popular SVGA chipsets (ET4000, S3, TRIDENT, CIRRUS LOGIC, PARADISE and OAK). The graphics interface is handled by the graph object in GRAPH.C. This controls all graphics and interface routines; including windowing, list and file loading.

The programs support three 256 colour video modes 1024x768, 800x600 and 640x480. Obviously the highest resolution mode is recommended for image clarity [75].

Most of the algorithms require floating point calculation, thus a mathematical co-processer (FPU) is highly recommended. Without one the program does emulate floating point processing but speed will sacrificed.

For the best results on a PC, it is recommended to use at least a 16 Mb pentium PC with a 1024 Mb VGA graphics accelerator card [3].

6.2 The LULU primitives

As discussed in chapter five, the initial research of two-dimensional LULU structures required test routines for verification of mathematical conclusions. The most obvious method was to look at the smallest possible geometric two-dimensional clusters of binary pixels [36]. A sound smoother should leave the cluster unchanged when no noise is introduced. When spot noise is added to the picture, the smoother should attempt to remove the noise, but still try to conserve the original image.

6.2.1 Manual tests

An easy, but cumbersome, manual method is to apply this process with a pen and paper method. For instance, a 'thick' straight line (in binary format) with one noise pixel added in the image domain is shown in figure 6.2.1. If **LULU_2D/5W** is applied, with a window size of three, the results of the ceiling and floor processes can clearly be verified. The line structure is retained in both processes, $\mathbf{P}_c$ and $\mathbf{P}_f$, while the noise pixel is removed in only the floor process. As shown before, **CF** and **FC** will ensure the same

end result and from the theory in chapter five we know that **CF** ≤ **FC**. Naturally this method takes too much time for larger images, especially where random noise is present.

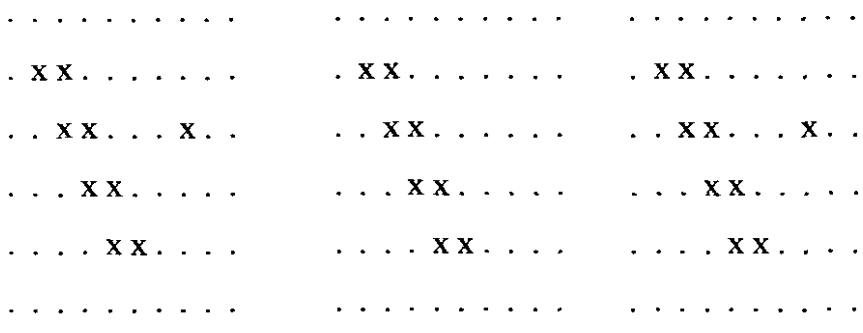


Figure 6.2.1 **O** $P_c = C(O)$ $P_f = F(O)$

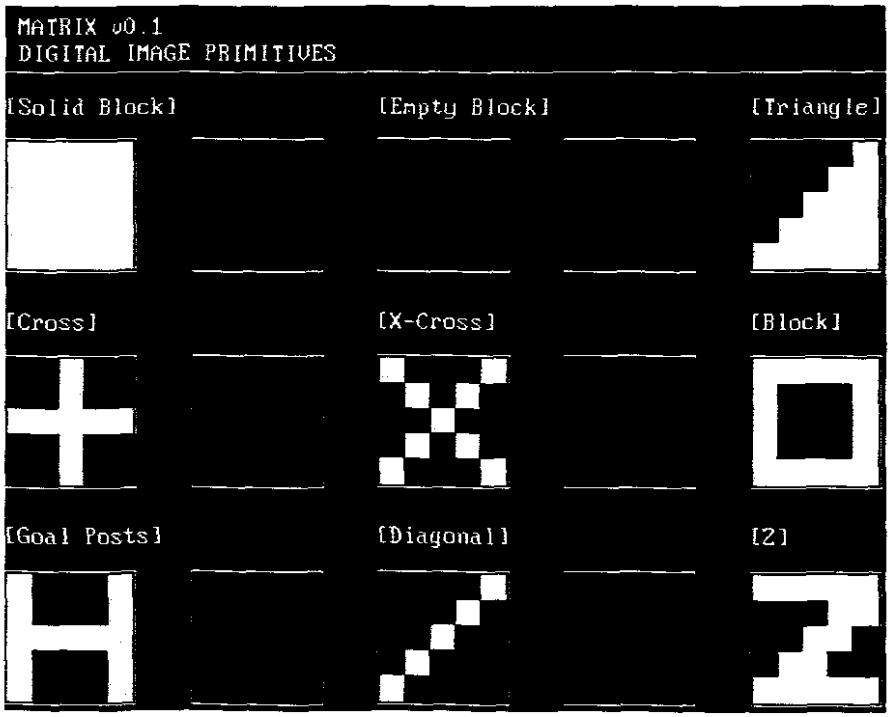
The next evolutionary step in the practical verification of **LULU** structures was to create programs which would illustrate the underlying **LULU** principles in a much faster and elegant fashion.

6.2.2 The MATRIX (Version 0.1) programs

The **MATRIX** set of programs (screendump 6.2.2) were developed, mostly as programming exercises for second year computer science students, to study small clusters of low-resolution pixels of general geometric shapes to test the behaviour of different **LULU** algorithms.

White low-resolution pixels are 'dropped and dragged' to form the 5X5 sub-matrices when a specific algorithm is activated. Note that for the sake of visibility the bordering matrix values are set to black, so that the behaviour of the algorithm is only tested on the image as shown. The resultant picture is then displayed in the empty block next to the original image.

The main objective of these programs is to verify that a **LULU** structure does what is predicted mathematically when it operates on specified geometric clusters of pixels. Idempotancy can also be checked visually.

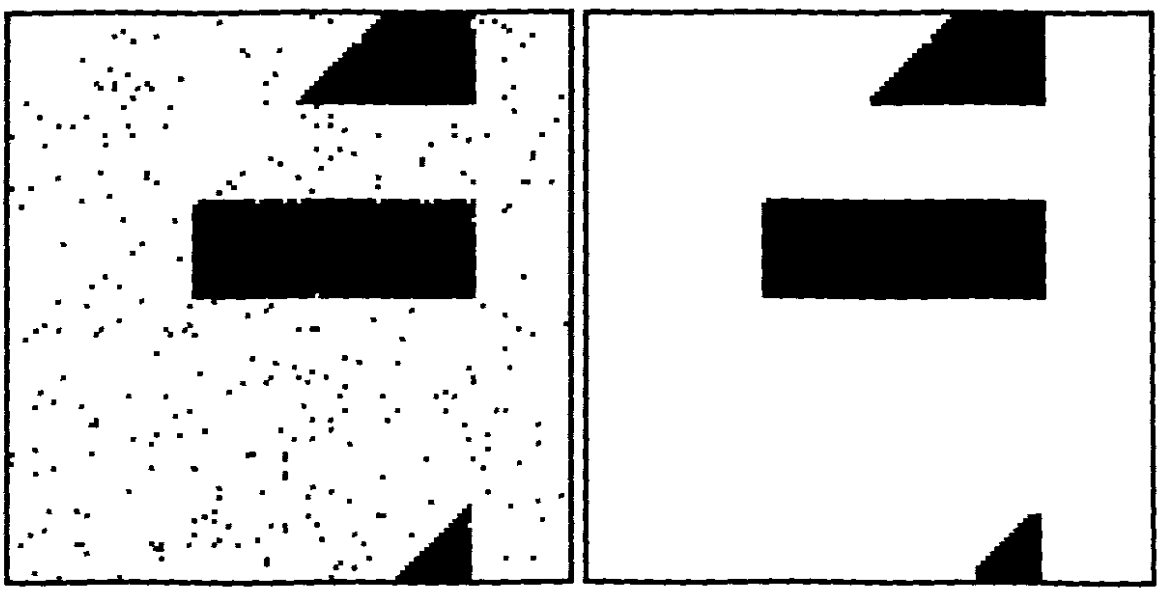


Screendump 6.2.2 **Screendump of the *MATRIX* set of programs**

Although these programs were quite useful in the initial stages of research, it was soon necessary to test algorithms on larger binary images.

6.2.3 Binary display images

The next set of programs were developed for studying 120X120 blocks of binary pixels in low-resolution. It is often difficult to detect the disappearance of a single pixel or a thin line of pixels in a high-resolution image. A typical example is the case where the tips of triangles disappears after the application of **LULU_2D/5W** (screendump 6.2.2). The **MATRIX v1.1** programs proved to be extremely useful in this regard and a large number of geometrical configurations were tested with the different smoothers and filters. Randomly seeded noise of various distributions were added to check the effectiveness of different algorithms. All the low-resolution screendumps of chapters three and four were also generated with these programs.



Screendump 6.2.2 **O** **Resultant picture, P_{hv}**

Another check to ensure that the output pixels corresponded to what was predicted, was by adding a count routine for pixels displayed. This provided a much more reliable and faster check for idempotancy, for instance, than visual checks.

The final proof of applicability of the **LULU** algorithms, was to test them in high-resolution images. Three sets of programs were developed for this purpose:

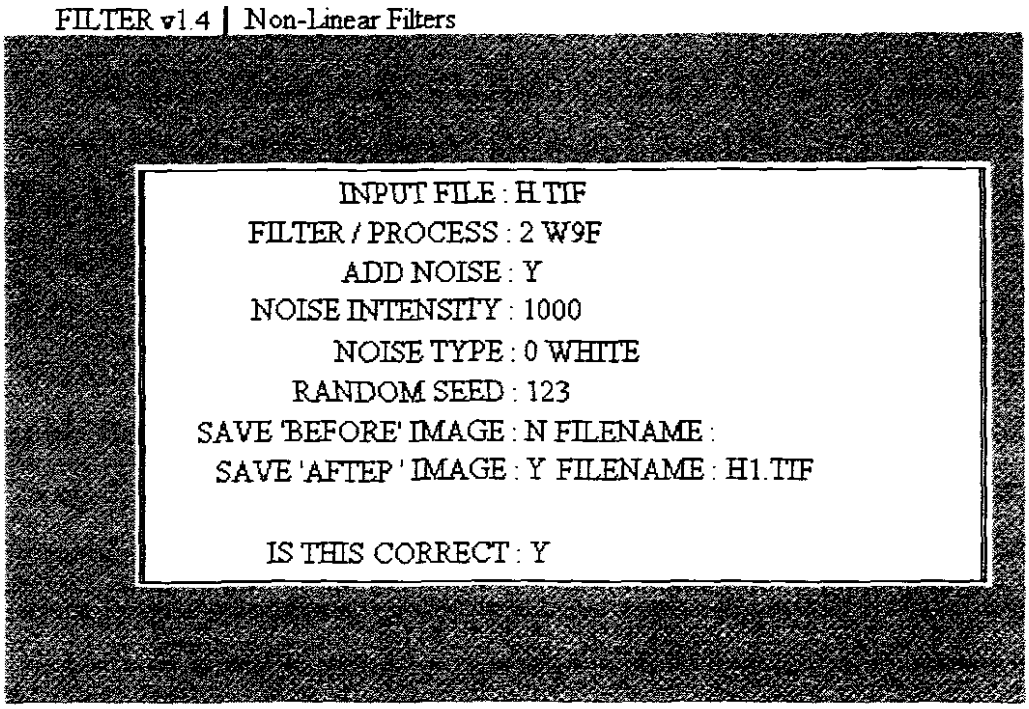
- **FILTER**
- **LULU**
- **ANALYSIS**

As most of the programming for this project went into the development of these prototype programs, important sections of the programs will be discussed fairly in detail in the sections to follow.

6.3 The FILTER program

The *FILTER* program contained the first grouped set of **LULU** algorithms for high-resolution images. These programs were designed to give visual insight into what happened when sub-programs (like **F** and **C**) were to operate independently on image data.

Screendump 6.3 shows a typical work page of this program.



Screendump 6.3 The title page of the *FILTER* program.

Although the program was written in a user-friendly manner, its capabilities will be described as follows:

6.3.1 The file is first specified. Note that the full file description (with path) is required.

6.3.2 The filter or smoother program is chosen:

0 and **1** select the **LULU_2D/5W** *floor* and *ceiling* algorithms;

2 and **3** select **LULU_2D/9W** *floor* and *ceiling* algorithms;

4 selects the **LULU_HV** algorithm;

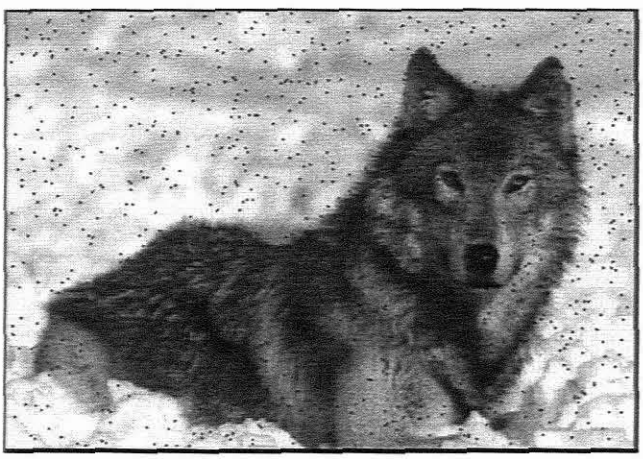
5 selects the **LULU_2VH** algorithm;

6 selects the **MEDIAN_2D/5W** smoother and

7 draws the **HISTOGRAM** of the input file.

6.3.3 *ADD NOISE* requires a response of *YES* or *NO*. If *NO* is selected the image can be processed without added *noise pixels*[1]. This is the case when an algorithm is tested on an image to see whether the original image changes when operated on. This is also the case when a *real*[2] image is tested. If only the histogram of an image is required, the *NO* option will also be selected.

6.3.4 If *YES* was selected in **6.3.3**, the intensity of the noise pixels can be regulated from 0-32767. It is necessary to have some control over the density of artificial noise distribution for several reasons. Firstly, if an already contaminated image is loaded, it would not be necessary to add further noise before filtering or smoothing is commenced. Another reason is more experimental of nature. The *strength* of a smoother can be compared against another one by increasing the noise intensity.



In the example shown in screendump 6.3.4, one thousand black noise pixels are scattered in a uniformly random fashion over the total image.

Screendump 6.3.4 Random noise.

6.3.5 *NOISE TYPE* sets the noise pixels to the following formats:

0	white noise pixels added to the image

1 Erratic pixel behaviour is artificially simulated by assigning certain (random) numeric values to valid image pixels.

2 This term is used for practical images, already contaminated with noise, that have to be smoothed.

1 black noise pixels added to the image

2 white and black noise pixels added to the image

3 randomly scattered pixels of all shades of grey added to the image (0-255)

6.3.6 *RANDOM SEED* controls the seed for the random number generator (0-32767). It may be noted that for a large number of tests it was necessary to keep the seed fixed as to compare what happens when different algorithms were applied to the same input image. The effectiveness of a specific filter can thus easily be monitored as shown in the throughput diagram, figure 6.3.6.

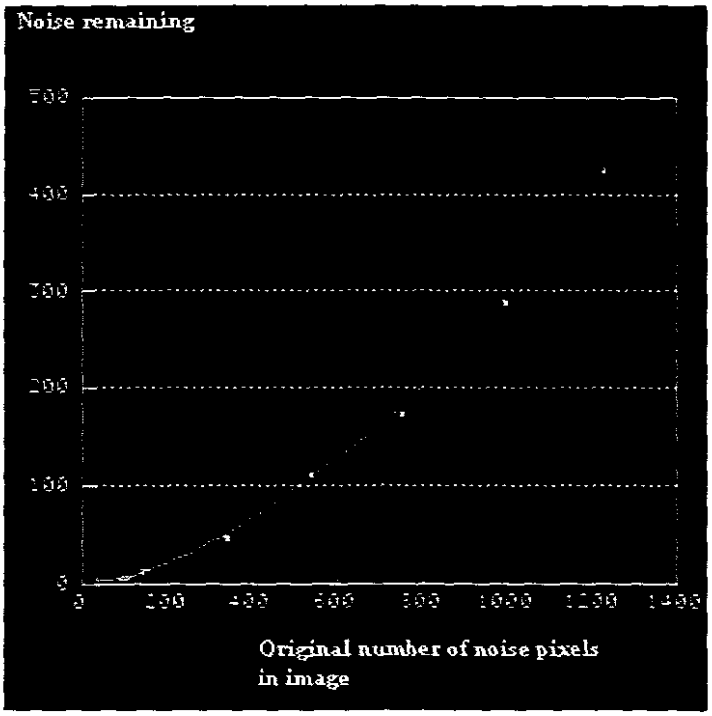


Figure 6.3.6 Number of pixels remaining after a five-point **LULU** sweep

The results illustrated in figure 6.3.6 were obtained when the noise was increased over the acceptable limits for a 120X120 low-resolution blank image. In this case a five-point **LULU** filter was employed. Although the algorithm operated on over-saturated noise levels, the resultant throughput curve remained smooth due

to the statistical behaviour of the noise remaining in the image. Similar results can be attained with all the **LULU_2D** smoothing algorithms for comparison purposes. The real proof of efficiency is however only visible when a real image with a specific noise distribution is tested. This was done with a large database of images and it was often evident that the theoretically *strongest*[1] smoother was not the always the 'best' algorithm for a specific application.

6.3.7 The next two options allow the saving of files before and after smoothing. If an image has to be smoothed successively, this option can be used. If **P1.TIF** is mentioned as the input file and also as the *AFTER* file, recursive smoothing can be implemented in batch format, because the letters in the title screen remains resident.

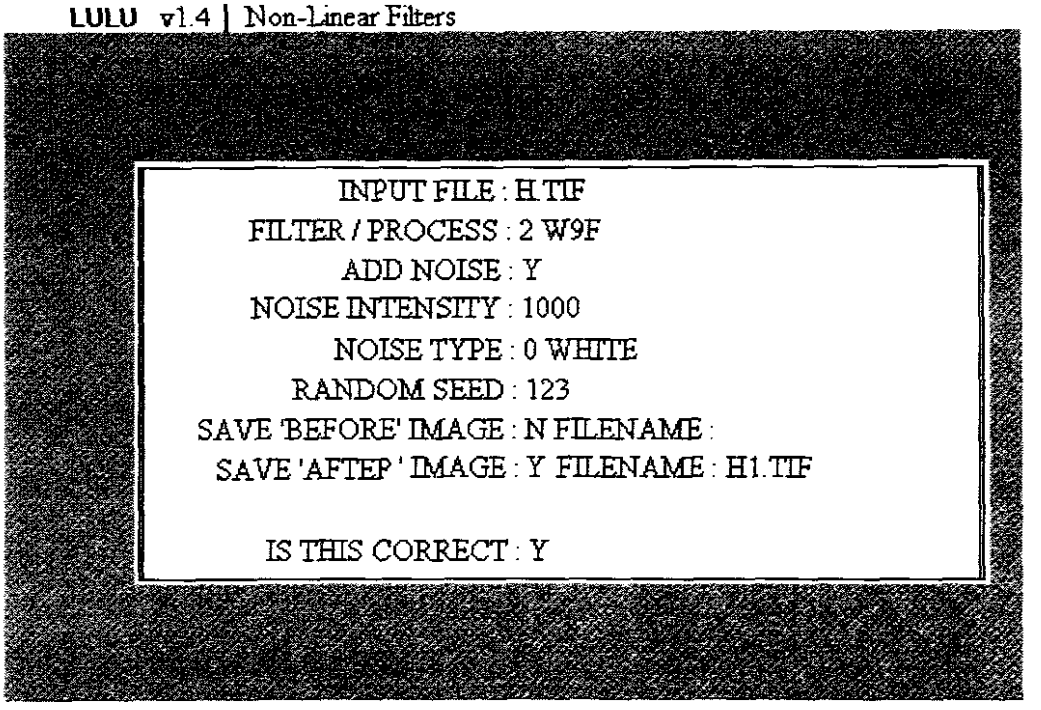
If 'S' is pressed, a screendump with both images displayed on the screen will be saved. This feature was used extensively in the illustrations in this thesis, because the right-hand image is usually the smoothed product of the left-hand image. Feature comparisons can be made much easier if both images are displayed simultaneously, rather than flipping screen pages.

6.4 The LULU program

The **FILTER** program was the prototype for the set of **LULU** programs.

---

[1] The opposite to the term *weakest*, as defined by Rohwer [80].



SCREENDUMP 6.4 The title page of the *LULU* program.

The difference between the **FILTER** and **LULU** programs is that the **LULU** programs had algorithms designed to compute *complete* images. The *floor* and *ceiling* composites are now not transparent to the user and only the final image is displayed after smoothing. Three useful linear algorithms are also added to the menu of programs.

Screendump 6.4 shows the work page of this program. Note that only the menu is different to FILTER and can be summarised as follows:

0 and 1 select the five-point and nine-point averaging filters,

2 and 3 select the five-point and nine-point median algorithms;

4 and 5 select the five-point and nine-point LULU algorithms,

6 selects the LULU_2VH algorithm, i.e. one dimensional horizontal LULU sweeps, followed by vertical sweeps[1].

---

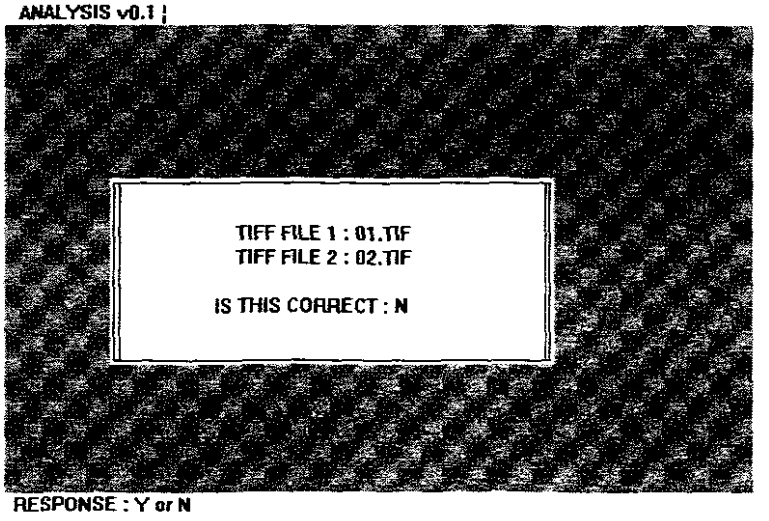
[1] Only the fastest one-dimensional hybrid method is included in **LULU**. See also §6.6.6.

7 selects a tool to enhance the brightness of an image, §2.4.

8 displays the histogram of the input file, similar to the same option in the **FILTER** program.

6.5 The ANALYSIS program

The **ANALYSIS** tool has been developed to obtain some statistical information of the image data, before and after smoothing and filtering. Screendump 6.5.1 shows the front page of this program. The program only requires two images to compare, 01.TIF and 02.TIF as in the example. The menu is built into a *HELP* function, **F1,** in an effort to simplify the operation of the program.



Screendump 6.5.1 The cover page of the *ANALYSIS* program.

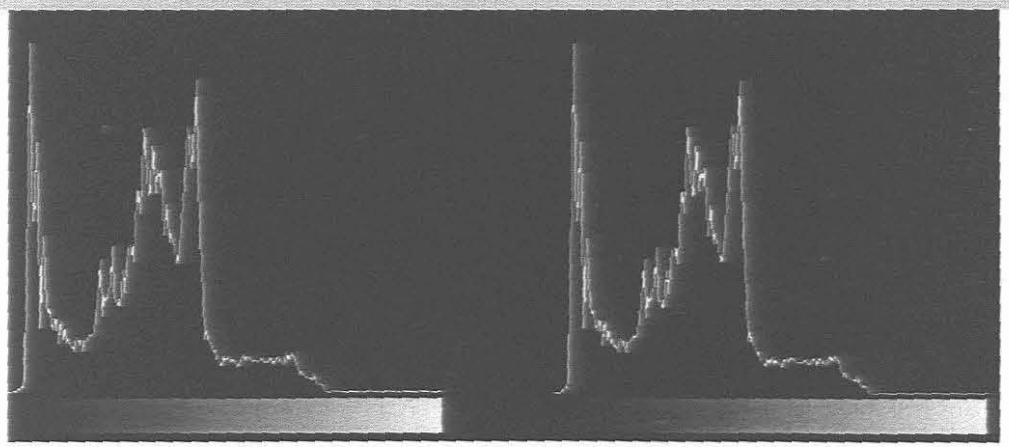
The following menu summarises the contents of this help function:

F1 Calls the *HELP* function.

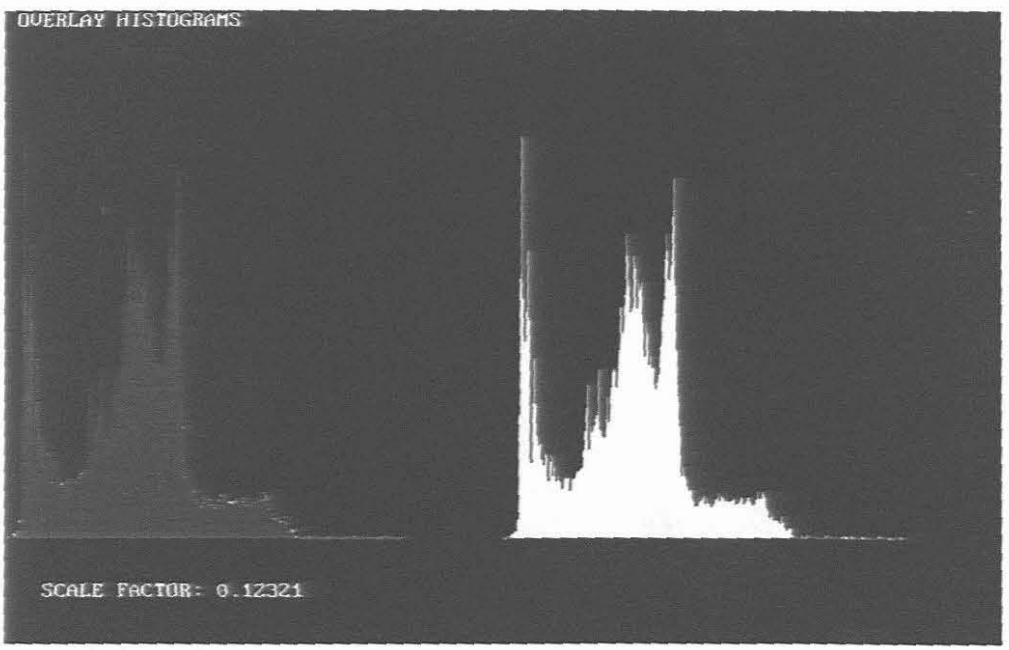
D Displays both images on the screen.

H Recalculate and display both histograms of the two images (see screendump 6.5.2). Note that the histogram on the left clearly shows the rippling effect of image noise in the original image 01.tif.

O This function performs an overlay of the two histograms (on both sides) so that differences in data between the input and output file can be visualised. Screendump 6.5.3 illustrates this effect.

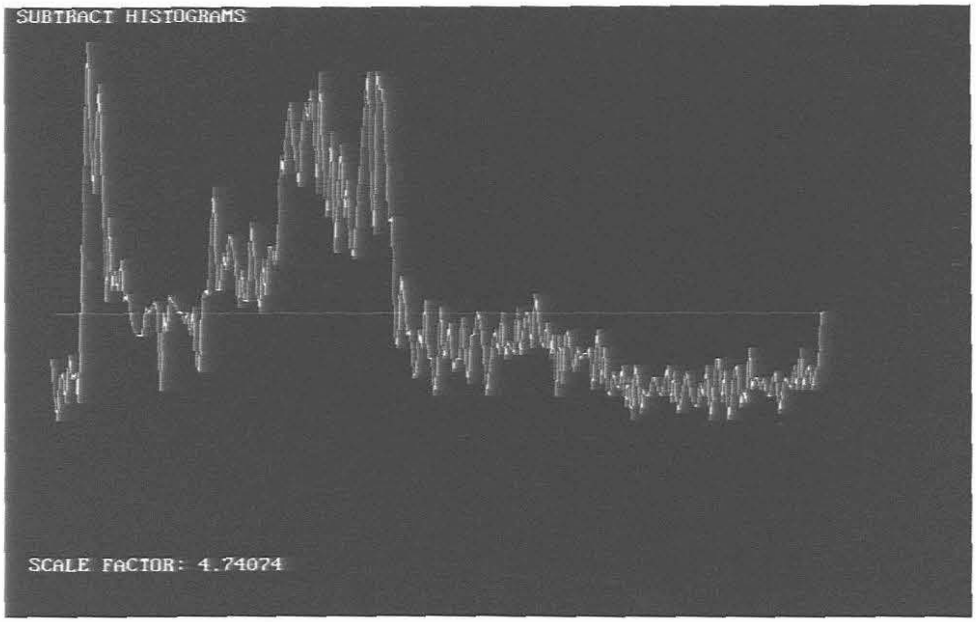


Screendump 6.5.2 Histogram screendump.



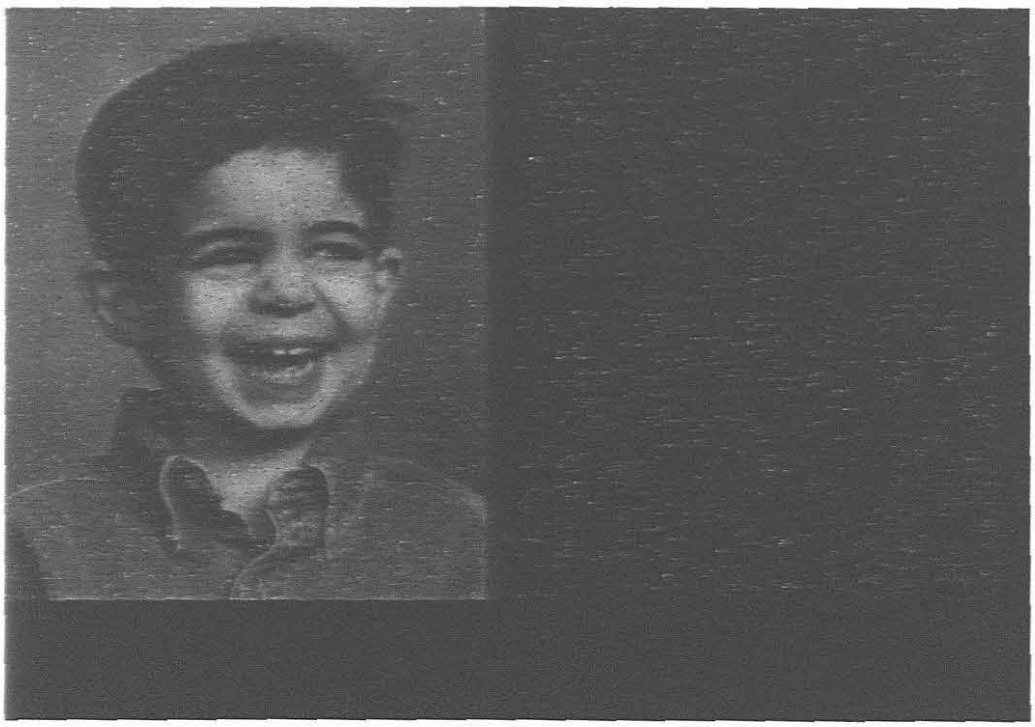
Screendump 6.5.3 Histogram overlays.

_ This function subtracts the data in the histograms and displays the resultant graph (screendump 6.5.4).



Screendump 6.5.4 Graph of difference of two histograms.

= This function subtracts the images so that the noise that has been removed by smoothing can be observed in a resultant image (screendump 6.5.5).



Screendump 6.5.5 The difference between a 'noisy' image and the com-

pletely cleaned (smoothed) one, gives the noise that existed in the original image.

1	Displays the histogram of image 1 full screen.
2	Displays the histogram of image 2 full screen.
S	Saves a screen dump of the full screen to a disc file.
ALT1	Saves the histogram data of image 1 to file.
ALT2	Saves the histogram data of image 2 to file.
ESC	Exit program.

Interesting experiments can be conducted by smoothing remaining noise clusters. It can be shown that if an image is severely contaminated with noise, and the noise remaining after smoothing is smoothed again, some of the original features of **O** can be observed. This is due to noise pixels adhering to the edges of the original image during the smoothing process.

6.6 Programming code

The C code for some of the kernel algorithms in the set of **LULU** programs are listed in this section. A complete listing of the **LULU** program can be found in the appendices.

6.6.1 A two-dimensional filter

The five-point filter program, as defined in §4.1, will have the following C code embedded in the algorithm:

```
for ( x = 1; x < TF_ImageWidth-1; x++ ) {
        tval = prev_array[x] + next_array[x];
        tval += curr_array[x-1] + curr_array[x+1] + curr_array[x];
```

```
        tval /=5;

        new_array[x] = tval;
        }
```

The data will be written to a *VESA* screen configuration, using the following command [3, 57]:

```
vsa_raster_line(out_x, (unsigned)(out_x+TF_ImageWidth-1), y, new_array);
```

Note that for a nine-point sweeping window, the body of the loop will be:

```
    tval = prev_array[x] + prev_array[x-1] + prev_array[x+1];
        tval += next_array[x] + next_array[x-1] + next_array[x+1];
        tval += curr_array[x-1] + curr_array[x+1] + curr_array[x];
        tval /= 9;
```

6.6.2 Image enhancement

The sub-program for linear image enhancement (*sharpening*), as defined in §2.4.2, is listed below [73]:

```
    void apply_sharpen( void )
    {
        float IMP_Kernel[9];
        int i,j,m,n,hf_kern,index,width,height,pixel,x1,y1,x2,y2;
        unsigned char array[72,69];
        float f_pix;
```

```
IMP_Kernel[0] =  0.0;
IMP_Kernel[1] = -0.5;
IMP_Kernel[2] =  0.0;
IMP_Kernel[3] = -0.5;
IMP_Kernel[4] =  3.0;
IMP_Kernel[5] = -0.5;
IMP_Kernel[6] =  0.0;
IMP_Kernel[7] = -0.5;
IMP_Kernel[8] =  0.0;

x1 = (int)(TF_ImageWidth-1);
y1 = (int)(TF_ImageLength-1);
x2 = 10+(unsigned)TF_ImageWidth;
y2 = 0;

width = x1+1;
height = y1+1;

hf_kern = 3/2;
for ( j=0; j < height; j++ ) {
for ( i = 0; i < 3; i++ ) {
index = MAXVAL( j+i-hf_kern, 0 );
index = MINVAL( index, y1 );
vsa_get_raster_line( 0, x1, index, array+i*width );
}

for ( i = 0; i < width; i++ ) {
f_pix = 0.0;
for ( n = 0; n < 3; n++ )
        for ( m = 0; m < 3; m++ ) {
```

```
                    index = MAXVAL( i+m-hf_kern,0 );
                    index = MINVAL( index,width-1 );

                    index = index +n * width;

                    f_pix += ((float)array[index]) * IMP_Kernel[m+n*3];
                    }

                f_pix = MAXVAL( f_pix, 0.0 );

                pixel = (unsigned char) MINVAL( f_pix, 255.0 );

                vsa_set_color( pixel );
                vsa_set_pixel( i+x2, j+y2 );
                }
    }
}
```

6.6.3 Two dimensional median code

The simplicity and relative power of the median techniques for one-dimensional smoothing can be expanded for applications in [1]. Instead of using elements from a one-dimensional string, the median is obtained from the nucleus and the north, east, south and west pixels in a five-point sweeping algorithm (see §3.3 and §4.1).

---

1 Median smoothers are often combined with *alpha-trimmed* filters to form hybrid algorithms, [21,56].

2 *Weighing techniques* are useful when larger sweeping windows are used and the weight of the pixels nearer to the nucleus are loaded for better localised support, [44].

The C declarations for this code are:

```
for ( x = 1; x < TF_ImageWidth-1; x++ ) {
        west = curr_array[x-1];
        east = curr_array[x+1];
        north = prev_array[x];
        south = next_array[x];
        nucleus = curr_array[x];

        out_array[x] = MEDIAN5( west, east, north, south, nucleus );
        }
```

Similarly, code for the nine-point median smoother could be written as:

```
for ( x = 1; x < TF_ImageWidth-1; x++ ) {
        out_array[x] = MEDIAN9( curr_array[x-1], curr_array[x],
                        curr_array[x+1],
                        prev_array[x-1], prev_array[x], prev_array[x+1],
                        next_array[x-1], next_array[x], next_array[x+1] );
                }
```

Although the class of two-dimensional median algorithms appears to be very strong in their smoothing capabilities, they suffer from the same drawbacks but more severe as mentioned for the one-dimensional case in §3.3. What is most significant in comparing a **LULU** algorithm with the corresponding median algorithm in practice, is that the **LULU** method clearly has better edge-preserving qualities, requires less computations and is idempotent [15,81]. The examples of chapter seven also clearly demonstrates this fact.

6.6.4 The LULU_2D/5W program

The *floor* part (§4.2) of the **LULU_2D/5W** algorithm can be coded as:

```
for ( x = 1; x < TF_ImageWidth-1; x++ ) {
        dest[x] = MAX4( MINVAL( curr[x-1], curr[x] ),
                        MINVAL( curr[x+1], curr[x] ),
                        MINVAL( prev[x], curr[x] ),
                        MINVAL( next[x], curr[x] ) );
                }
```

while the *ceiling* section is:

```
for ( x = 1; x < TF_ImageWidth-1; x++ ) {
        dest[x] = MIN4( MAXVAL( curr[x-1], curr[x] ),
                        MAXVAL( curr[x+1], curr[x] ),
                        MAXVAL( prev[x], curr[x] ),
                        MAXVAL( next[x], curr[x] ) );
                }
```

The code is straightforward and simple to program. As in the previous algorithms of this nature, MAXVAL, MINVAL, MAX4 AND MIN4 are user-defined functions which are called in the **LULU** algorithms.

6.6.5 The LULU_2D/9W program

The code in §6.6.4 can be modified for the nine-point *floor* algorithm (§4.3.1):

```
for ( x = 1; x < TF_ImageWidth-1; x++ ) {
        new_array[x] = MAX8( MINVAL( curr_array[x-1], curr_array[x] ),
```

```
                MINVAL( curr_array[x+1], curr_array[x] ),
                MINVAL( prev_array[x], curr_array[x] ),
                MINVAL( prev_array[x-1], curr_array[x] ),
                MINVAL( prev_array[x+1], curr_array[x] ),
                MINVAL( next_array[x], curr_array[x] ),
                MINVAL( next_array[x-1], curr_array[x] ),
                MINVAL( next_array[x+1], curr_array[x] ) );
        }
```

Similarly the *ceiling* code will be:

```
for ( x = 1; x < TF_ImageWidth-1; x++ ) {
        new_array[x] = MIN8( MAXVAL( curr_array[x-1], curr_array[x
                        MAXVAL( curr_array[x+1], curr_array[x] ),
                        MAXVAL( prev_array[x], curr_array[x] ),
                        MAXVAL( prev_array[x-1], curr_array[x] ),
                        MAXVAL( prev_array[x+1], curr_array[x] ),
                        MAXVAL( next_array[x], curr_array[x] ),
                        MAXVAL( next_array[x-1], curr_array[x] ),
                        MAXVAL( next_array[x+1], curr_array[x] ) );
                }
```

6.6.6 The LULU_2D/HV and LULU_2D/VH programs

Screendump 4.4.1.1 shows the effect of one-dimensional **LULU** sweeps on a two-dimensional picture. In practical PC programming, as highlighted above, it is evident that the one-dimensional horizontal sweeps will be much slower than the vertical sweeps

when performed on the entire image. This due to the fact that the pixel elements in the horizontal case must be addressed one by one, while the whole raster vector can be processed in the vertical sweep application.

Due to the relative simplicity of the **LULU_HV** algorithm, the verification of code for this program is omitted.

6.7 Graphics programming

Programming for graphics involves much more than merely developing code in a graphic environment. It becomes a lifestyle of continuously collecting images, ideas and writing code segments. Computing has come a long way from the days where humans had to communicate with machines using punch cards and long lists of incomprehensible data. Visual communication using graphic display made the computer industry much more productive in virtually all applications where human interaction is required. Most of the successful general purpose software packages of today have graphics included in one form another. As the power of compact computers increase, graphics becomes more accessible to the man in the street, whether it be for the display of graphs only or for watching real-time television [6, 25].

A logical future expansion of the **LULU** set of programs is hence to collect and organise all related image enhancement software in an object oriented bundle for Windows application, preferably with colour options [66,63].

6.7.1 Colour graphics

Another logical extension of the programming of **LULU** algorithms, is to re-develop the programs for colour smoothing. A quick-and-fast technique might seem to select a colour

model for C programming and to repeat the smoothing in batches for red, blue and green. Although this method works, it was decided not to include colour smoothing as part of this thesis and rather look at this topic as a completely separate research field that has to be a thorough undertaking in all its research and development stages. A grey-scale image primarily presents a picture, where a colour image is much more complex in the ways it represents the display of light within the electromagnetic spectrum. To repeat the work of this thesis to include colour would therefore firstly involve an investigation of different colour models for efficient colour display on the hardware used. It also has to be kept in mind that colour representation on a visual display unit and a hard copy device is often largely different. Although colour printing has recently become quite inexpensive, high quality wax printing is still not within the reach of most budgets. Once a suitable colour model is chosen, smoothers and filters can be tested for visual efficiency and a set of colour algorithms for noise reduction can be formulated [19,92].

6.7.2 Acceleration techniques

Fast display of large digital images plays a significant role in image processing. Images from satellites, for example, are normally improved in one or other way before released. Interference or picture noise has to be removed before results can be published. The quality of an image can be seen in direct proportion to the amount of processing it will require for display and manipulation. This is particularly of importance for public broadcasters, medical applications and the military where high photographic precision is of crucial importance. In the past most of these methods of improving the quality of images were time-consuming, costly and often had to be sacrificed for speed.

The computer industry has for decades had an insatiable need for speed enhancement and this will continue in the future. Even with the best sequential hardware and software developments, applications that require more processing power will remain. Dedicated graphics engines frequently make use of parallel processing [46,56,79] and VLSI [9,74]

techniques and it is thus also logical to encourage further research for similar parallel **LULU** structures.

6.8 The decomposition of the two-dimensional problem for parallel processing

It is a well-known fact that smoothing, when utilised as a part of image enhancement, should be a fast and cost effective process [30,90]. If images are to be cleaned of noise in a real-time situation, like live television, high resolution picture frames have to be cleaned in the order of twenty to fifty frames per second, which is really beyond the computing power of conventional hardware without sacrificing quality. Various *look-ahead* schemes have been advocated in the literature [64,93] to try and spread the work-load to enable digital filtering to take place before an image is displayed. Although viable, these methods are known sometimes to lead to the introduction of stability problems.

Another alternative is to investigate the natural *granularity*[1] of a smoother process and to search for a suitable parallel programming framework. One of the first decisions to be taken in this process is to determine whether a synchronous (*SIMD*[2]) or an asynchronous method (*MIMD*) should be opted for [16,86]. If the problem is reduced to merely parallelising the smoothing problem for the best *speed-up*[3] on available hardware, some of the optimality of a solution may well be lost and one would have to take the *efficiency*[4] of the solution into account.

The intention is not to formulate new parallel algorithms in this section, but rather to give the interested reader some indication of how a **LULU** smoother could be programmed to be significantly faster than on most sequential computers. From a theoretical view-point

---

1 Each process is associated with a separate process and controlled by its own software [29].

2 SIMD: single-instruction multiple data, MIMD: multiple-instruction multiple data.

3 The ratio of time taken to solve a problem on a parallel computer versus the time taken to solve the same problem on a sequential computer [39, 83].

4 Efficiency is normally described as the ratio between speed-up and the number of processors utilised to solve a given problem [39,67].

the assumption can be made that ample processing power is available in compact, efficient parallel processing units, with very little communication delay between the processors. This type of hardware is realisable in VLSI configurations for low-latency pipe-lined configurations [64, 74]. The problem thus reduces to the most efficient implementation of parallel processes; in this case the application of a suitable parallel decomposition for the class of **LULU** two-dimensional smoothers. From the numerous smoothing possibilities, a single example, namely the **LULU_2D/5W** smoother will be selected and analysed for parallel implementation.

6.8.1 The complexity of LULU_2D/5W

If an nxn input matrix, X, is used to display the (n-2)x(n-2) image, **X**, it can be shown that the floor operation of the five-point window, $w_{i,j}$ at position (i,j), takes 10 comparisons. This has to be repeated n-2 times for a row sweep, for the n-2 rows. The total number of operations for the floor sweep is therefore $10(n-2)^2$ comparisons for X. The ceiling sweep will take the same number of operations, giving a resulting complexity of $20(n-2)^2$ operations for the complete smoothing process.

6.8.2 Computing sub-matrices in parallel

A logical subdivision of the matrix X would be for synchronous operation on a number of parallel processors. A logical way of planning a parallel process is to divide the main problem into sub-problems for parallel computing. Consider the four sub-matrices of X, each dedicated to the four identical processors, P_i, i=1,2,3,4.

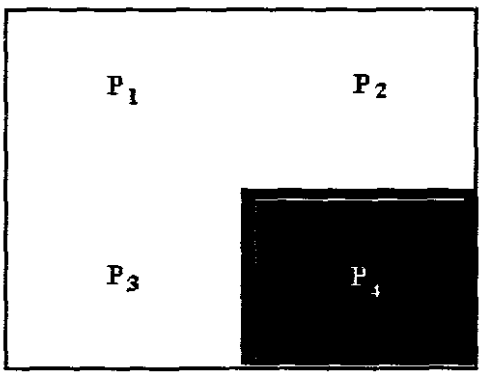


Figure 6.8.2 illustrates the matrix X, with the data area for P_4 highlighted. The computational time for such an arrangement can usually be broken down into three basic parts, namely initialisation of data (T_i), computational time (T_c) and finally the communication and display (T_d) of the resultant data.

Figure 6.8.2

The initialisation of data will include setting up each processor with its own data. Other complications can arise: for instance, border data might be required from two adjacent matrices for the calculation of a smoothed sub-matrix. For example, P_4, will require a row from P_2 and a column from the dataset in P_3. It is obvious that the objective of this exercise is to cut computing time to approximately 25% of the time[1] it would take a single processor of the same strength to compute the complete algorithm. This can only be achieved in a near optimal situation where the processors are finely synchronised and communication times are virtually negligible.

Definition 6.8.3 The speed-up of a parallel process, using n papro's, can be defined as $S_n = T_s/T_p$, where T_s is the time taken on an equivalent serial processor to solve the same problem which is solved in T_p, the parallel time using a parallel processor.

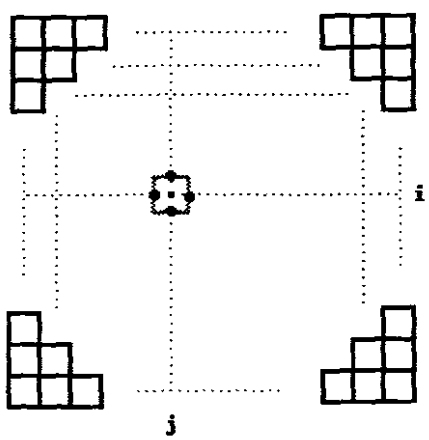
The speed-up can be shown to be $S_4 \approx 1/\ 0.25 = 4$, if the times T_i and T_c are ignored. If the problem is decomposed to sixteen sub-matrices and the same assumptions hold, the execution time will be approximately sixteen times faster than the serial time.

---

[1] A speed-up of four.

It can be pointed out that if the matrix is large and enough parallel processors are available, a situation arises where $S_\infty = \lim_{n\to\infty}(T_s/T_n)$, which leads to complications, because $T_n \to 0$ as n becomes very large. In the literature, the well-known *Amdahl's law* [29,20] addresses this topic.

It is not unrealistic to think of *pixelputers*[1] to solve problems of this nature. If such a parallel processing configuration is available, with the usual assumptions about processing speed and inter-communication times between individual papro's, an algorithm can be devised based on the smallest sub-matrices to house the sweeping five-point **LULU** windows. Figure 6.8.4 illustrates such a scheme, with $P_{i,j}$ the processor element at the matrix position (i,j).



It is not difficult to see that this parallel arrangement only needs a total computational time for the twenty comparisons for each pixel if *interlocking* or *data clashes* [49,87] can be avoided.

Figure 6.8.4

In conclusion, it might be pointed out that many interesting alternatives to the parallelisation of **LULU** algorithms can be designed. It might be worthwhile to mention that theoretical speedup of logarithmic nature is possible with certain array processing architectures, but in practice it might fail due to unrealistic start-up times for data preparation [65,87].

---

1 A term decided on by the author in the absence of a suitable name for an image processing computer where each pixel is basically driven by its own processor. The 2m-processor of Cray Research is a SIMD example of such a machine [67].

CHAPTER SEVEN

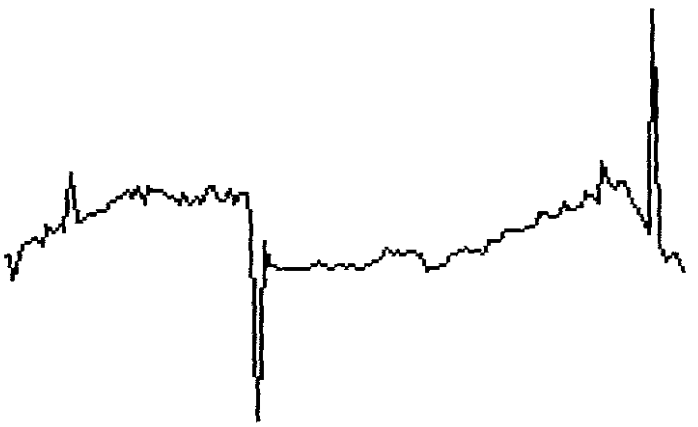
The Practical Application of LULU Operators

7.1 The one-dimensional problem

The problem of removing impulsive noise from a valid signal sequence in one dimension was described in chapters' three and five. A typical application of one-dimensional **LULU** smoothing arises when an acoustic wave has to be cleaned of impulsive noise [19, 78]. The resulting signal must be as distortion free as possible, while retaining the original wave trends. Research of the one-dimensional **LULU** structures has shown that **U, L** and some predictable unsymmetric pairs of these operators, will leave a monotonic increasing or decreasing signal unaltered if no noise is present [62,82]. This is the first and most desirable characteristic expected of a good non-linear smoother. Spot noise has to be removed and the original sequence corrected as accurately as possible. The advantage of this type of smoothing to other well-known smoothers, like the median and its variants, is described in chapter three and also referred to in [80].

As an example to finally illustrate the effect of **LULU_1D** a function was constructed to simulate a portion of a typical sound wave. Let **x** be the sequence as expected in practice:

$$\mathbf{x} = \zeta + \varepsilon_g + \varepsilon_s$$



For experimental purposes a known sequence, ζ has low-frequency noise, ε_g, as well as high-frequency noise, ε_s added.

The original wave, **x**, is illustrated in figure 7.1.1.

Figure 7.1.1 The original wave, **x**.

For the sake of comparison, two basic procedures are tested in addition to the **LULU_1D** algorithm.

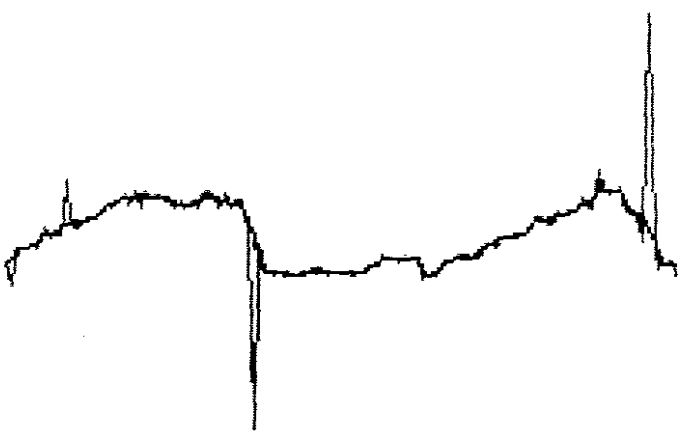
Let Φ be a linear process (averaging) as defined in §3.2. The window-size is restricted to the smallest frame that will allow the median, i.e. $w_i = 3$. The effect of this filter is shown in figure 7.1.2. The *smearing* that occurs at the two large outshooters is clearly visible in the graph.

Figure 7.1.2 $\mathbf{y}_\Phi = \Phi(\mathbf{x})$

For the sake of comparison, the window size of the running window will be fixed in these examples. If, however, a larger window size was used for averaging, the smearing at the outshooters would have spread out over a wider region [15,62]. In practice the window size is chosen to suit the type of noise which is expected. If no *a priori* knowledge about the noise is available, it is recommended to use the smallest possible window size as not to change the original data unduly.

Let **M** represent the median algorithm, with $w_i = 3$. The result of $\mathbf{y_M} = \mathbf{M(x)}$ is shown in figure 7.1.3. The effectiveness of a small median sweeping window is illustrated by the fact that both outshooters are eliminated successfully.

Figure 7.1.3 $\mathbf{y_M} = \mathbf{M(x)}$



If Λ is the **LULU-1D** process, with $w_i = 3$, then the result of the algorithm is shown in figure 7.1.4. The power of the non-linear smoothers **M** and Λ are clearly illustrated in the graphs 7.1.3-4.

Figure 7.1.4 $\mathbf{y_\Lambda} = \Lambda(\mathbf{x})$

For further comparison between the filter and the smoothers of this example, it is interesting to investigate the following norms:

Let $\mathbf{L_\infty}$ be the *maximum norm* and $\mathbf{L_2}$ the *Euclidean norm* of the differences between the sequences **x** and **y**.

7.1.5 $\mathbf{L}_{\infty} = (\sum_{i=1}^{n} (\max|x_i - y_i|))/n$

7.1.6 $\mathbf{L}_2 = (\sum_{i=1}^{n} (x_i - y_i)^2/n)^{1/2}$

Similarly,

Let $\mathbf{L}_{\infty}^*$ be the ***maximum norm*** and $\mathbf{L}_2^*$ the ***Euclidean norm*** between the sequences ζ and **y**, $\forall$ I $\in$ S, the set of single subscripts.

7.1.7 $\mathbf{L}_{\infty}^* = (\sum_{i=1}^{n} (\max|\zeta_i - y_i|))/n$

7.1.8 $\mathbf{L}_2^* = (\sum_{i=1}^{n} (\zeta_i - y_i)^2/n)^{1/2}$

The fact that the original *noise-free* function, ζ, is known, allows the measurement of the error between ζ and **x**. Even if a priori knowledge existed on the noise present in **x**, it would be advisable to first remove spot noise and then pay attention to Gaussian noise. In this simulated example the same procedure will be followed and an estimate of the error introduced by the addition of spot noise is obtained by first smoothing to the function **x** and then comparing it with ζ and **x**.

Table 7.1.8 shows a summary of these calculations where the output of the smoothers and the filter is compared with the input sequence:

	$\zeta \mid \mathbf{x}$	$\mathbf{y}_\Phi \mid \mathbf{x}$	$\mathbf{y}_M \mid \mathbf{x}$	$\mathbf{y}_\Lambda \mid \mathbf{x}$
$\mathbf{L}_\infty$	133.6	123.2	112.2	110.7
$\mathbf{L}_2$	3.3	3.2	2.4	2.7
		$\mathbf{y}_\Phi \mid \zeta$	$\mathbf{y}_M \mid \zeta$	$\mathbf{y}_\Lambda \mid \zeta$
$\mathbf{L}'_\infty$	-	65.7	24.6	28.7
$\mathbf{L}'_2$	-	2.7	1.6	1.8

Table 7.1.8 Comparison of errors.

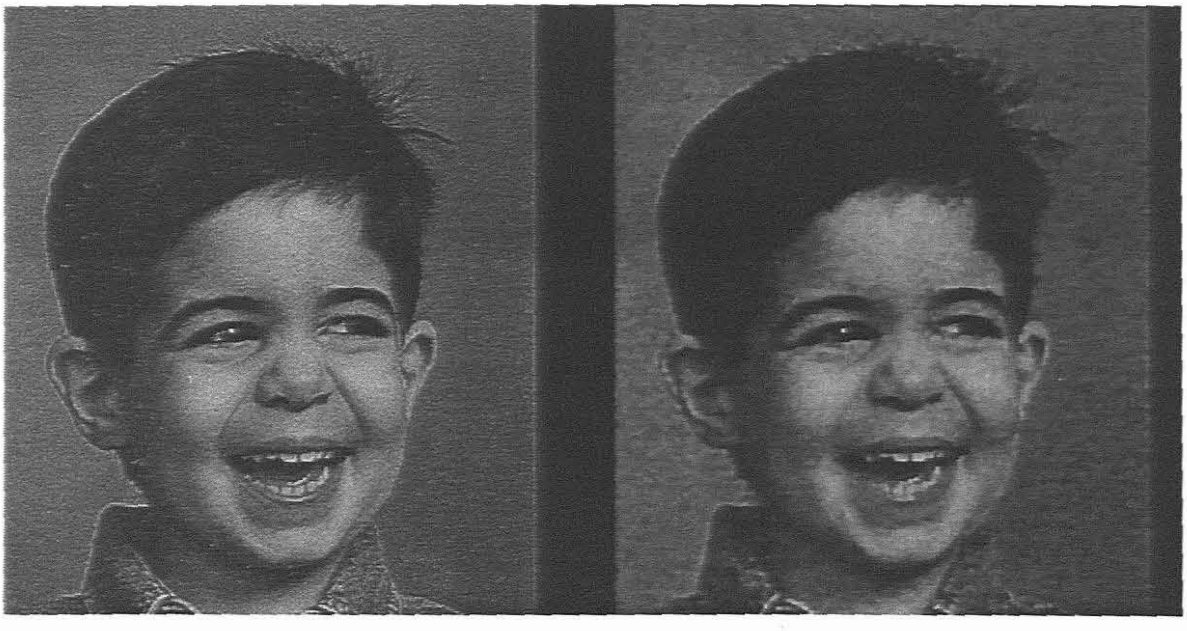
In the first column $\mathbf{L}_\infty$ and $\mathbf{L}_2$ is found between the sequences ζ and **x**. This measurement gives some indication of the error introduced with the addition of Gaussian noise, ε_g, and impulsive noise, ε_s. Although this is an artificial example, it supplies some information of what will happen in practice where information about the noise type is available. In the case where only impulsive noise is present, the function will be freed of the noise and reconstructed as good as possible by **LULU_1D** [80]. If the presence of low-frequency noise is suspected, it would be advisable to follow the smoothing process with a filter. From the above test the superiority of the smoothers is obvious for the removal of impulsive noise. Further demonstrations and particularly the performance of **LULU_1D** algorithms in comparison with the median is reported in the literature [61,80,82].

7.2 LULU_2D in practice

Many two-dimensional test examples that were corrupted by adding artificial noise and hence cleaned have been illustrated in this thesis. That includes simple *OXO*[1], low-resolution binary examples and high-resolution images. The following two examples are selected to further illustrate the difference between smoothing and filtering in the two-dimensional case.

7.2.1 An overexposed image

In the process of trying to improve an image, caution must be taken to select the best method for the noise type.



Screendump 7.2.1.1 **O** $\mathbf{P}_{L9w}$

As a counter-example to show the effect of a non-linear smoother that did more harm than good, an original image[2], **O**, was created by subjecting it to brightness enhancement until the image started deteriorating and showing spot noise. A **LULU_2D/9W** sweep

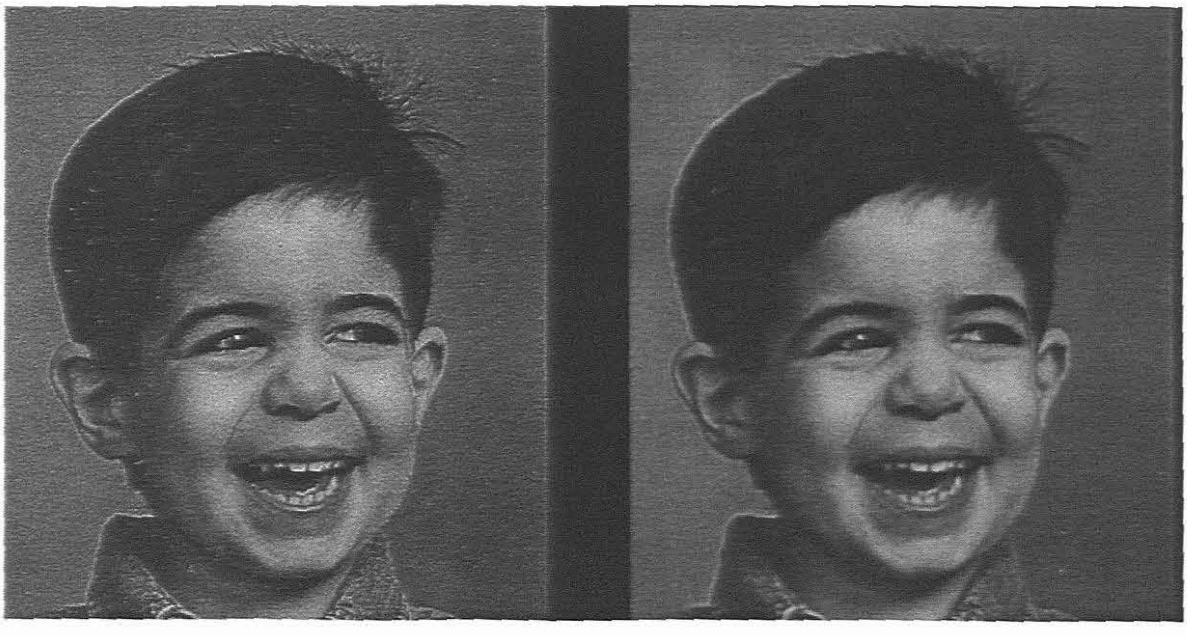
---

1 Binary sub-matrices with empty cells set to zero (0) and the 'on' cells set to X.

2 Image taken from [72].

was performed on the image as shown in screendump 7.2.1.1 and the results compared with that obtained after a **FILTER_2D/5W** algorithm was applied to the same original image. It is easily observed that the linear algorithm restored the image to a certain extent, while the non-linear algorithm caused the image to deteriorate.

The fact that better results were obtained with the linear process of screendump 7.2.1.2 can be attributed to the smaller window size used in the filter and the Gaussian nature of the noise. In most smoothing applications in practice no or little a priori information will be available and care must be taken to select the correct procedure for each class of smoothing applications. To illustrate the effect of the smoothers of this thesis on an example that is severely affected by noise, an image was captured from a very weak television signal and tested with different combinations of smoothers and filters.

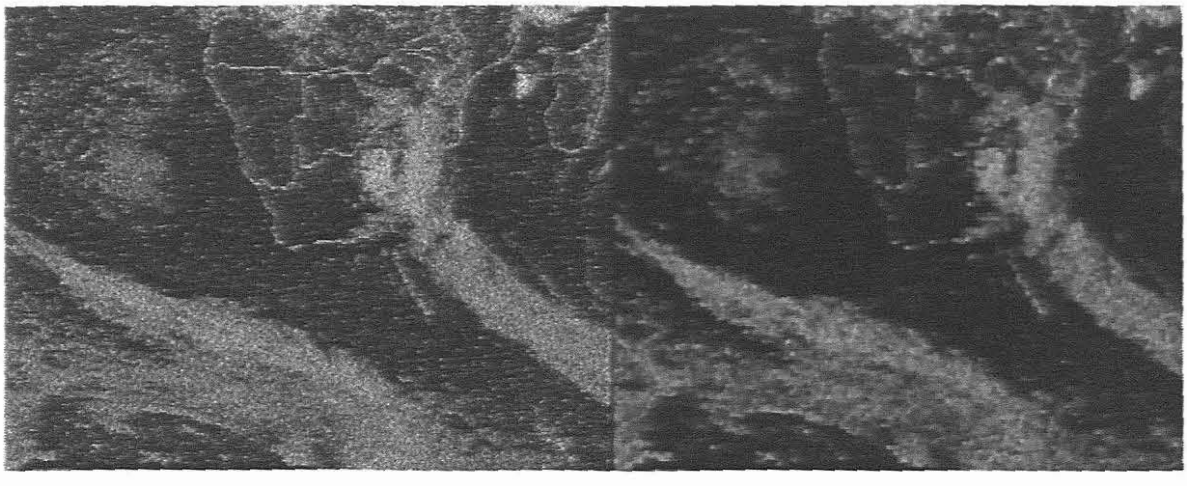


Screendump 7.2.1.2 **O** $\mathbf{P_{F5w}}$

7.2.2 A weak television signal

The original image, **O**, was a weather chart broadcasted on 24 February 1996 in the Cape Town area, South Africa. No other information about the signal was available. By conducting a number of stand-alone tests, it was noted that better results can be obtained by combining non-linear methods (hybrids) in some cases. This can be attributed to the fact that due to oversaturation of impulsive noise, a root solution could not be achieved. By applying further linear filters the visual appearance of the final image can still be improved.

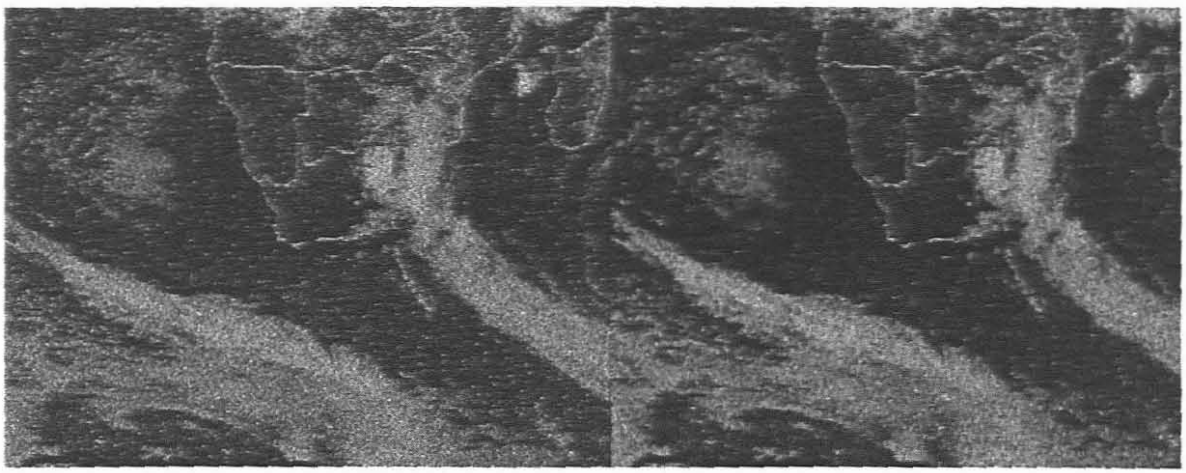
The image, **O**, of screendump 7.2.2.1 was severely affected by spot noise in all levels of grey; very similar to the synthesised examples of chapter four and those in §6.3.5. At a first glance it seemed hardly possible to expect any feasible results by using non-linear pre-filters as was the case with isolated spot noise.



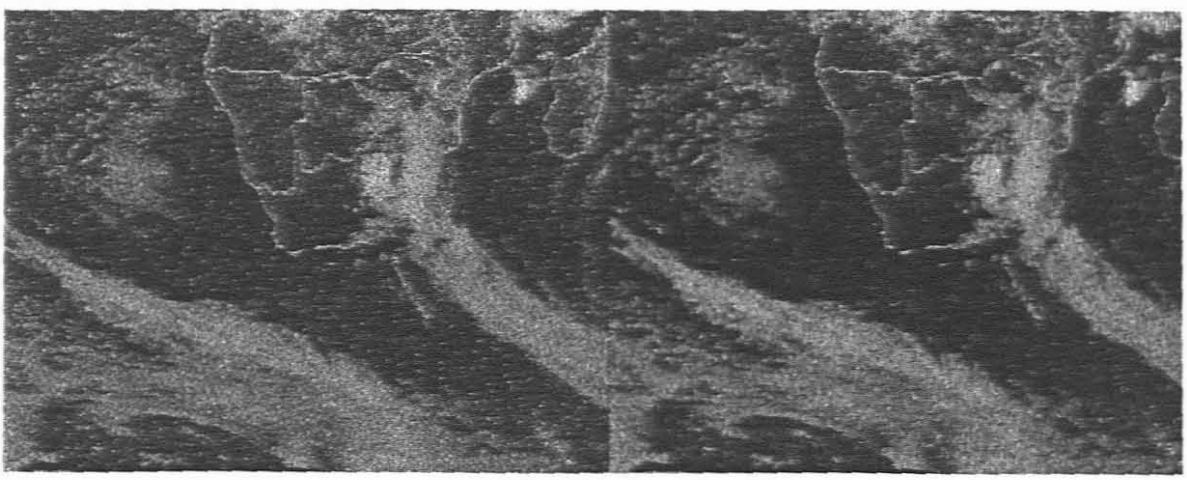
Screendump 7.2.2.1 **O** $\mathbf{P_{F5w}}$

In this case a five-point average filter is tried again, but a significant part of the border line have disappeared and the crispness of the image was affected. Further enhancement will only amplify the quality loss due to changes that have taken place and the filter is not

recommended as a pre-filter. The borders between the countries on the map are dithered and the border line of Madagascar has virtually disappeared. If larger window sizes are to be used, boundary detail will completely disappear. **LULU_2D/5W** (screendump 7.2.2.2) used on the original matrix, gives much better results. Although a fair amount of noise remains, the border lines are preserved and a suitable linear process should enhance the final display to an acceptable standard.



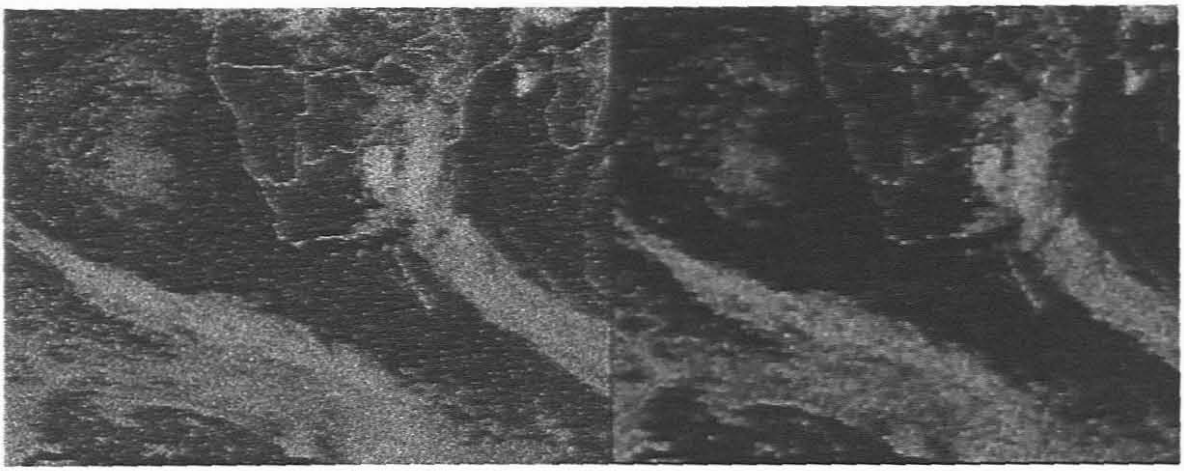
Screendump 7.2.2.2 **O** $\mathbf{P}_{L5w}$



Screendump 7.2.2.3 **O** $\mathbf{P}_{M5w}$

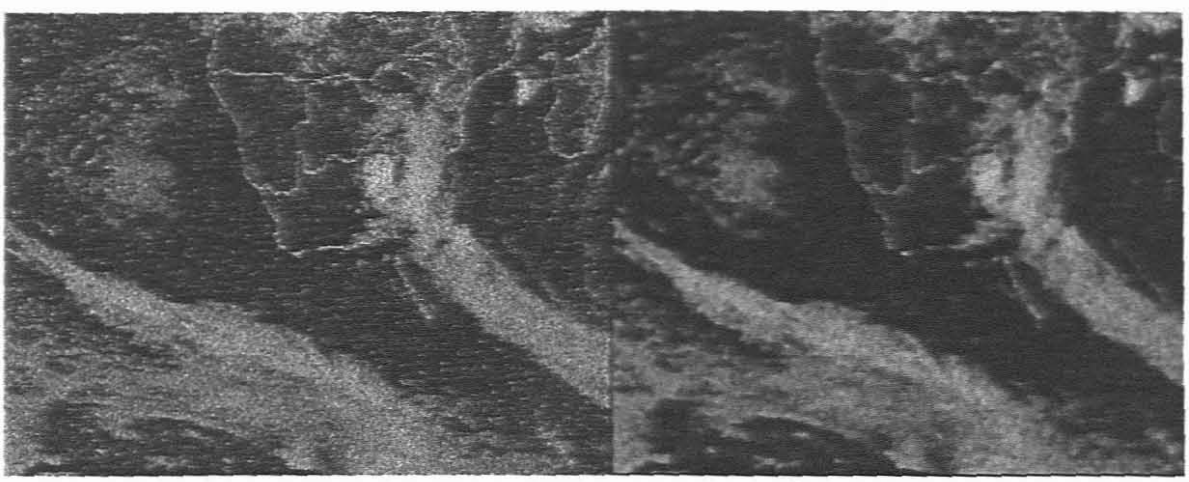
A similar result is obtained with the five-point median (screendump 7.2.2.3), although it seems that the grain of the resulting pixel display of $\mathbf{P}_{M5w}$ has a smoother techture.

In an effort to suppress the noise intensity, a bigger window size can be used for the median and the **LULU** algorithms.



Screendump 7.2.2.4 **O** $\mathbf{P_{L9w}}$

Screendumps 7.2.2.4-7.2.2.6 illustrates the effect of three nine-point smoothers on the original image. The design of these smoothers cause less noise to remain after the smoothers have been applied, but the aggressiveness of the smoothing can effect picture quality in some instances. An exception to this rule is the *weaker*[1] nine-point smoother, **LULU_2D/9W_1**. Due to the *open* nature of the selectors of this smoother, virtually no difference can be detected between the original image and the smoothed one.

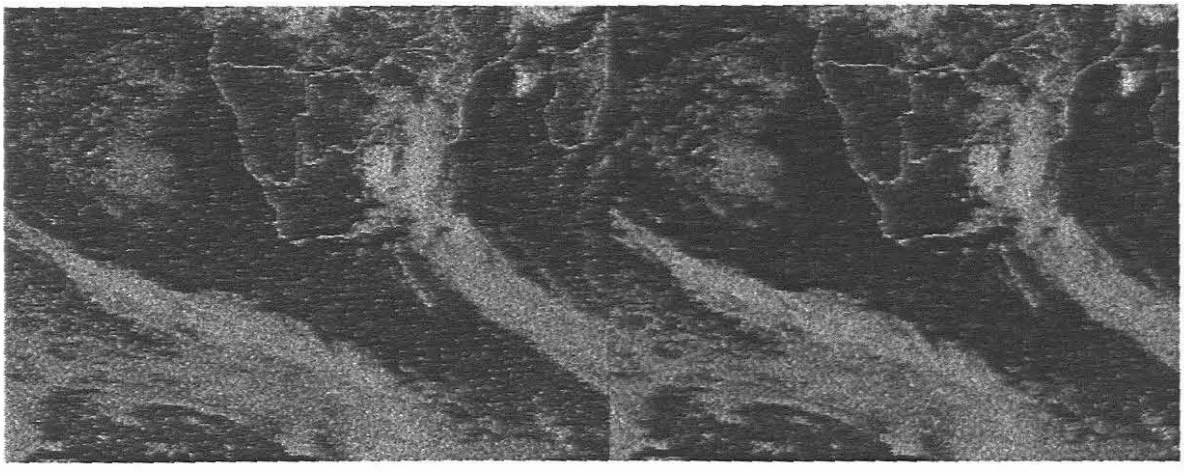


Screendump 7.2.2.5 **O** $\mathbf{P_{M9w}}$

---

1 A term introduced by Rohwer [80] to compare non-linear smoothers.

Depending on which criteria is important for final enhancement of the image, a number of smoothers could have been selected for the pre-filter process. A good proposition would be to use the **LULU_2D/5W** due to the preservation of the border lines and also



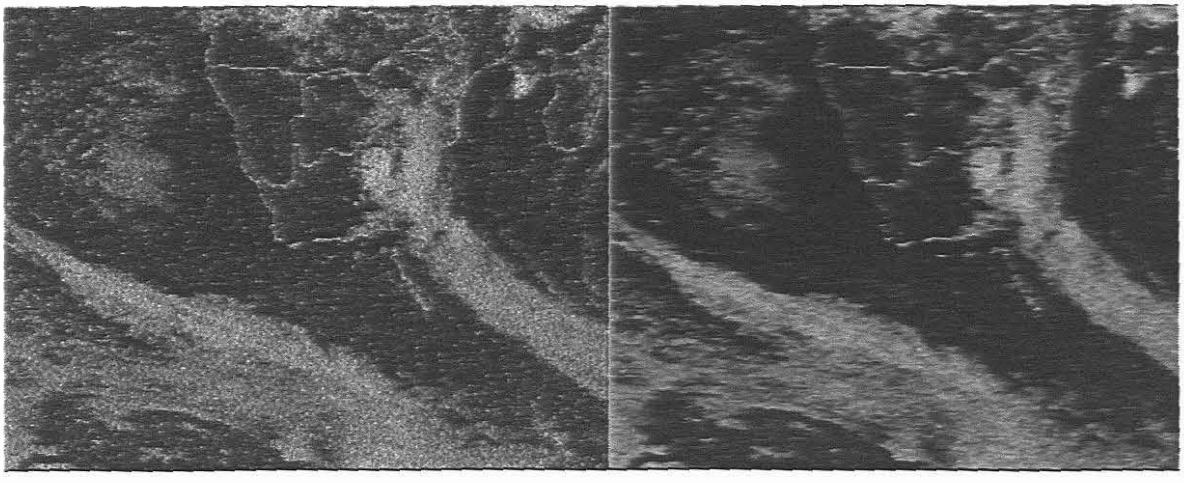
Screendump 7.2.2.6 **O** $\mathbf{P_{L9w1}}$

because idempotency exists. Any of the two median algorithms could be used next as another pre-filter sweeps to *soften* the appearance of remaining noise. The order of such a hybrid computation is important and exhibits the underlying differences in operation of median and max-min algorithms [36].

The process of deciding a suitable smoother for a specific task remains a complex one if no prior information is available. If images are to be smoothed in batches and the noise is predictable, an optimal procedure can be defined and implemented. If images, like video frames, have to be smoothed, the problem becomes more demanding and automation of the process becomes even more complex..

Screendump 7.2.2.7 shows the result of **LULU** when one-dimensional vertical sweeps are followed by horizontal sweeps. Spot noise is removed quite successfully, but vertical thin lines deteriorated. This amplifies an important aspect of smoothing: to what extent should thin lines be retained or should their deterioration be regarded as unimportant?

Paragraph 7.4 pays attention to this problem, while the removal of spot noise in real applications is illustrated in the next section.



Screendump 7.2.2.7 **O** $\mathbf{P_{Lhv}}$

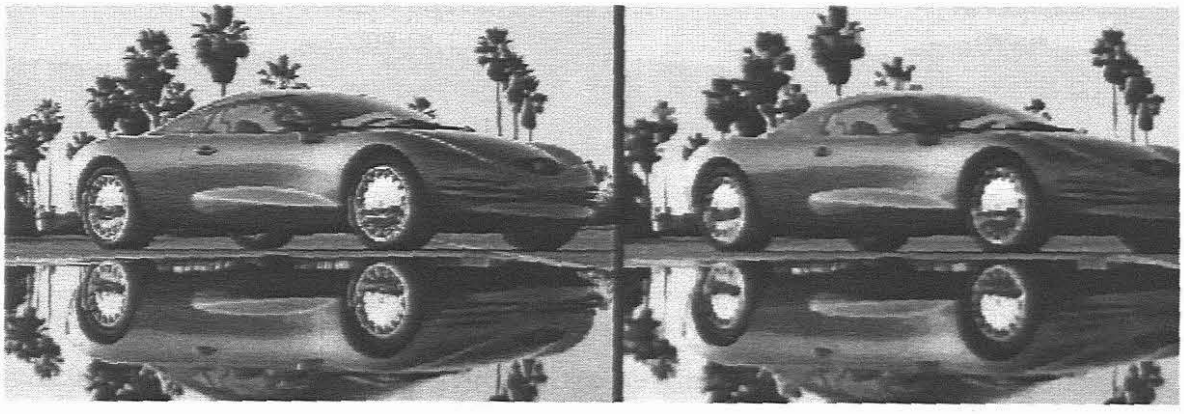
7.3 Two-dimensional images with spot noise

If the noise distribution in an image is sparse, most of the non-linear algorithms researched in this thesis will remove the noise spots, without changing the original image unduly. The test results in §7.2 suggest that a smoother should be selected as a suitable pre-filter before any other enhancement to the image is done. It was also pointed out that it would normally be unwise to use a large window for smoothing if a smaller window can remove noise effectively. Smoothers with idempotency features have a definite advantage to smoothers that have to be repeated to obtain the required results [17,26].

The five point **LULU** algorithm has two distinct advantages in this regard:

7.3.1 The fact that the **LULU** smoother is idempotent, means that no improvement will result from further smoothing with the same smoother and the process will be completed in a predictable and economic number of steps.

7.3.2 By repeating a five point median smoother the final picture will deteriorate with each run of the program due to the composition of the rank based selectors. Screendump 7.3.3 serves as a good example. After only a few repeated applications of the median the definition of the trees started deteriorating and the gloss on the car reduced.



Screendump 7.3.3 **O** $(\mathbf{P}_{M5})^n$

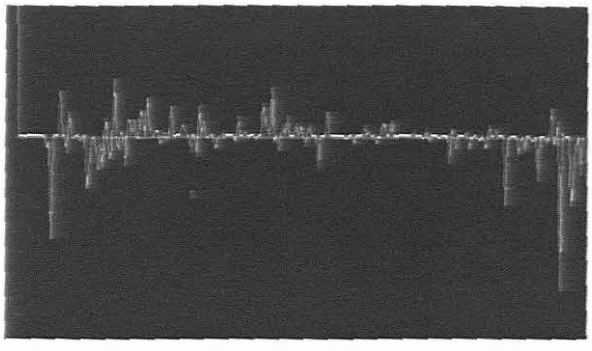
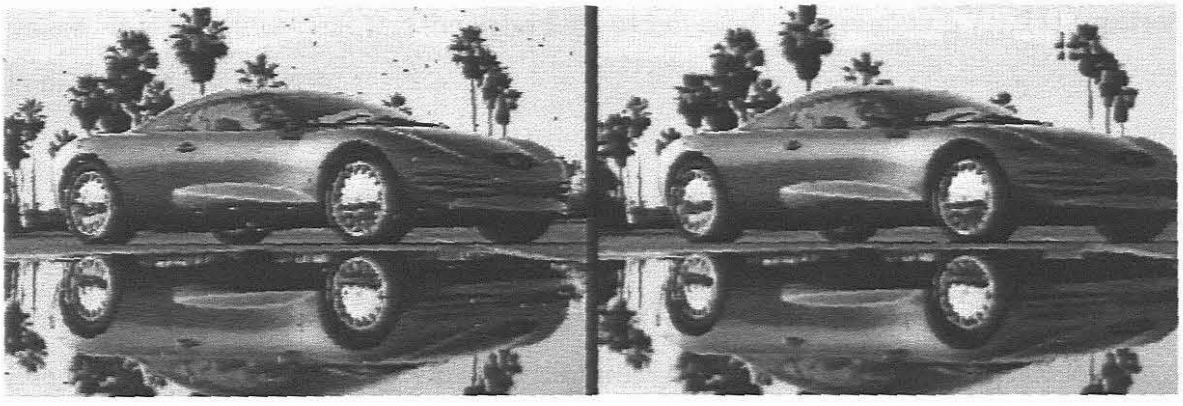


Figure 7.3.4 is a graph of the difference between the histograms of the two images in screendump 7.3.3. This indicates that the image was affected over the full spectrum of grey levels.

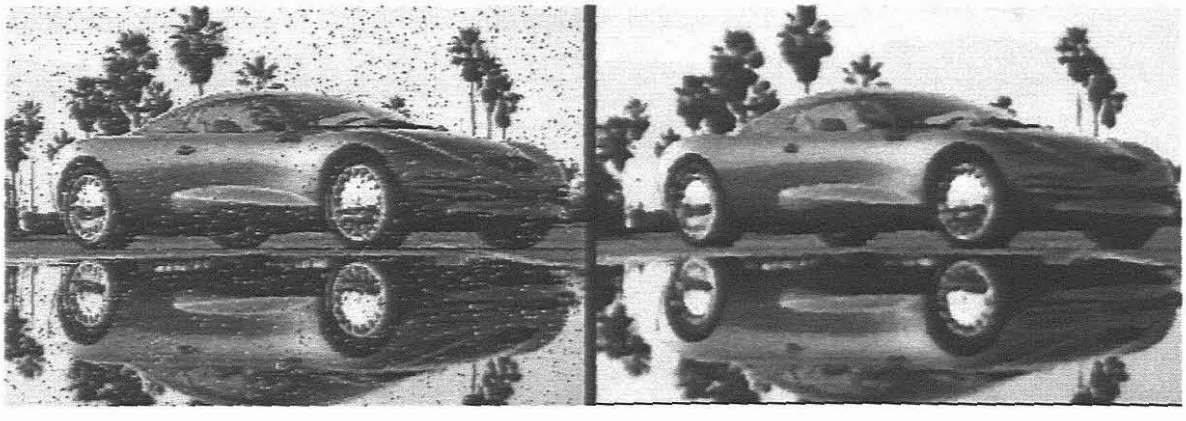
Figure 7.3.4

This is not the case when the **LULU_2D/5W** smoother is employed. Screendump 7.3.5 shows the result of an image that was smoothed with the five point **LULU** algorithm, $\mathbf{P}_{L5w}$.



Screendump 7.3.5 $\mathbf{P}_{L5w}$ $\mathbf{P} = \mathbf{M}_{5w}(\mathbf{P}_{L5w})$

Even though the image was slightly oversaturated with impulsive noise and some noise remained after the smoothing process, the original features of the image did not deteriorate with further sweeps of the **LULU_2D/5W** algorithm. To illustrate how a hybrid pre-filter can be constructed, the **LULU** sweep was followed by a five point median sweep to remove the remaining noise and still retain good image quality.



Screendump 7.3.6 $\mathbf{O}$ $\mathbf{M}_{9w}(\mathbf{O})$

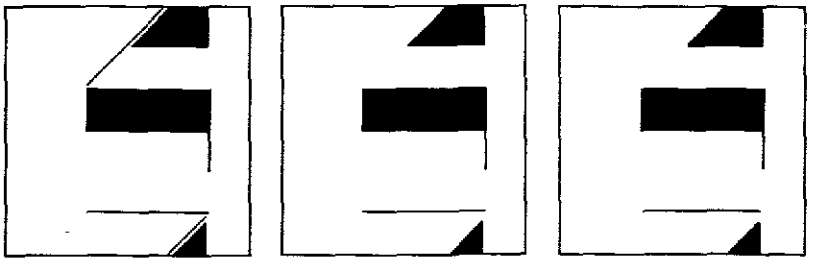
Screendump 7.3.6 shows the same original noisy image that was used to produce the left-hand picture in screendump 7.3.5. If this image is smoothed with a conventional nine-point median smoother, the noise is removed, as can be observed in the right-hand side of the screendump. Although the algorithm did a fine job in removing the noise, it was even more destructive than its five-point equivalent. Fine detail, like the tree trunks, were virtually annihilated. Features on the car also seems much duller than in the original

image. It is therefore clear that the hybrid algorithm used in screendump 7.3.5 is superior to the **MEDIAN_2D/9W**.

The preceding tests prompted further investigation of the smoothing of images where thin lines are present[1]. In the initial stages of the research effort, one of the key issues in the design criteria of new two-dimensional smoothers, was to observe what happens if a thin line is encountered. In a binary picture thin lines have much more significance than in a compound, high-resolution image. This is be practically illustrated in the next section.

7.4 Thin lines in low-resolution images

Often, during video playback, *thin noise lines*[2] are visible on a television screen. These lines could be in any direction, but is commonly observed horizontally due to the movement of the audio-visual tape over the machine heads. The question is obvious: could the interaction of smoothers eliminate or reduce such noise?



(a) **O** **(b)** P_{L5w} **(c)** P_{M5w}

Figure 7.4

It might be worthwhile to first examine the effect of the smoothers on a binary example, such as the one in figure 7.4 (a), with thin lines present. Figure 7.4 (b) illustrates the action of **LULU_2D/5W** on the original image. Note that everything remains intact,

---

1 Much research has been conducted in the fields of *edge-preservation* and *corner-detection*, [38,73, 77].

2 A thin noise line consists out of single pixels forming a line formation.

except for the slanted line that disappears. This was expected, because it was shown in chapter four that the five-point **LULU** operators will treat the pixels of a skew line as noise due to the construction of the window. A *thick*[1] noise line of minimum thickness is constructed when two or more parallel thin noise lines are positioned immediately next to each other. It can quickly be verified that the five-point **LULU** algorithm will have no effect on such a line. Figure 7.4 (c) shows the effect of the five-point median on thin lines. The result seems to be similar to that in figure 7.4 (b) at a first glance, but the experienced eye will notice the decay that occurred after only one application of the median algorithm. The open tips of the lines, as well as the tips of the triangular part disappeared. Although this seems harmless enough, the image will deteriorate after each application of the median, especially as far as thin lines are concerned. Nine-point median sweeps will cause much more damage and the image will decay even faster with repeated application of the algorithm. If images are encountered where thin lines are important to preserve, another smoother should be used, such as the **LULU_9W_1** algorithm, as defined in §4.4.5.

7.5 Noise lines in the original image

Screendump 7.5.1 shows an image of a video still with typical noise originating from video playback. Most of the geometrical borders in the image have noise lines, but the vertical poles are most severely effected. The poles have effectively doubled in thickness due to the shifted pixel lines.

---

[1] A thick noise line is formed by two or more thin noise lines adjacent to each other.

Screendump 7.5.1 **O** **Screendump 7.5.2** **$P_{A9w}(O)$**

The original image, **O**, is firstly tested with the nine-point average filter, but without much visual gain, as shown in screendump 7.5.2. Except for sections of the display, like the hair and ear which seems to improve visually, the noise lines on the poles only became 'blurred' and no acceptable improvement has occurred.

Screendump 7.5.3 **$P_{Lv}(O)$** **Screendump 7.5.4** **$P_{M5w}(O)$**

Screendump 7.5.5 $P_{L5W}(O)$ **Screendump 7.5.6** Pixel line noise.

Screendump 7.5.3 shows the result after a vertical **LULU** algorithm has been applied to remove the horizontal noise. Although the image has been smoothed and all thin noise disappeared, the poles appear to be thicker (due to smearing effects). Screendump 7.5.4 has the results of the five-point median. The outline of the man's facial features improved, but the noise on the poles could not be reduced or improved. The same holds for the five-point **LULU** algorithm, as indicated in screendump 7.5.5.

It seems conclusive that the smaller smoothing windows cannot improve the image due to the nature of the noise lines (screendump 7.5.6). The **MEDIAN_2D/9W** algorithm, screendump 7.5.7, has an interesting effect on the image. Most noise has been removed, except for some single lines remaining on the vertical poles. This method also had the capability of smoothing to the correct pole diameter .

Screendump 7.5.7 $\mathbf{P}_{M9w}(\mathbf{O})$ **Screendump 7.5.8** $\mathbf{P}_{LV}(\mathbf{P}_{M9w}(\mathbf{O}))$

If $\mathbf{P}_{M9w}$ is smoothed again with a vertical **LULU** smoother, the image has no noise lines left (screendump 7.5.8) and the poles are back to the correct dimensions. This example further points out the typical difficulties of working with unknown noise types. It is often difficult to predict which method is 'best' for a specific situation. It might be that other hybrid methods would have obtained the same final result, but which method is optimal? To complicate matters, this question will most probably have to be answered without human interaction during machine processes. This topic is discussed more generally in the next section.

7.6 Practical smoothing in hardware and software

Filtering and smoothing devices are essential in the electronics industry. One-dimensional noise-suppression is an integral part of sound reproduction and amplification and is commonly used in hardware and software applications for speech recognition, speech synthesis and pitch detection [19]. The two-dimensional application has similar utilisation to provide noise-free and crisp digital display. A prospective buyer of video equipment, such as a camera or player, will nowadays inquire about the comparability of

image quality between certain makes, thus indicating the importance of data stability, even to the man in the street. In big organisations the collection of digital data can be very expensive, depending on the application and the sampling methods. The emphasis is therefore often to analyse and *treat* raw data in-house, rather than to pay dearly for data which was enhanced by collecting and distributing agencies. This is a sure way of protecting not only the quality of data, but also the integrity of the original signal.

From the many applications of two-dimensional smoothing, one very important application will be discussed briefly, namely television display. Even though this media form has become one of the most important techniques of communication of modern times, it still suffers from defects such as data corruption. On the best and most expensive digital television display, various distributions of spot noise can still be noticed, depending on the physical geographical position, atmospheric conditions and size of the receiving antenna, to name but a few factors. It is therefore quite logical to ask the following question: can super-fast smoothers, such as a *parallel* ***LULU*** *algorithm*, imbedded in a hardware chip, be used for real-time television noise suppression?

This research has indicated that the **LULU** class of non-linear smoothers are capable of eliminating spot noise, similar to the noise often seen on television screens due to electromagnetic interference, within acceptable bounds. The fact that the smoothers are easily programmable and computationally fast is commendable. The inherent parallelism of the operators make them ideal for acceleration techniques by manipulating parallel programming and VLSI technology. The software application of some **LULU** smoothers has been shown to be stable due to the idempotency feature and they can be successfully combined with other smoothers or filters.

CHAPTER EIGHT

Conclusion

8.1 Results

The development of a new theory for one-dimensional smoothing, using max-min selectors, prompted this research in two-dimensional smoothing. The primary goal was to search for, and test a new class of nearest-neighbour algorithms as they emerged for two-dimensional noise attenuation in practice. Initially there was no guarantee that **LULU** algorithms could be constructed to compete with generally accepted smoothers, such as the widely acclaimed two-dimensional median smoother, but with good faith that the one-dimensional theory could be extended and a lot of endeavour, success was attainable. Although new theories in image enhancement are continually developed for research and academic purposes, the results of such studies only become evident when the theory can be proven explicitely, and, possibly most important, can be illustrated visually. This is what happened with the programming of **LULU** algorithms. In the search for optimal and efficient smoothers, various new two-dimensional non-linear algorithms were designed and tested against noise reduction algorithms. In addition to the five-point and nine-point **LULU** algorithms, as introduced by Rohwer [81], interesting variations were found in the **LULU_2D/HV**, **LULU_2D/9W_1** and other hybrid methods.

The power of idempotency in **LULU** smoothing was pointed out throughout the thesis and the breakdown of some smoothers, when repeated, were shown clearly by practical means. This helped to give insight into some of the most intricate concepts of non-linear mathematical ideas and verified the hierarchy of smoothers in two dimensions. One of the pleasant surprises of designing new two-dimensional smoothers with the

hindsight of a very complete one-dimensional theory, was the simplicity and cost-effectiveness of the new class of smoothers. The speed with which a high-density image can be cleaned of spot noise was appreciated by most people who observed demonstrations of the **LULU** smoothers on personal computers. This was particularly evident where the basic (five-point) **LULU** smoother was compared with the corresponding median smoother. The fact that **LULU_2D/5W** is faster than the **MEDIAN_2D/5W** when applied on a similar input image was a pleasant bonus in the development stages. Another indirect advantage, which only could be appreciated after testing real images, was the crispness of resultant pictures after smoothing. This can be attributed to the robustness and edge preserving qualities of the max-min operators.

Smoothing becomes *optimal* when a non-linear algorithm completely eliminates impulses of high energy (if they are sufficiently separated) without damaging the original image. Some of the **LULU** smoothers in this thesis are candidates for *optimal smoothing algorithms* and have been tested successfully on binary, as well as real images. For different noise distributions, whether it be simulated or from real examples, it was adequate to illustrate how the *strength* of a **LULU** smoother could be increased by using a larger window size, or by concatenating it with another smoother or filter. The practical implications (negative, as well as positive) of hybrid smoothers were shown in chapter seven.

8.2 Future expectations

The promising results attained by applying **LULU** algorithms in a preliminary image processing environment, guarantees interesting research and development to follow. Secondary benefits of this project are the viability of low-cost graphics programming and image processing laboratories for schools, technical institutions and universities. The educational value of a programming task often has very little to do with the subsequent

output of a program. Many textbooks are filled with pages of short routines that are never really integrated into a workable project. Graphics programming offers scope to the entrepreneurial student to learn a considerable amount of programming, while constructing valuable and interesting interfaces between his routines and the end-user. This topic deserves further attention, particularly with the importance of graphics programming in mind for educational, technical and scientific projects. Traditionally, image processing was a topic that could not be managed without the use of high-powered computer hardware and a solid budget. That concept has changed with the availability of inexpensive, high-powered desk-top computers that can handle some of the most complex programming assignments efficiently. The subsequent impact of graphics modules in commercial software is a sign of things to come and should be seriously considered as an integral part of syllabi in information technologies.

The fact that **LULU** smoothing is typical of software that can be imbedded in hardware applications, also exemplifies the practicality of this theory for engineering utilisation. The blending of traditional low-current engineering filter theory with related software demonstrates the logical extension for non-linear applications. Further research is already on the way for speed-up techniques using parallel processing principles for **LULU** algorithms. This will without doubt lead to applications in VLSI implementation, neural networks and related fields for real-time smoothing of high volumes of images.

It is only natural to expect that further successful research areas for the expansion of **LULU** theory are to be found in higher dimension smoothing windows, such as three-dimensional smoothing. Although the complexity of the research domain increases with each expansion of dimension, the robustness of the underlying **LULU** structure is predictable if the results achieved in the first two dimensions are taken into account.

Bibliography

1. Acton, F.S. 1959. **Analysis of Straight-line Data**. New York, Wiley.

2. Almasi, G. & Gotlieb, A. 1994. **Highly Parallel Computing.** Redwood City, Benjamin/Cumming.

3. Angell, I.O. & Tsoubelis, D. 1992. **Advanced Graphics on VGA and XGA Cards Using Borland C++.** London, Macmillan.

4. Ataman, E. et al. 1980. **A Fast Method for Real-Time Median Filtering**. IEEE Trans. on Acoustics, Speech an Signal Processing, 28(4), 415-421.

5. Banerjee, U. et al. 1993. **Automatic Program Parallelization.** IEEE Trans on Parallel and Distributed Systems, 81(2), 211-243.

6. Barron, J. 1995. **Optical-Computing Power Comes to Light.** Byte, 18, 40-41.

7. Baskett, F. & Hennessy, J.L. 1995. **Microprocessors: From Desktops to Supercomputers.** Science, 261, 864-871.

8. Bedford, M. 1995. **Processed Ease.** Computer Shopper, 12(2), 43-49.

9. Bowen, B.A. & Brown, W.R. 1982. **VLSI Systems Design for Digital Signal Processing**. Vol. 1. Englewood Cliffs, Prentice-Hall.

10. Bransky, D. 1995. **Improved DSP.** Electronic Design, 41, 69-70.

11. Burrus, C. et al. 1994 **Computer-based Exercises fir Signal Processing using Mathlab.** Englewood Cliffs, Prentice-Hall.

12. Butz, A.R. 1986. **A Class of Rank Order Smoothers.** IEEE Transactions on Acoustics, Speech and Signal Processing, 34(1), 157-165.

13. Castleman, K.R. 1979. **Digital Image Processing.** London, Prentice-Hall.

14. Cloete, E. 1992. **Parallel Processing.** Baantyd, Overberg Toetsbaan.

15. Cloete, E. 1995. **The Design of a 2D Non-linear Filter for Digital Image Processing**. SAICSIT-95, 75-79.

16. Cloete, E. 1984. **The Solution of Systems of Linear Equations on an MIMD Parallel Processor.** M.Sc. thesis, University of Natal.

17. Cloete, E. & Rohwer, C.H., (to be published). **The Design of Non-linear Smoothers for Image Processing**.

18. Cohen, D. & Kaufman, A. 1995. **Fundamentals of Surface Voxelization.** Graphical Models and Image Processing, 57, 453-461.

19. Creasey, D.J. (Ed.). 1985. **Advanced Signal Processing**. London, Perigrinus.

20. Corriero, N & Gelernter, D. 1992. **How to Write Parallel Programs.** London, MIT Press.

21. Davies, E.R. 1991. **Accurate Filter for Removing Impulse Noise from One- or Two-Dimensional Data.** IEE Proc.- E., 139(2), 111-116.

22. Dudgeon, D.E. & Mersereau, R.M. 1984. **Multidimensional Digital Signal Processing.** Englewood Cliffs, Prentice-Hall.

23. Due, O. & Taxt, T. 1995 **Evaluation of Binarization Methods for Document Images.** IEEE Trans. on Pattern Analysis and Machine Intelligence, 17, 312-315.

24. Do, V. & Barry, A.O. 1993. **A Real-Time Model of the Synchronous Machine Based on Digital Signal processors.** IEEE Trans. on Power Systems, 23, 660-666.

25. Dufaux, F. & Moscheni, F. 1995. **Motion Estimate Techniques for Digital TV : A Review and a New Contribution.** IEEE Trans. on Circuits and Systems, 83, 874-875.

26. Edmonson, W.W. & Alexander, W. 1995. **Transient Suppression at the Boundary for 2-D Digital Systems.** IEEE Trans. on Circuits and Systems, 82, 716-717.

27. Eggers, D.D. & Ackerman, E. 1993. **High Speed Image Rotation in Embedded Systems.** Computer Vision and Image Understanding, 61, 270-277.

28. Fornaro, G. & Franceschetti, G. 1995. **Image Registration in Interferometric SAR Processing.** IEE Proc. on Radar, Sonar and Navigation, 142, 313-320.

29. Fox, G. et al. 1988. **Solving Problems on Concurrent Processors.** Englewood Cliffs, Prentice-Hall.

30. Fuchs, H. et al. 1989. **Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System using Process Enhanced Memories.** Chapel Hill, University of North-Carolina Press, 79-88.

31. Gallagher, N.C. 1981. **A Theoretical Analysis of the Properties of Median Filters.** IEEE Trans. on Acoustics, Speech and Signal Proc., 29(6), 1136-1141.

32. Gallagher, R.S. 1995. **Computer Visualization : Graphics Techniques for Scientific and Engineering Analysis.** New York, CRC Press.

33. Gelenbe, E. 1989. **Multiprocessor Performance.** London, Wiley.

34. Gerald, C.F. & Wheatley, P.O. 1982 **Applied Numerical Analysis.** Reading, Addison-Wesley.

35. Gerogiannis, D. & Orphanoudakis, C. 1993. **Load Balancing Requirements in Parallel Implementations of Image Feature Extraction Tasks.** IEEE Trans. on Parallel and Distributed Systems, 4, 994-1013.

36. Gil, J. & Werman, M. 1993. **Computing 2-D Min, Median and Max Filters.** IEEE Transactions on Pattern Analysis and Machine Intelligence, 15(5), 504-507.

37. Gill, P.E. & Murray, W. 1978. **Algorithms for the Solution of the Non-linear Least-squares Problem.** SIAM J. Number Anal., 15(5), 977-993.

38. Glanz, J. 1995. **Computer Processing gives Imaging a Sharper View.** Science, 269, 1338-1348.

39. Golub, G. 1993. **Scientific Computing : An Introduction with Parallel Programming.** London, Academic Press.

40. Goosen, H., et al. 1994. **Chiron Parallel Program Performance Visualization System.** Computer Aided Design, 26(12), 899-906.

41. Grassmann, W.K. & Tremblay, J. 1996. **Logic and Discrete Mathematics.** Englewood Cliffs, Prentice-Hall.

42. Hall, R.W. 1995. **Optimally Small Operator Supports for Fully Parallel Thinning Algorithms.** IEEE Trans. on Pattern Analysis and Machine Intelligence, 15, 828-833.

43. Hamming, R.W. 1973. **Numerical Methods for Scientists and Engineers.** New York, McGraw-Hill.

44. Hamming, R.W. 1983. **Digital Filters.** Englewood Cliffs, Prentice-Hall.

45. Hearn, D. & Baker, M.P. 1994. **Computer Graphics.** Englewood Cliffs, Prentice-Hall.

46. Hoang, P.D. & Rabaey, Jan M. 1993. **Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput.** IEEE Trans. on Signal Processing, 41(6), 2225-2235.

47. Huntley, J.M. & Goldrein, H.T. and Benckert, L.R. 1993. **Parallel Processing System for Rapid Analysis of Speckle-Photography and Particle-Image-Velocimetry Data.** Applied Optics, 32, 3152-3155.

48. Hultquist, P.F. 1988. **Numerical Methods for Engineers and Computer Scientists.** Reading, Benjamin/Cummings.

49. Hussein, A.O. & Fahmy, M.M. 1991. **Design of 2-D Linear Phase Variable Recursive Digital Filters for Parallel Form Implementation.** IEE Proceedings-G, 138(3), 335-340.

50. Jenkins, T.E. 1987. **Optimal Sensing Techniques and Signal Processing.** Englewood Cliffs, Prentice-Hall.

51. Johnsonbaugh, R. & Kalin, M. 1993. **Applications Programming in ANSI C: Second Edition.** New York, Macmillan.

52. Kay, S.M. 1993. **Fundamentals of Statistical Signal Processing: Estimation Theory.** Englewood Cliffs, Prentice-Hall.

53. Kerr, T.H. 1991. **Comments on 'Federated Square Root Filter for Decentralized Parallel Processes'.** IEEE Trans. on Aerospace and Electronic Systems., 27(6), 946-956.

54. Ketcham, D.J. 1976. **Real Time Image Enhancement Technique.** Proceedings SPIE/OSA Conference on Image Processing, Pacific Grove, California, 74, 120-125.

55. Lee, M.E. & Redner, R.A. 1990. **A Note on the Use of Nonlinear Filtering in Computer Graphics.** Amaco Production Company, Tulas Research Centre, 23-29.

56. Lee, C. et al. 1994. **A Parallel Bit-level Maximum-Minimum Selector for Digital and Video Signal Processing.** IEEE Trans. on Circuits and Systems, 41, 693-695.

57. Luse, M. 1993. **Bitmapped Graphics Programming in C++.** New York, Addison-Wesley.

58. Mallows, C.L. 1980. **Some Theory of Nonlinear Smoothers.** The Annals of Statistics, 8(4), 695-715.

59. Maragos, P. & Schafer, R.W. 1990. **Morphological Systems for Multidimensional Signal Processing.** Proc. of IEEE, 78(4), 690-710.

60. Marshall, R. et al. 1990. **Visualisation Methods and Simulation Steering for a 3D Turbulance Model of Lake Erie.** The Ohio Supercomputer Centre, 89-97.

61. Marquardt, D.W. 1963. **An Algorithm for Least Squares Estimation of Nonlinear Parameters.** J. Soc. Indust. Appl. Math., 11(2), 431-441.

62. Marquardt, A.E., Toerien, L.M. & Terblanche, E. 1991. **Applying Nonlinear Smoothers to Remove Impulsive Noise from Experimentally Sample Data.** R&D Journal, 7(1), 15-18.

63. McCormick, B.H. et al. 1987. **Visualization in Scientific Computing.** Computer Graphics, ACM SIGGRAPH.

64. McQuillan, S.E. & McCanny, J.V. 1995. **A Systematic Methodology for the Design of High Performance Recursive Digital Filters.** IEEE Trans. on Comp., 44(8), 971-982.

65. Meinhart, C.D., Prasad, A.K. & Adrian, R.J. 1993. **A Parallel Digital processor System for Particle Image Velocimetry.** Measurement Science & Technology, 4, 619-626.

66. Meyer, F. 1995. **Object-Based Systems do Windows Better.** InTech, 40, 39-41.

67. Miola, A.M. (Ed.). 1990. **Computing Tools for Scientific Problem Solving.** Rome, Academic Press.

68. Murray, C.J. 1993. **Cray Introduces Massively Parallel Computer.** Design News, 48(9), 30-1.

69. Nakagawa, Y. & Rosenfeld, A. 1978. **A Note on the Use of Local min and max Operations in Digital Picture Processing.** IEEE Trans. on Systems, Man, and Cybernetics, 8(8), 632-635.

70. Nakamura, S. 1993. **Applied Numerical Methods in C.** London, Prentice-Hall.

71. Ngo, C. 1996 **Image Resizing and Enhanced Digital Video Compression.** EDN, 41, 145-148.

72. Nobakht, R.A. 1993. **Adaptive Filtering Of Nonlinear Systems with Memory by Quantized Mean Field Annealing.** IEEE Trans. on Signal Proc., 41(2), 913-925.

73. Oliver, D., et al. 1993. **Tricks of the Graphics Guru's.** New-York, SAMS.

74. Papananos, Y. & Anastassiou, D. 1991. **Analysis and VLSI Architecture of a Nonlinear Edge-Preserving Noise Smoothing Filter.** IEE Proceedings-G, 138(4), 433-440.

75. Pournelle, J. 1995 **Digital Models.** Byte, 20, 257-261

76. Pratt, W.K. 1978. **Digital Image Processing**. New York, Wiley.

77. Rabiner, L.R. et al. 1975. **Applications of a Nonlinear Smoothing Algorithm to Speech Processing.** IEEE Trans. on Acoustics, Speech and Signal Processing, 23(6), 552-557.

78. Raghaven, V., et al. 1995. **Automatic Lineament Extraction from Digital Images Using a Segment Tracing and Rotation Transformation Approach.** Computers and Geosciences, 21, 555-591.

79. Raghuramireddy, D. & Ubehauen, R. 1992. **A New Realization for Multiprocessor Implementation of 2-D Denominator-Separable Digital Filters for Real-Time Processing.** IEEE Trans. on Signal Processing, 40(9), 2349-2353.

80. Rohwer, C.H. 1986. **Idempotent One-Sided Approximation of Median Smoothers.** Institute of Maritime Technology, Simon's Town, South-Africa. 151-163.

81. Rohwer, C.H. (To be published). **LULU-operators for Two-dimensional Data.**

82. Rohwer, C.H. & Toerien, L.M. 1991. **Locally Monotone Robust Approximation Sequences.** Journal of Computational and Applied Mathematics, 36, 399-408.

83. Russ, J.C. 1995. **The Image Processing Handbook.** Boca Raton (Florida), CRC Press.

84. Schendel, U. & Schyska, M. 1984. **Parallel Algorithmen in der Nichtlinearen Optimierung.** Freie Universität Berlin, 1-41.

85. Silver, S. 1995 **Steve Silver on: Image Processing.** Computer Design, 34, 137-140.

86. Smith, J.R. 1993 **The Design and Analysis of Parallel Algorithms.** New York, Oxford University Press.

87. Sung, W. et al. 1992. **Multiprocessor Implementation of Digital Filtering Algorithms Using a Parallel Block Processing Method.** IEEE Trans. on Parallel and Distributed Systems, 3(1), 110-120.

88. Truss, J.K. 1991. **Discrete Mathematics for Computer Scientists.** London, Addison-Wesley.

89. Tukey, J. W. 1974. **Non-linear methods for smoothing data**. Cof. Ref., Eascon.

90. Vainio, O., Yin, L. & Neuvo, Y. 1991. **Parallelism in Generalized Median Operators.** From Pixels to Features II, Amsterdam, North-Holland.

91. Vassiliadis, S., Phillips, J. & Blaner, B. 1995. **Interlock Collapsing ALU's.** IEEE Trans. on Computers, 42, 825-839.

92. Wyszecki, G. & Styles, W.S. 1982 **Color Science.** New York, Wiley & Sons.

93. Zadeh, L.A. 1965. **Fuzzy Sets.** Inform. Contr., 8, 338-353.

Appendices

A-1 Image files

All screendumps, images and graphs were converted to PCX format for publication purposes. It is recommended to use a standard image viewer (DOS or Windows) to view the images supplied with the text of this thesis. Although the printing was done with a HP5L laserjet with 600 dpi resolution, much more detail can be observed on the screen than on the hard copy. The images are numbered according to the numbers in the chapters to assist in this process. If an image **SCREENDUMP 4.3.2.1** needs to be looked up, it will be found as **4_3_2_1.PCX** on the disk.

A-2 List of tables and graphical illustrations

A-3 Symbols often used in this thesis

$o_{i,j}$	Element of the original matrix, O [1], at matrix position i,j.
w_i	Sweeping window at scan-line position i.
$w_{i,j}$	Sweeping window at matrix position (i,j).
A(j,k;uv)	Forward Fourier transform kernel.
B(j,k;uv)	Backward Fourier transform kernel.
C	Ceiling operator in 2D.
C(**P**)	Ceiling operator acting on an image **P**.
$\mathbf{C}_5$	Ceiling operator in 2D, using a five-point operatoring window.
$\mathbf{C}_9$	Ceiling operator in 2D, using a nine-point operatoring window.
C*	Special ceiling operator.
D	The set of double-indexed integers.
F	Floor operator in 2D.
F(**P**)	Floor operator acting on an image **P**.
$\mathbf{F}_5$	Floor operator in 2D, using a five-point operatoring window.
$\mathbf{F}_9$	Floor operator in 2D, using a nine-point operatoring window.
F*	Special floor operator.
f(j,k)	Sampled data for a Fourier transform.
I	Identity operator.
L_2	Euclidean norm.
L_∞	Maximum norm.
L	Lower limit, smoothing operator.
M	Median operator.
$\mathbf{M}_5$	The median operator, using a five-point window.
$\mathbf{M}_9$	The median operator, using a nine-point window.
O(n)	Order n, as in speed-up.

[1] Note that the image, **P**, presented by a matrix, is always printed in capitalized bold letters, while the matrix, P, is itself presented in ordinary print.

$\mathbf{O}$	Image of an input matrix, O; *original* matrix.
$\mathbf{P}$	Image of an output matrix, P; output *picture.*
$\mathbf{P}_f$	Picture after a floor operator was applied.
$\mathbf{P}_{f5}$	Picture after a floor operator was applied, using a five-point window..
$\mathbf{P}_c$	Picture after a ceiling operator was applied.
$\mathbf{P}_{c5}$	Picture after a ceiling operator was applied, using a five-point window.
$\mathbf{P}_{A5}$	The resultant picture after a five-point average operator was applied.
$\mathbf{P}_{A9}$	The resultant picture after a nine-point average operator was applied.
$\mathbf{P}_{Lf}$	Same as $\mathbf{P}_f$, the symbol L indicates a **LULU** operator.
$\mathbf{P}_{Lc}$	Same as $\mathbf{P}_c$, the symbol L indicates a **LULU** operator.
$\mathbf{P}_{fc}$	The resulting image after the application of a ceiling sweep[1], followed by a floor sweep.
$\mathbf{P}_{cf}$	The resulting image after the application of a floor sweep, followed by a ceiling sweep.
$\mathbf{P}_v$	Apply **LULU_2D/V** on an image **P**.
$\mathbf{P}_h$	Apply **LULU_2D/H** on an image **P**.
$\mathbf{P}_{vh}$	Apply **LULU_2D/VH** on an image **P**.
$\mathbf{P}_{hv}$	Apply **LULU_2D/HV** on an image **P**.
$\mathbf{P}_{5f}$	Apply a five-point floor algorithm on an image **P**.
$\mathbf{P}_{5c}$	Apply a five-point ceiling algorithm on an image **P**.
$\mathbf{P}_5$	Apply a *complete* five-point **LULU** algorithm, **LULU_2D/5W,** on an image **P**.
$\mathbf{P}_{9f}$	Apply a nine-point floor algorithm on an image **P**.
$\mathbf{P}_{9c}$	Apply a nine-point ceiling algorithm on an image **P**.
$\mathbf{P}_9$	Apply a *complete* nine-point **LULU** algorithm, **LULU_2D/9W,** on an image **P**.
$\mathbf{P}_{9w1}$	Apply a *complete* nine-point **LULU** algorithm, **LULU_2D/9W_1**, on an image **P**.
$\mathbf{P}_{M5}$	Apply a five-point median algorithm on an image **P**.
$\mathbf{P}_{M9}$	Apply a nine-point median algorithm on an image **P**.
$\mathbf{P}_{M5hv}$	Apply a five-point median algorithm after **LULU_2D/HV**.
S	The set of single-indexed integers.
U	Upper limit, smoothing operator.
$\forall$	All elements of a set.
δ	Tolerance interval.
ξ	Error interval.
ε_g	Gaussian noise.
ε_s	Spot noise.
$\mathfrak{I}$	Fourier transform.
$\mathcal{L}_1$	One-dimensional space.

[1] 'Sweep' in the sense of scanline sweeps.

$\mathcal{L}_2$	Two-dimensional space.
$\eta_{i,j}$	Nucleus of sweeping window, at the matrix position (i,j).
Φ	Linear filter operator.
Γ	General purpose smoother notation.
Γ^n	Applying a smoother applied n times.
Λ	A complete **LULU** process.
ζ	Error free function.

A-4 List of commonly used abbreviations

1D	One-dimensional.
2D	Two-dimensional.
3D	Three-dimensional.
ASCII	American Standard Code for Information Interchange.
BET	Best estimate interval.
CAT scan	Computer assisted tomography.
ccd	Charge coupled device.
CLUT	Colour look-up table.
GIS	Geological information system.
DIP	Digital image processing.
dpi	Dots per inch.
IP	Image processing.
LULU	Lower-upper bound algorithms.
LUT	Look-up table.
Mb	Megabyte.
NWES	North, west, east and south directions.
OCR	Optical character recognition.
OXO	A binary diagram consisting of zero's and x's.
papro	Parallel processor unit.
pel	Same as pixel.
pixel	Smallest image element.
PC	Personal computer.
PCX	PC Paintbrush format.
RGB	Red, blue and green, colour display.
START	Segment tracing and rotation transformation.
SVGA	Super video graphics adapter.
TIFF	Tagged information file format.
VGA	Video graphics adapter.
voxel	Volume pixel.

A-5 Listing of lulu.c

In addition to the program sections of chapter six, the complete listing of the **LULU** and **ANALYSIS** set of programs is included for reference purposes. The programs can be compiled using a Borland compiler with the memory model set as LARGE. For the VESA display and the TIFF original images the following libraries are necessary: VSA256BC.LIB and
TIFF_BCL.LIB.

```
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#include <io.h>
#include <stdio.h>
#include <conio.h>
#include <fcntl.h>
#include <math.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <ctype.h>
#include <dir.h>
#include <mem.h>
#include <dos.h>
#include "tiff.h"
#include "vsa.h"
#include "screen.h"

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#define MINVAL(x,y) ((x)<(y)?(x):(y))
#define MAXVAL(x,y) ((x)>(y)?(x):(y))

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// Config/Options input before processing can begin

typedef struct {
  char name[MAXPATH];
  char noise;             // yes / no
  char filter;            // yes / no
```

A-5 Listing of lulu.c

In addition to the program sections of chapter six, the complete listing of the **LULU** and **ANALYSIS** set of programs is included for reference purposes. The programs can be compiled using a Borland compiler with the memory model set as LARGE. For the VESA display and the TIFF original images the following libraries are necessary: VSA256BC.LIB and
TIFF_BCL.LIB.

```
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#include <io.h>
#include <stdio.h>
#include <conio.h>
#include <fcntl.h>
#include <math.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <ctype.h>
#include <dir.h>
#include <mem.h>
#include <dos.h>
#include "tiff.h"
#include "vsa.h"
#include "screen.h"

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#define MINVAL(x,y) ((x)<(y)?(x):(y))
#define MAXVAL(x,y) ((x)>(y)?(x):(y))

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// Config/Options input before processing can begin

ypedef struct {
  char name[MAXPATH];
  char noise;          // yes / no
  char filter;         // yes / no
```

```
  char noise_type;
  unsigned int seed;
  unsigned int intensity;
  char save_before;
  char name_before[MAXPATH];
  char save_after;
  char name_after[MAXPATH];
  int vesa_mode;
} filter_cfg;

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%

extern unsigned _stklen = 15000U;

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%

unsigned char _far TF_Byte_Buf[4096];
unsigned long TF_ImageWidth, TF_ImageLength;
unsigned _far TF_BitsPerSample[3],TF_Num_Ifd;
unsigned TF_ResolutionUnit,TF_SamplesPerPixel;
unsigned TF_PhotometricInterpretation;
unsigned long TF_XResolution_int,TF_XResolution_frac;
unsigned long TF_YResolution_int,TF_YResolution_frac;
unsigned TF_Black,TF_Red,TF_Orange,TF_Yellow,TF_Green;
unsigned TF_Aqua,TF_Blue,TF_Violet,TF_White;
unsigned XResolution, YResolution, XCharResolution, YCharResolution;
unsigned char XCharSize, YCharSize;
unsigned char BitsPerPixel;

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%

void load_tiff( char * filename, unsigned x );
void filter_exit( void );
void image_stats( char * name );

// Image processing functions
void apply_average_5W( void );
void apply_average_9W( void );
void apply_median_5W( void );
void apply_median_9W( void );
void apply_sharpen( void );

void apply_lulu_2dv( void );

void apply_lulu_w5f( void );
void apply_lulu_w5c( void );
void apply_lulu_w9f( void );
void apply_lulu_w9c( void );

void apply_new_9W_floor( void );
```

```
void apply_new_9W_ceil( void );

void copyr2l( void );

void hist_test( void );
void zoom_box( void );

void add_noise( filter_cfg * cfg, int offset );
void get_config( filter_cfg * cfg );

int MIN3( int a, int b, int c );
int MAX3( int a, int b, int c );
int MIN4( int a, int b, int c, int d );
int MAX4( int a, int b, int c, int d );
int MIN8( int n1, int n2, int n3, int n4, int n5, int n6, int n7, int n8);
int MAX8( int n1, int n2, int n3, int n4, int n5, int n6, int n7, int n8);
int MEDIAN5( int a, int b, int c, int d, int e );
int MEDIAN9( int a, int b, int c, int d, int e, int f, int g, int h, int i );

void save_config( filter_cfg * fcfg );
void load_config( filter_cfg * fcfg );
char exists( char * name );

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

char * version = "v0.1á";
char error_string[160] = "\0";

unsigned char prev_array[1024];
unsigned char curr_array[1024];
unsigned char next_array[1024];
unsigned char new_array[1024];
unsigned char y[1024];
unsigned char u[1024];
unsigned char v[1024];
unsigned char l[1024];

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

int main( int argc, char ** argv )
{
  filter_cfg cfg;
  int error, out_x;
  int i, key;
  char name[20];

  if(!init_text()) {
    puts( "Sorry you must be in 80 column text mode to run FILTER." );
    return 1;
  }
```

```
  text_screen( SAVE );
  atexit( filter_exit );

  setmem( &cfg, sizeof( filter_cfg ), 0 );
  cfg.noise = TRUE;

  load_config( &cfg );
  get_config( &cfg );

  if ( argc == 2 )
    cfg.vesa_mode = atoi( argv[1] );
  else
    cfg.vesa_mode = 0x101;

  error = vsa_init( cfg.vesa_mode );
  if ( error ) {
    sprintf( error_string, "Error initialising VESA mode %Xh\n", cfg.vesa_mode );
    switch ( error ) {
      case 1 : strcat( error_string, "Did You Load Correct VESA Driver (TSR) ?");
           break;
      case 2 : strcat( error_string, "VESA BIOS Extensions (Driver) Not Loaded");
           break;
      case 3 : strcat( error_string, "Requested Video Mode Not Supported by this Card");
           break;
      case 4 : strcat( error_string, "Mode Not an SVGA Mode Supported by this Card");
           break;
      case 5 : strcat( error_string, "VESA Driver Not Returning Mode Information");
           break;
      case 6 : strcat( error_string, "Text I/O Not Supported by your VESA BIOS TSR");
           break;
    }
    return 1;
  }

  strupr( cfg.name );
  load_tiff( cfg.name, 0 );
//  image_stats( cfg.name );
  if (cfg.noise) add_noise( &cfg, 0 );

  switch ( cfg.filter ) {
    case 0 : apply_average_5W();
         break;
    case 1 : apply_new_9W_ceil();
         copyr2l();
         apply_new_9W_floor();
         load_tiff( cfg.name, 0 );
         if (cfg.noise) add_noise( &cfg, 0 );
         break;
    case 2 : apply_median_5W();
         break;
    case 3 : apply_median_9W();
         break;
    case 4 : apply_lulu_w5c();
```

```
        copyr2l();
        apply_lulu_w5f();
        load_tiff( cfg.name, 0 );
        if (cfg.noise) add_noise( &cfg, 0 );
        break;
  case 5 : apply_lulu_w9c();
        copyr2l();
        apply_lulu_w9f();
        load_tiff( cfg.name, 0 );
        if (cfg.noise) add_noise( &cfg, 0 );
        break;
  case 6 : apply_lulu_2dv();
        break;
  case 7 : apply_sharpen();
        break;
  case 8 : hist_test();
        break;
}

key = toupper( getch() );
switch ( key ) {
  case 'S' : for ( i = 0; i < 1000; i++ ) {
             sprintf( name, "DUMP%03d.TIF", i );
             if (! exists( name ) ) break;
           }
           tf_save_file( 0, 0, XResolution-1, YResolution-1, name );
           sound( 1000 ); delay( 70 );
           sound( 400 );  delay( 25 );
           nosound();
           if (!getch()) getch();
           break;
  case 'Z' : zoom_box();
           break;
}

while ( kbhit() ) getch();

if ( cfg.save_before )
  tf_save_file( 0, 0, (unsigned)TF_ImageWidth-1, (unsigned)TF_ImageLength-1, cfg.name_before);

if ( cfg.save_after ) {
  out_x = XResolution >> 1;
  tf_save_file( out_x, 0, out_x+(unsigned)TF_ImageWidth-1, (unsigned)TF_ImageLength-1,
:fg.name_after);
}

save_config( &cfg );

return 0;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
#define WX1 10
#define WY1  6
#define WX2 69
#define WY2 18

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void get_config( filter_cfg * cfg )
{
  int i;
  char ok = FALSE, pos = 0;
  char buffer[160];
  char * heading[] = { { "        INPUT FILE :" },
                 { "   FILTER / PROCESS :" },
                 { "         ADD NOISE :" },
                 { "   NOISE INTENSITY :" },
                 { "        NOISE TYPE :" },
                 { "       RANDOM SEED :" },
                 { "SAVE 'BEFORE' IMAGE :   FILENAME :" },
                 { " SAVE 'AFTER' IMAGE :   FILENAME :" },
                 { "    IS THIS CORRECT :" } };

  char * noise_str[] = { { "WHITE         " },
                 { "BLACK         " },
                 { "WHITE & BLACK" },
                 { "ALL LEVELS    " } };

  char * filt_str[] = { { "AVERAGE 5W " },
                 { "NEW 9W      " },
                 { "MEDIAN 5W   " },
                 { "MEDIAN 9W   " },
                 { "LULU 5W     " },
                 { "LULU 9W     " },
                 { "2D VERTICAL" },
                 { "SHARPEN     " },
                 { "HISTOGRAM   " } };


  char * help_str[] = { { "TIF IMAGE TO LOAD" },
                 { "0=A5W 1=N9W 2=M5W 3=M9W 4=L5W 5=L9W 6=2DV 7=SHARP 8=HIST" },
                 { "Y/N" },
                 { "0..32767" },
                 { "0 = WHITE, 1 = BLACK, 2 = WHITE & BLACK, 3 = ALL LEVELS" },
                 { "0..32767" },
                 { "Y/N" },
                 { "OUTPUT TIF NAME" },
                 { "Y/N" },
                 { "OUTPUT TIF NAME" },
                 { "Y/N" } };
```

```
block_fill( 0,  0, 79,  24, 177, 0x1F );
block_fill( 0,  0, 79,   0,  32, 0x1F );
block_fill( 0, 24, 79,  24,  32, 0x1F );
block_clear( WX1, WY1, WX2, WY2 );
box_draw( WX1, WY1, WX2, WY2, DBL_SIN );

for ( i = 0; i < 8; i++ )
  outstringxy( WX1+3, WY1+1+i, heading[i] );
outstringxy( WX1+3, WY1+2+i, heading[i] );
sprintf( buffer, "LULU %s ³ Non-Linear Filters", version );
outstringxy( 2, 0, buffer );

do {
  for ( pos = 0; pos < 11; pos++ ) {

    sprintf( buffer, "RESPONSE ³ %-65s", help_str[pos] );
    outstringxy( 1, 24, buffer );
    setmem( buffer, 160, 0 );

    switch (pos) {
      case 0 : string( WX1+25, WY1+pos+1, cfg->name, 33 );
               break;

      case 1 : buffer[0] = cfg->filter+48;
               do {
                 string( WX1+25, WY1+pos+1, buffer, 1 );
                 cfg->filter = buffer[0]-48;
               } while (cfg->filter < 0 || cfg->filter > 8);
               outstringxy( WX1+27, WY1+pos+1, filt_str[cfg->filter] );
               break;

      case 2 : if ( cfg->noise ) strcpy( buffer, "Y" );
                 else strcpy( buffer, "N" );
               do {
                 string( WX1+25, WY1+pos+1, buffer, 1 );
               } while ( buffer[0] != 'Y' && buffer[0] != 'N' );
               if ( buffer[0] == 'Y' ) cfg->noise = TRUE;
                 else {
                   cfg->noise = FALSE;
                   pos+=2;
                 }
               break;

      case 3 : itoa( cfg->intensity, buffer, 10 );
               do {
                 string( WX1+25, WY1+pos+1, buffer, 5 );
                 cfg->intensity = atoi( buffer );
               } while ( cfg->seed > 32767 );
               break;

      case 4 : buffer[0] = cfg->noise_type+48;
               do {
                 string( WX1+25, WY1+pos+1, buffer, 1 );
```

```
            cfg->noise_type = buffer[0]-48;
          } while (cfg->noise_type < 0 || cfg->noise_type > 3);
          outstringxy( WX1+27, WY1+pos+1, noise_str[cfg->noise_type] );
          break;

    case 5 : itoa( cfg->seed, buffer, 10 );
          do {
            string( WX1+25, WY1+pos+1, buffer, 5 );
            cfg->seed = atoi( buffer );
          } while ( cfg->seed > 32767 );
          break;

    case 6 : if ( cfg->save_before ) strcpy( buffer, "Y" );
            else strcpy( buffer, "N" );
          do {
            string( WX1+25, WY1+pos+1, buffer, 1 );
          } while ( buffer[0] != 'Y' && buffer[0] != 'N' );
          if ( buffer[0] == 'Y' ) cfg->save_before = TRUE;
            else {
              cfg->save_before = FALSE;
              pos++;
            }
          break;

    case 7 : string( WX1+38, WY1+pos, cfg->name_before, 20 );
          break;

    case 8 : if ( cfg->save_after ) strcpy( buffer, "Y" );
            else strcpy( buffer, "N" );
          do {
            string( WX1+25, WY1+pos, buffer, 1 );
          } while ( buffer[0] != 'Y' && buffer[0] != 'N' );
          if ( buffer[0] == 'Y' ) cfg->save_after = TRUE;
            else {
              cfg->save_after = FALSE;
              pos++;
            }
          break;

    case 9 : string( WX1+38, WY1+pos-1, cfg->name_after, 20 );
          break;

    case 10: buffer[0] = 'N';
          do {
            string( WX1+25, WY1+pos, buffer, 1 );
          } while ( buffer[0] != 'Y' && buffer[0] != 'N' );
          if (buffer[0] == 'Y' ) ok = TRUE;
          break;
   }
  }
} while (!ok);
```

```
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void copyr2l( void )
{
  int y, out_x;

  out_x = XResolution>>1;

  for ( y = 0; y < TF_ImageLength; y++ ) {
    vsa_get_raster_line(out_x, (unsigned)(out_x+TF_ImageWidth-1), y, u );
    vsa_raster_line(0, (unsigned)(TF_ImageWidth-1), y, u );
  }

}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void photo_negative( void )
{
  unsigned char array[1024];
  int x, y, out_x;

  out_x = XResolution >> 1;

  for ( y = 0; y < TF_ImageLength; y++ ) {
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y, array);

    for ( x = 0; x < TF_ImageWidth; x++ )
      array[x] = 255 - array[x];

    vsa_raster_line(out_x, (unsigned)(out_x+TF_ImageWidth-1), y, array);
  }


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

id add_noise( filter_cfg * cfg, int offset )

 nsigned int i, max;
 nax = cfg->intensity;

 rand( cfg->seed );
 witch ( cfg->noise_type ) {
  case 0 : vsa_set_color( TF_White );
        break;
  case 1 : vsa_set_color( TF_Black );
        break;
  case 2 : vsa_set_color( TF_White );
        max >>= 1;
```

```
        for ( i = 0; i < max; i++ )
          vsa_set_pixel( (unsigned) offset+(rand() % (TF_ImageWidth-1)), (unsigned) (rand() %
(TF_ImageLength-1)) );
        vsa_set_color( TF_Black );
        break;
   case 3 : for ( i = 0; i < max; i++ ) {
          vsa_set_color( rand() % 255 );
          vsa_set_pixel( (unsigned) offset+(rand() % (TF_ImageWidth-1)), (unsigned) (rand() %
(TF_ImageLength-1)) );
        }
        return;
  }
  for ( i = 0; i < max; i++ )
    vsa_set_pixel( (unsigned) (offset+(rand() % (TF_ImageWidth-1))), (unsigned) (rand() %
(TF_ImageLength-1)) );
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void apply_average_5W( void )
{
  int x, y, out_x, tval;

  out_x = XResolution >> 1;

  for ( y = 1; y < TF_ImageLength-1; y++ ) {
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y-1, prev_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y, curr_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y+1, next_array);

    for ( x = 1; x < TF_ImageWidth-1; x++ ) {
      tval = prev_array[x] + next_array[x];
      tval += curr_array[x-1] + curr_array[x+1] + curr_array[x];
      tval /=5;
      new_array[x] = tval;
    }

    vsa_raster_line(out_x, (unsigned)(out_x+TF_ImageWidth-1), y, new_array);


  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%

               e = nucleas of 9 point window
      abc        A = MAX( a, b, d, e );
      def        C = MAX( b, c, e, f );
      ghi        G = MAX( d, e, g, h );
               I = MAX( e, f, h, i );

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  ;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
void apply_new_9W_ceil( void )
{
  int x, y, out_x, tval;
  int a, b, c, d, e, f, g, h, i;
  int A, C, G, I;

  out_x = XResolution >> 1;

  for ( y = 1; y < TF_ImageLength-1; y++ ) {
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y-1, prev_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y, curr_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y+1, next_array);

    for ( x = 1; x < TF_ImageWidth-1; x++ ) {
     A = MIN4( prev_array[x-1], prev_array[x], curr_array[x-1], curr_array[x] );
     C = MIN4( prev_array[x], prev_array[x+1], curr_array[x], curr_array[x+1] );
     G = MIN4( curr_array[x-1], curr_array[x], next_array[x-1], next_array[x] );
     I = MIN4( curr_array[x], curr_array[x+1], next_array[x], next_array[x+1] );
     new_array[x] = MAX4( A, C, G, I );
    }

    vsa_raster_line(out_x, (unsigned)(out_x+TF_ImageWidth-1), y, new_array);
  }
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void apply_new_9W_floor( void )
{
  int x, y, out_x, tval;
  int a, b, c, d, e, f, g, h, i;
  int A, C, G, I;

  out_x = XResolution >> 1;

  for ( y = 1; y < TF_ImageLength-1; y++ ) {
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y-1, prev_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y, curr_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y+1, next_array);

    for ( x = 1; x < TF_ImageWidth-1; x++ ) {
     A = MAX4( prev_array[x-1], prev_array[x], curr_array[x-1], curr_array[x] );
     C = MAX4( prev_array[x], prev_array[x+1], curr_array[x], curr_array[x+1] );
     G = MAX4( curr_array[x-1], curr_array[x], next_array[x-1], next_array[x] );
     I = MAX4( curr_array[x], curr_array[x+1], next_array[x], next_array[x+1] );
     new_array[x] = MIN4( A, C, G, I );
    }

    vsa_raster_line(out_x, (unsigned)(out_x+TF_ImageWidth-1), y, new_array);
  }
```

```
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%

void apply_average_9W( void )
{
  int x, y, out_x, tval;

  out_x = XResolution >> 1;

  for ( y = 1; y < TF_ImageLength-1; y++ ) {
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y-1, prev_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y, curr_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y+1, next_array);

    for ( x = 1; x < TF_ImageWidth-1; x++ ) {
      tval = prev_array[x] + prev_array[x-1] + prev_array[x+1];
      tval += next_array[x] + next_array[x-1] + next_array[x+1];
      tval += curr_array[x-1] + curr_array[x+1] + curr_array[x];
      tval /= 9;
      new_array[x] = tval;
    }

    vsa_raster_line(out_x, (unsigned)(out_x+TF_ImageWidth-1), y, new_array);
  }
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%

void apply_sharpen( void )
{
  float IMP_Kernel[9];
  int i,j,m,n,hf_kern,index,width,height,pixel,x1,y1,x2,y2;
  unsigned char array[7168];
  float f_pix;

  IMP_Kernel[0] =  0.0;
  IMP_Kernel[1] = -0.5;
  IMP_Kernel[2] =  0.0;
  IMP_Kernel[3] = -0.5;
  IMP_Kernel[4] =  3.0;
  IMP_Kernel[5] = -0.5;
  IMP_Kernel[6] =  0.0;
  IMP_Kernel[7] = -0.5;
  IMP_Kernel[8] =  0.0;

  x1 = (int)(TF_ImageWidth-1);
  y1 = (int)(TF_ImageLength-1);
  x2 = 10+(unsigned)TF_ImageWidth;
  y2 = 0;

  width = x1+1;
```

```
  height = y1+1;

  hf_kern = 3/2;
  for ( j=0; j < height; j++ ) {
    for ( i = 0; i < 3; i++ ) {
      index = MAXVAL( j+i-hf_kern, 0 );
      index = MINVAL( index, y1 );
      vsa_get_raster_line( 0, x1, index, array+i*width );
    }

    for ( i = 0; i < width; i++ ) {
      f_pix = 0.0;
      for ( n = 0; n < 3; n++ )
        for ( m = 0; m < 3; m++ ) {
          index = MAXVAL( i+m-hf_kern,0 );
              index = MINVAL( index,width-1 );
              index = index +n * width;
              f_pix += ((float)array[index]) * IMP_Kernel[m+n*3];
            }
            f_pix = MAXVAL( f_pix, 0.0 );
            pixel = (unsigned char) MINVAL( f_pix, 255.0 );

            vsa_set_color( pixel );
            vsa_set_pixel( i+x2, j+y2 );
      }
    }
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void apply_lulu_w5f( void )
{
  int x, y, out_x;

  out_x = XResolution >> 1;

  for ( y = 1; y < TF_ImageLength-1; y++ ) {
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y-1, prev_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y, curr_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y+1, next_array);
    memcpy( new_array, curr_array, 1024 );

    for ( x = 1; x < TF_ImageWidth-1; x++ ) {
      new_array[x] = MAX4( MINVAL( curr_array[x-1], curr_array[x] ),
                 MINVAL( curr_array[x+1], curr_array[x] ),
                 MINVAL( prev_array[x], curr_array[x] ),
                 MINVAL( next_array[x], curr_array[x] ) );
    }

    vsa_raster_line(out_x, (unsigned)(out_x+TF_ImageWidth-1), y, new_array);
  }
```

```
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%

void apply_lulu_w5c( void )
{
  int x, y, out_x;

  out_x = XResolution >> 1;

  for ( y = 1; y < TF_ImageLength-1; y++ ) {
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y-1, prev_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y, curr_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y+1, next_array);
    memcpy( new_array, curr_array, 1024 );

    for ( x = 1; x < TF_ImageWidth-1; x++ ) {
      new_array[x] = MIN4( MAXVAL( curr_array[x-1], curr_array[x] ),
                MAXVAL( curr_array[x+1], curr_array[x] ),
                MAXVAL( prev_array[x], curr_array[x] ),
                MAXVAL( next_array[x], curr_array[x] ) );
    }

    vsa_raster_line(out_x, (unsigned)(out_x+TF_ImageWidth-1), y, new_array);
  }

}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%

void apply_lulu_w9f( void )
{
  int x, y, out_x;

  out_x = XResolution >> 1;

  for ( y = 1; y < TF_ImageLength-1; y++ ) {
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y-1, prev_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y, curr_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y+1, next_array);
    memcpy( new_array, curr_array, 1024 );

    for ( x = 1; x < TF_ImageWidth-1; x++ ) {
      new_array[x] = MAX8( MINVAL( curr_array[x-1], curr_array[x] ),
                MINVAL( curr_array[x+1], curr_array[x] ),
                MINVAL( prev_array[x], curr_array[x] ),
                MINVAL( prev_array[x-1], curr_array[x] ),
                MINVAL( prev_array[x+1], curr_array[x] ),
                MINVAL( next_array[x], curr_array[x] ),
                MINVAL( next_array[x-1], curr_array[x] ),
                MINVAL( next_array[x+1], curr_array[x] ) );
```

```
    }

    vsa_raster_line(out_x, (unsigned)(out_x+TF_ImageWidth-1), y, new_array);
  }

}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%

void apply_lulu_w9c( void )
{
  int x, y, out_x;

  out_x = XResolution >> 1;

  for ( y = 1; y < TF_ImageLength-1; y++ ) {
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y-1, prev_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y, curr_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y+1, next_array);
    memcpy( new_array, curr_array, 1024 );

    for ( x = 1; x < TF_ImageWidth-1; x++ ) {
      new_array[x] = MIN8( MAXVAL( curr_array[x-1], curr_array[x] ),
                MAXVAL( curr_array[x+1], curr_array[x] ),
                MAXVAL( prev_array[x], curr_array[x] ),
                MAXVAL( prev_array[x-1], curr_array[x] ),
                MAXVAL( prev_array[x+1], curr_array[x] ),
                MAXVAL( next_array[x], curr_array[x] ),
                MAXVAL( next_array[x-1], curr_array[x] ),
                MAXVAL( next_array[x+1], curr_array[x] ) );
    }

    vsa_raster_line(out_x, (unsigned)(out_x+TF_ImageWidth-1), y, new_array);
  }

}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%

void apply_median_5W( void )
{
  int x, y, out_x;
  int west, east, north, south, nucleus;

  out_x = XResolution >> 1;

  for ( y = 1; y < TF_ImageLength-1-1; y++ ) {
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y-1, prev_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y, curr_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y+1, next_array);
```

```
    for ( x = 1; x < TF_ImageWidth-1; x++ ) {
      west = curr_array[x-1];
      east = curr_array[x+1];
      north = prev_array[x];
      south = next_array[x];
      nucleus = curr_array[x];
      new_array[x] = MEDIAN5( west, east, north, south, nucleus );
    }

    vsa_raster_line(out_x, (unsigned)(out_x+TF_ImageWidth-1)-1, y, new_array);
  }
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void apply_median_9W( void )
{
  int x, y, out_x;

  out_x = XResolution >> 1;

  for ( y = 1; y < TF_ImageLength-1-1; y++ ) {
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y-1, prev_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y, curr_array);
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), y+1, next_array);

    for ( x = 1; x < TF_ImageWidth-1; x++ ) {
      new_array[x] = MEDIAN9( curr_array[x-1], curr_array[x], curr_array[x+1],
                      prev_array[x-1], prev_array[x], prev_array[x+1],
                      next_array[x-1], next_array[x], next_array[x+1] );
    }

    vsa_raster_line(out_x, (unsigned)(out_x+TF_ImageWidth-1)-1, y, new_array);
  }
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void hist_test( void )
{
  unsigned char rline[1024];
  int map[256];
  int x, y, out_x, max;
  double scale_val;
  char buffer[60];

  out_x = XResolution >> 1;
  setmem( map, 256*sizeof( int ), 0 );

  for ( y = 0; y < TF_ImageLength; y++ ) {
    vsa_get_raster_line( 0, (unsigned) (TF_ImageWidth-1), y, rline);
```

```
    for ( x = 0; x < TF_ImageWidth; x++ )
      map[rline[x]]++;
  }

  // Get max val
  for ( x = 0, max = -1 ; x < 256; x++ )
    if ( map[x] > max ) max = map[x];
  scale_val = (double)(YResolution-170)/(double)max;


  vsa_set_color( TF_Black );
  vsa_move_to( out_x, 0 );
  vsa_rect_fill( (unsigned) (out_x+TF_ImageWidth), (unsigned) YResolution );

  y = (unsigned) (TF_ImageLength - (long)((float)map[0]*scale_val) );
  vsa_set_color( TF_White );
  vsa_move_to( out_x, y - 20);
  for ( x = 1; x < 256; x++ ) {
    y = (unsigned) (TF_ImageLength - (long)((float)map[x]*scale_val) );
    vsa_line_to( out_x + x, y - 20 );
  }

  for ( x = 0; x < 256; x++ ) {
    vsa_set_color( x );
    vsa_v_line( out_x + x, (unsigned) TF_ImageLength - 15, (unsigned) ( TF_ImageLength + 15 ) );
  }

  sprintf( buffer, "SCALE FACTOR: %02.05f", scale_val );
  vsa_write_string( 27, out_x/8, TF_White, buffer );
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void apply_lulu_2dv( void )
{
  int max_x, max_y, out_x, count, j;
  unsigned char y[1024];
  unsigned char u[1024];
  unsigned char v[1024];
  unsigned char l[1024];

  out_x = XResolution >> 1;
  max_y = (int) TF_ImageLength;
  max_x = (int) TF_ImageWidth;

  for ( j = 0; j < max_y; j++ ) {
    vsa_get_raster_line(0, (unsigned)(TF_ImageWidth-1), j, y);

    /*
    Calculate the lower bound, l(x)
    for a w=3 for the vector y[]
    */
```

```
    for ( count = 0; count <= max_x; count++ )
      u[count] = MIN3( y[count-1], y[count], y[count+1] );

    for ( count = 0; count <= max_x; count++ )
      v[count] = MAX3( u[count-1], u[count], u[count+1] );

    for ( count = 0; count <= max_x; count++ )
      u[count] = MAX3( v[count-1],v[count],v[count+1] );

    for ( count = 0; count <= max_x; count++ )
      l[count] = MIN3( u[count-1], u[count], u[count+1]);


    /*
    END OF LOWER BOUND
    */

    /*
    3. Calculate the upper bound, v[]
    for a w=3 for the vector y[]
    */

    for ( count = 0; count <= max_x; count++ )
      u[count] = MAX3( y[count-1], y[count], y[count+1] );

    for ( count = 0; count <= max_x; count++ )
      v[count] = MIN3( u[count-1], u[count], u[count+1] );

    for ( count = 0; count <= max_x; count++ )
      u[count] = MIN3( v[count-1], v[count], v[count+1] );

    for ( count = 0; count <= max_x; count++ )
      v[count] = MAX3( u[count-1], u[count], u[count+1] );

    /*
    END OF UPPER BOUND
    */

    /*
    5. Compute the mean m(i)=avg{l(i),v(i)}
    */

    for ( count = 0; count <= max_x; count++ )
      y[count] = ( l[count] + v[count] ) >> 1;
    vsa_raster_line( out_x, out_x+max_x, j, y );
  }
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void zoom_box( void )
{
  char quit = FALSE, left = FALSE;
  byte key, inc = 1;
  int x, y, width, height, i, x1, y1, zoom_x, zoom_y;
  int min_x, max_x, min_y, max_y;
  char save[11][11];
  char zsave[101][101];

  min_x = 0; min_y = 0;
  max_x = (int)TF_ImageWidth-1;
  max_y = (int)TF_ImageLength-1;

  x = 0; y = 0;
  height = 10; width = 10;

  zoom_x = (int)((XResolution >> 1) + (TF_ImageWidth >> 1) - 50);
  zoom_y = (int)((TF_ImageLength >> 1) - 50);

  for ( i = 0; i < 101; i++ )
    vsa_get_raster_line( zoom_x, zoom_x+100, zoom_y+i, (byte*)zsave[i] );

  do {
    for ( i = 0; i < 11; i++ )
      vsa_get_raster_line( x, x+width, y+i, (byte*)save[i] );

    for ( y1 = 0; y1 < height; y1++ )
      for ( x1 = 0; x1 < width; x1++ ) {
        vsa_set_color( save[y1][x1] );
        vsa_move_to( zoom_x+x1*10, zoom_y+y1*10 );
        vsa_rect_fill( zoom_x+(x1+1)*10, zoom_y+(y1+1)*10 );
      }

    vsa_move_to( x, y );
    vsa_set_color( TF_White );
    vsa_rect( x+width, y+height );

    key = (byte) getch();

    for ( i = 0; i < 11; i++ )
      vsa_raster_line( x, x+width, y+i, (byte*)save[i] );

    if ( !key ) {
      key = (byte) getch();
      switch ( key ) {
        case UP : y-=inc;
              break;
        case DN : y+=inc;
              break;
        case LT : x-=inc;
```

```
            break;
      case RT : x+=inc;
            break;
    }

    if ( y < min_y ) y = min_y;
    if ( y + height > max_y ) y = max_y - height;
    if ( x < min_x ) x = min_x;
    if ( x + width > max_x ) x = max_x - width;

  } else {
    switch ( toupper(key) ) {
      case 9  : if (left) left = 0;
               else left = 1;
             for ( i = 0; i < 101; i++ )
               vsa_raster_line( zoom_x, zoom_x+100, zoom_y+i, (byte*)zsave[i] );
             switch (left) {
               case 0 : min_x = 0; min_y = 0;
                     max_x = (int)TF_ImageWidth-1;
                     max_y = (int)TF_ImageLength-1;
                     zoom_x = (int)((XResolution >> 1) + (TF_ImageWidth >> 1) - 50);
                     zoom_y = (int)((TF_ImageLength >> 1) - 50);
                     x -= (XResolution >> 1);
                     break;
               case 1 : min_x = (XResolution >> 1); min_y = 0;
                     max_x = (int) (min_x+TF_ImageWidth-1);
                     max_y = (int)TF_ImageLength-1;
                     zoom_x = (int)((TF_ImageWidth >> 1) - 50);
                     zoom_y =(int)( (TF_ImageLength >> 1) - 50);
                     x += (XResolution >> 1);
                     break;
             }
             for ( i = 0; i < 101; i++ )
               vsa_get_raster_line( zoom_x, zoom_x+100, zoom_y+i, (byte*)zsave[i] );
      case '+' :
      case '=' : inc++;
             if ( inc > 10 ) inc = 10;
             break;
      case '-' : inc--;
             if (inc < 1) inc = 1;
             break;
      case 27  :
      case 'Q' :
      case 'X' : quit = TRUE;
             break;
    }
  }

  } while (!quit);
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
void load_tiff( char * filename, unsigned x )
{
  if ( tf_open_file( filename ) == -1 ) {
    sprintf( error_string, "Error loading TIF image %s", filename );
    exit( 255 );
  }

  if ( tf_get_file_info() == 1 ) {
    strcpy( error_string, "Not a valid TIF image" );
    exit( 255 );
  }

  tf_set_defaults();

  if( tf_read_ifd() == 1 ) {
    tf_close_file();
    strcpy( error_string, "This TIF format not supported" );
    exit( 255 );
  }

  tf_display_image( x, 0 );
  tf_set_prime_colors();
  tf_close_file();
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void filter_exit( void )
{
  vsa_set_svga_mode(0x3);
  text_screen( RESTORE );
  printf( "LULU %s\n", version );

  if (error_string[0]) {
    puts( "A fatal error has occurred." );
    puts( error_string );
  }
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void image_stats( char * name )
{
  char text[100];
  char * cm_ptr;
  int bpp, y, x;

  bpp = TF_BitsPerSample[0];
  switch ( TF_PhotometricInterpretation ) {
    case 0 :
```

```
    case 1 : cm_ptr = "Bilevel / Greyscale";
          break;
    case 2 : cm_ptr = "True Color"; bpp = 24;
          break;
    case 3 : cm_ptr = "Palette";
          break;
    default: cm_ptr = "Unknown";
          break;
  }

  y = (int) ((TF_ImageLength >> 2) / YCharSize) + 1;
  x = (int) ((XResolution >> 1) / XCharSize) + 1;

  vsa_write_string( y++, x, TF_White, "TIF INFORMATION");
  sprintf( text, "TIF name    : %s", name );
  vsa_write_string( y++, x, TF_White, text);
  sprintf( text, "Width       : %d", TF_ImageWidth );
  vsa_write_string( y++, x, TF_White, text);
  sprintf( text, "Length      : %d    ", TF_ImageLength );
  vsa_write_string( y++, x, TF_White, text);
  sprintf( text, "Color model : %s", cm_ptr);
  vsa_write_string( y++, x, TF_White, text);
  sprintf( text, "Bits/pixel  : %d", bpp);
  vsa_write_string( y++, x, TF_White, text);
  vsa_write_string( ++y, x, TF_White, "Press a key to process image ..");
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

int MIN3( int a, int b, int c )
{
  if (MINVAL(a,b) < c) return MINVAL(a,b);
  return c;
}

int MAX3( int a, int b, int c )
{
  if (MAXVAL(a,b) > c) return MAXVAL(a,b);
  return c;
}

int MIN4( int a, int b, int c, int d )
{
  if (MINVAL(a,b) < MINVAL(c,d)) return MINVAL(a,b);
  return MINVAL(c,d);
}

int MAX4( int a, int b, int c, int d )
{
  if (MAXVAL(a,b) > MAXVAL(c,d)) return MAXVAL(a,b);
  return MAXVAL(c,d);
}
```

```
int MIN8( int n1, int n2, int n3, int n4, int n5, int n6, int n7, int n8)
{
  int temp1, temp2;

  if (MINVAL(n1,n2) < MINVAL(n3,n4)) temp1 = MINVAL(n1,n2);
    else temp1 = MINVAL(n3,n4);

  if (MINVAL(n5,n6) < MINVAL(n7,n8)) temp2 = MINVAL(n5,n6);
    else temp2 = MINVAL(n7,n8);

  return MINVAL(temp1, temp2);
}

int MAX8( int n1, int n2, int n3, int n4, int n5, int n6, int n7, int n8)
{
  int temp1, temp2;

  if (MAXVAL(n1,n2) > MAXVAL(n3,n4)) temp1 = MAXVAL(n1,n2);
    else temp1 = MAXVAL(n3,n4);

  if (MAXVAL(n5,n6) > MAXVAL(n7,n8)) temp2 = MAXVAL(n5,n6);
    else temp2 = MAXVAL(n7,n8);

  return MAXVAL(temp1, temp2);
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

int MEDIAN5( int a, int b, int c, int d, int e )
{
  int source[5];
  int dest[5];
  int i, j, s, s_ndx;

  source[0] = a; source[1] = b;
  source[2] = c; source[3] = d;
  source[4] = e;

  for ( i = 0; i < 5; i++ ) {

    for (j = 0, s = 999; j < 5; j++ ) {
      if ( source[j] != -1 ) {
        if ( source[j] < s ) {
          s = source[j];
          s_ndx = j;
        }
      }
    }

    dest[i] = s;
    source[s_ndx] = -1;
```

```
  }

  return dest[2]; // the median
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

int MEDIAN9( int a, int b, int c, int d, int e, int f, int g, int h, int i )
{
  int source[9];
  int dest[9];
  int k, j, s, s_ndx;

  source[0] = a; source[1] = b;
  source[2] = c; source[3] = d;
  source[4] = e; source[5] = f;
  source[6] = g; source[7] = h;
  source[8] = i;

  for ( k = 0; k < 9; k++ ) {

    for (j = 0, s = 999; j < 9; j++ ) {
      if ( source[j] != -1 ) {
        if ( source[j] < s ) {
          s = source[j];
          s_ndx = j;
        }
      }
    }

    dest[k] = s;
    source[s_ndx] = -1;
  }

  return dest[4]; // the median
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void save_config( filter_cfg * fcfg )
{
  FILE * stream;

  stream = fopen( "FILTER.CFG", "wb" );
  if ( stream == NULL ) {
    strcpy( error_string, "Error creating FILTER.CFG" );
    exit( 1 );
  }

  fwrite( fcfg, sizeof( filter_cfg ), 1, stream );
```

```
  fclose( stream );
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%


void load_config( filter_cfg * fcfg )
{
  FILE * stream;

  stream = fopen( "FILTER.CFG", "rb" );
  if ( stream == NULL ) return;   // Not there so return

  fread( fcfg, sizeof( filter_cfg ), 1, stream );

  fclose( stream );
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

char exists( char * name )
{
  FILE * stream;
  stream = fopen( name, "rb" );
  if ( stream == NULL ) return FALSE;
  fclose( stream );
  return TRUE;
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A-6 Listing of analysis.c

```
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#include <io.h>
#include <stdio.h>
#include <conio.h>
#include <fcntl.h>
#include <math.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <ctype.h>
#include <dir.h>
```

```
#include <mem.h>
#include <dos.h>
#include "tiff.h"
#include "vsa.h"
#include "screen.h"

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#define MINVAL(x,y) ((x)<(y)?(x):(y))
#define MAXVAL(x,y) ((x)>(y)?(x):(y))

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// Config/Options input before processing can begin

typedef struct {
  char file_1[MAXPATH];
  char file_2[MAXPATH];
  int vesa_mode;
} analysis_cfg;

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

extern unsigned _stklen = 15000U;

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

unsigned char _far TF_Byte_Buf[4096];
unsigned long TF_ImageWidth, TF_ImageLength;
unsigned _far TF_BitsPerSample[3],TF_Num_Ifd;
unsigned TF_ResolutionUnit.TF_SamplesPerPixel;
unsigned TF_PhotometricInterpretation;
unsigned long TF_XResolution_int,TF_XResolution_frac;
unsigned long TF_YResolution_int,TF_YResolution_frac;
unsigned TF_Black,TF_Red,TF_Orange,TF_Yellow,TF_Green;
unsigned TF_Aqua,TF_Blue,TF_Violet,TF_White;
unsigned XResolution, YResolution, XCharResolution, YCharResolution;
unsigned char XCharSize, YCharSize;
unsigned char BitsPerPixel;

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void load_tiff( char * filename, unsigned x );
void analysis_exit( void );
void image_stats( char * name );

void histogram_1( int out_x, long * map );
void histogram_2( int out_x, long * map );
void histogram_2_fs( long * map );
```

```
void get_config( analysis_cfg * cfg );

void save_config( analysis_cfg * fcfg );
void load_config( analysis_cfg * fcfg );
char exists( char * name );

void subtract_histos( void );
void overlay_histos( void );

void clear_screen( void );
void beeeeeep( void );
void display_help( void );

void write_map( char * name, long * map );

void subtract_images( void );

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

char error_string[160] = "\0";
long map_1[256];
long map_2[256];
double scale_val;
char * version = "v0.1";

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

int main( int argc, char ** argv )
{
  analysis_cfg cfg;
  int error;
  byte key, quit = FALSE;
  char buffer[80];
  int i;

  if (!init_text()) {
    puts( "Sorry you must be in 80 column text mode to run ANALYSIS." );
    return 1;
  }

  text_screen( SAVE );
  atexit( analysis_exit );

  setmem( &cfg, sizeof( analysis_cfg ), 0 );

  load_config( &cfg );
  get_config( &cfg );

  if ( argc == 2 )
    cfg.vesa_mode = atoi( argv[1] );
```

```
else
  cfg.vesa_mode = 0x101;

error = vsa_init( cfg.vesa_mode );
if ( error ) {
  sprintf( error_string, "Error initialising VESA mode %Xh\n", cfg.vesa_mode );
  switch ( error ) {
    case 1 : strcat( error_string, "Did You Load Correct VESA Driver (TSR) ?");
          break;
    case 2 : strcat( error_string, "VESA BIOS Extensions (Driver) Not Loaded");
          break;
    case 3 : strcat( error_string, "Requested Video Mode Not Supported by this Card");
          break;
    case 4 : strcat( error_string, "Mode Not an SVGA Mode Supported by this Card");
          break;
    case 5 : strcat( error_string, "VESA Driver Not Returning Mode Information");
          break;
    case 6 : strcat( error_string, "Text I/O Not Supported by your VESA BIOS TSR");
          break;
  }
  return 1;
}

strupr( cfg.file_1 );
strupr( cfg.file_2 );
ungetch( 'H' );

do {
  key = ( byte ) getch();
  if ( !key ) {
    key = ( byte ) getch();     /* Clear KBD buffer */
    switch ( key ) {
      case 120 :
        write_map( cfg.file_1, map_1 );
        beeeeeep();
        break;
      case 121 :
        write_map( cfg.file_2, map_2 );
        beeeeeep();
        break;
      case F1  :
        display_help();
        break;
    }
  } else {
    switch ( key ) {
      case '1' :
        histogram_2_fs( map_1 );
        break;
      case '2' :
        histogram_2_fs( map_2 );
        break;
      case 'd' :
```

```
        case 'D' :
          clear_screen();
          load_tiff( cfg.file_1, 0 );
          load_tiff( cfg.file_2, XResolution>>1 );
          break;
        case 'h' :
        case 'H' :
          clear_screen();
          load_tiff( cfg.file_1, 0 );
          load_tiff( cfg.file_2, XResolution>>1 );
          histogram_2( 0, map_1 );
          histogram_2( XResolution>>1, map_2 );
          break;
        case '=' :
          clear_screen();
          load_tiff( cfg.file_1, 0 );
          load_tiff( cfg.file_2, XResolution>>1 );
          subtract_images();
          break;
        case '-' :
          subtract_histos();
          break;
        case 'o' :
        case 'O' :
          overlay_histos();
          break;
        case 's' :
        case 'S' :
          for ( i = 0; i < 1000; i++ ) {
            sprintf( buffer, "DUMP%03d.TIF", i );
            if ( !exists( buffer ) ) break;
          }
          tf_save_file( 0, 0, XResolution-1, YResolution-1, buffer );
          beeeeeep();
          break;
        case 27  :
          quit = TRUE;
          break;
      }
    }
  } while (!quit);

  save_config( &cfg );

  return 0;
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void subtract_images( void )
{
   char lt_line[1024], rt_line[1024], n_line[1024];
```

```
  int x,y,out_x = XResolution>>1;

  for ( y = 0; y < TF_ImageLength; y++ ) {
    vsa_get_raster_line( 0, TF_ImageWidth-1, (unsigned)y, lt_line );
    vsa_get_raster_line( out_x, out_x+TF_ImageWidth-1, (unsigned)y, rt_line );
    for ( x = 0; x < TF_ImageLength; x++ )
      n_line[x] = rt_line[x]-lt_line[x];
    vsa_raster_line( out_x, out_x+TF_ImageWidth-1, (unsigned)y, n_line );

  }
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void overlay_histos( void )
{
  int x, y;
  long max;
  char buffer[80];

  clear_screen();
  strcpy( buffer, "OVERLAY HISTOGRAMS" );
  vsa_write_string( 0, 1, TF_White, buffer );

  /* GET MAXIMUM VALUE */
  for ( x = 0, max = -1 ; x < 256; x++ ) {
    if ( map_1[x] > max ) max = map_1[x];
    if ( map_2[x] > max ) max = map_2[x];
  }

  scale_val = (double)(YResolution-170)/(double)max;

  /* MAP 1 */
  vsa_set_color( TF_White );
  for ( x = 0; x < 256; x++ ) {
    y = (unsigned) (TF_ImageLength - (long)((float)map_1[x]*scale_val) );
    vsa_move_to( x, (int)(TF_ImageLength-1) );
    vsa_line_to( x, y );
  }

  /* MAP 2 */
  vsa_set_color( 64 );
  for ( x = 0; x < 256; x++ ) {
    y = (unsigned) (TF_ImageLength - (long)((float)map_2[x]*scale_val) );
    vsa_move_to( x, (int)(TF_ImageLength-1) );
    vsa_line_to( x, y );
  }

  /* MAP 2 */
  vsa_set_color( 64 );
  for ( x = 0; x < 256; x-- ) {
    y = (unsigned) (TF_ImageLength - (long)((float)map_2[x]*scale_val) );
```

```
    vsa_move_to( x+(XResolution>>1), (int)(TF_ImageLength-1) );
    vsa_line_to( x+(XResolution>>1), y );
  }

  /* MAP 1 */
  vsa_set_color( TF_White );
  for ( x = 0; x < 256; x++ ) {
    y = (unsigned) (TF_ImageLength - (long)((float)map_1[x]*scale_val) );
    vsa_move_to( x+(XResolution>>1), (int)(TF_ImageLength-1) );
    vsa_line_to( x+(XResolution>>1), y );
  }

  sprintf( buffer, "SCALE FACTOR: %02.05f", scale_val );
  vsa_write_string( 27, 3, TF_White, buffer );
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void subtract_histos( void )
{
  int i, x, y;
  long result[256];
  int out_x = 30;
  long int max, min;
  float scale_min, scale_max;
  char buffer[80];

  clear_screen();

  strcpy( buffer, "SUBTRACT HISTOGRAMS" );
  vsa_write_string( 0, 1, TF_White, buffer );

  vsa_set_color( 64 );
  vsa_move_to( out_x, (YResolution>>1)+1 );
  vsa_line_to( out_x+512, (YResolution>>1)+1 );

  setmem( result, 256*sizeof( int ), 0 );

  /* CALCULATE RESULT */
  max=-999999L;
  min= 999999L;
  for ( i = 0; i < 256; i++ ) {
    result[i] = map_1[i] - map_2[i];
    if ( result[i] > max ) max = result[i];
    if ( result[i] < min ) min = result[i];
  }

  if ( max != 0 )
    scale_max = ((float)YResolution/2.25)/(float)fabs(max);
  else
    scale_max = 0;
```

```
  if ( min != 0 )
    scale_min = ((float)YResolution/2.25)/(float)fabs(min);
  else
    scale_min = 0;

  if ( scale_min < scale_max )
    scale_val = scale_min;
  else
    scale_val = scale_max;

  y = (unsigned) ((YResolution>>1) + (long)(result[0]*scale_val) );
  vsa_set_color( TF_White );
  vsa_move_to( out_x, y );
  for ( x = 1; x < 256; x++ ) {
    y = (unsigned) ((YResolution>>1)+ (long)(result[x]*scale_val) );
    vsa_line_to( out_x + (x<<1), y );
  }

  sprintf( buffer, "SCALE FACTOR: %02.05f", scale_val );
  vsa_write_string( 27, 2, TF_White, buffer );

}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#define WX1 10
#define WY1  9
#define WX2 69
#define WY2 14

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void get_config( analysis_cfg * cfg )
{
  int i;
  char ok = FALSE, pos = 0;
  char buffer[80];
  char * heading[] = { { "       TIFF FILE 1 :" },
                { "       TIFF FILE 2 :" },
                { "    IS THIS CORRECT :" } };

  char * help_str[] = { { "TIF IMAGE 1 TO ANALYZE" },
                { "TIF IMAGE 2 TO ANALYZE" },
                { "Y/N" } };

  block_fill( 0, 0, 79, 24, 177, 0x1F );
  block_fill( 0, 0, 79,  0, 32, 0x1F );
  block_fill( 0, 24, 79, 24, 32, 0x1F );
  block_clear( WX1, WY1, WX2, WY2 );
  box_draw( WX1, WY1, WX2, WY2, DBL_SIN );
```

```
  for ( i = 0; i < 2; i++ )
    outstringxy( WX1+3, WY1+1+i, heading[i] );
  outstringxy( WX1+3, WY1+2+i, heading[i] );
  sprintf( buffer, "ANALYSIS %s ³", version );
  outstringxy( 2, 0, buffer );

  do {
    for ( pos = 0; pos < 3; pos++ ) {
      sprintf( buffer, "RESPONSE ³ %-65s", help_str[pos] );
      outstringxy( 1, 24, buffer );
      setmem( buffer, 80, 0 );
      switch (pos) {

        case 0 : string( WX1+25, WY1+pos+1, cfg->file_1, 33 );
              break;

        case 1 : string( WX1+25, WY1+pos+1, cfg->file_2, 33 );
              break;

        case 2 : buffer[0] = 'N';
              do {
                string( WX1+25, WY1+pos+2, buffer, 1 );
              } while ( buffer[0] != 'Y' && buffer[0] != 'N' );
              if (buffer[0] == 'Y' ) ok = TRUE;
              break;
      }
    }
  } while (!ok);

}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%

void histogram_1( int out_x, long * map )
{
  unsigned char rline[1024];
  int x, y;

  setmem( map, 256*sizeof( int ), 0 );

  for ( y = 0; y < TF_ImageLength; y++ ) {
    vsa_get_raster_line( (unsigned)out_x, (unsigned) (out_x+TF_ImageWidth-1), (unsigned)y, rline );
    for ( x = 0; x < TF_ImageWidth; x++ )
      map[rline[x]]++;
  }

  vsa_set_color( TF_Black );
  vsa_move_to( out_x, 0 );
  vsa_rect_fill( (unsigned) (out_x+TF_ImageWidth), (unsigned) TF_ImageLength );

  y = (unsigned) (TF_ImageLength - (long)(map[0]/8) );
  vsa_set_color( TF_White );
```

```
  vsa_move_to( out_x, y - 20);
  for ( x = 1; x < 256; x++ ) {
    y = (unsigned) (TF_ImageLength - (long)(map[x]/8) );
    vsa_line_to( out_x + x, y - 20 );
  }

  for ( x = 0; x < 256; x++ ) {
    vsa_set_color( x );
    vsa_v_line( out_x + x, (unsigned) TF_ImageLength - 19, (unsigned) ( TF_ImageLength + 19 ) );
  }
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void histogram_2( int out_x, long * map )
{
  unsigned char rline[1024];
  int x, y;
  long max;
  char buffer[80];

  setmem( map, 256*sizeof( int ), 0 );

  sprintf( buffer, "CALCULATING...", scale_val );
  vsa_write_string( 26, out_x/8+12, TF_White, buffer );

  vsa_set_color( 80 );
  vsa_move_to( (int)(out_x+(TF_ImageWidth>>1)-100), (int)TF_ImageLength+30 );
  vsa_rect_fill( (int)(out_x+(TF_ImageWidth>>1)+99), (int)TF_ImageLength+60 );

  scale_val = (double)200.0/TF_ImageLength;
  vsa_set_color( TF_White );
  for ( y = 0; y < TF_ImageLength; y++ ) {
    vsa_get_raster_line( (unsigned)out_x, (unsigned) (out_x+TF_ImageWidth-1), (unsigned)y, rline );
    for ( x = 0; x < TF_ImageWidth; x++ )
      map[rline[x]]++;
    vsa_move_to( (int)(out_x+(TF_ImageWidth>>1)-100)+y*scale_val, (int)TF_ImageLength+30 );
    vsa_line_to( (int)(out_x+(TF_ImageWidth>>1)-100)+y*scale_val, (int)TF_ImageLength+60 );
  }

  /* GET MAXIMUM VALUE */
  for ( x = 0, max = -1 ; x < 256; x++ )
    if ( map[x] > max ) max = map[x];

  scale_val = (double)(YResolution-170)/(double)max;

  vsa_set_color( TF_Black );
  vsa_move_to( out_x, 0 );
  vsa_rect_fill( (unsigned) (out_x+TF_ImageWidth), (unsigned) YResolution );

  y = (unsigned) (TF_ImageLength - (long)((float)map[0]*scale_val) );
  vsa_set_color( TF_White );
```

```
  vsa_move_to( out_x, y - 20);
  for ( x = 1; x < 256; x++ ) {
    y = (unsigned) (TF_ImageLength - (long)((float)map[x]*scale_val) );
    vsa_line_to( out_x + x, y - 20 );
  }

  for ( x = 0; x < 256; x++ ) {
    vsa_set_color( x );
    vsa_v_line( out_x + x, (unsigned) TF_ImageLength - 15, (unsigned) ( TF_ImageLength + 15 ) );
  }

  sprintf( buffer, "SCALE FACTOR: %02.05f", scale_val );
  vsa_write_string( 27, out_x/8, TF_White, buffer );

}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void histogram_2_fs( long * map )
{
  int x, y;
  long max;
  char buffer[80];
  char out_x = 45;

  clear_screen();
  vsa_write_string( 0, 1, TF_White, "FULL SCREEN HISTOGRAM" );

  /* GET MAXIMUM VALUE */
  for ( x = 0, max = -1 ; x < 256; x++ )
    if ( map[x] > max ) max = map[x];

  scale_val = (double)((float)YResolution/1.4)/(double)max;

  y = (unsigned) (YResolution - (long)(map[0]*scale_val) - 80 );
  vsa_set_color( TF_White );
  vsa_move_to( out_x, y );
  for ( x = 1; x < 256; x++ ) {
    y = (unsigned) (YResolution - (long)(map[x]*scale_val) - 80 );
    vsa_line_to( out_x + (x<<1), y );
  }

  vsa_move_to( out_x-2, YResolution-57 );
  vsa_rect( out_x+2+512, YResolution-33 );

  for ( x = 0; x < 256; x++ ) {
    vsa_set_color( x );
    vsa_v_line( out_x + (x<<1), (unsigned) YResolution - 55, (unsigned) YResolution - 35 );
    vsa_v_line( out_x + (x<<1)+1, (unsigned) YResolution - 55, (unsigned) YResolution - 35 );
  }

  sprintf( buffer, "SCALE FACTOR: %02.05f", scale_val );
```

```
  vsa_write_string( 29, 2, TF_White, buffer );
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%

void load_tiff( char * filename, unsigned x )
{
  if ( tf_open_file( filename ) == -1 ) {
    sprintf( error_string, "Error loading TIF image %s", filename );
    exit( 255 );
  }

  if ( tf_get_file_info() == 1 ) {
    strcpy( error_string, "Not a valid TIF image" );
    exit( 255 );
  }

  tf_set_defaults();

  if( tf_read_ifd() == 1 ) {
    tf_close_file();
    strcpy( error_string, "This TIF format not supported" );
    exit( 255 );
  }

  tf_display_image( x, 0 );
  tf_set_prime_colors();
  tf_close_file();
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%

void analysis_exit( void )
{
  vsa_set_svga_mode( 0x3 );
  text_screen( RESTORE );

  if ( error_string[0] ) {
    puts( "A fatal error has occurred." );
    puts( error_string );
  } else
    printf( "Thanks for using ANALYSIS %s", version );
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%

void image_stats( char * name )
{
  char text[100];
  char * cm_ptr;
```

```
  int bpp, y, x;

  bpp = TF_BitsPerSample[0];
  switch ( TF_PhotometricInterpretation ) {
    case 0 :
    case 1 : cm_ptr = "Bilevel / Greyscale";
          break;
    case 2 : cm_ptr = "True Color"; bpp = 24;
          break;
    case 3 : cm_ptr = "Palette";
          break;
    default: cm_ptr = "Unknown";
          break;
  }

  y = (int) ((TF_ImageLength >> 2) / YCharSize) + 1;
  x = (int) ((XResolution >> 1) / XCharSize) + 1;

  vsa_write_string( y++, x, TF_White, "TIF INFORMATION");
  sprintf( text, "TIF name    : %s", name );
  vsa_write_string( y++, x, TF_White, text);
  sprintf( text, "Width       : %d", TF_ImageWidth );
  vsa_write_string( y++, x, TF_White, text);
  sprintf( text, "Length      : %d    ", TF_ImageLength );
  vsa_write_string( y++, x, TF_White, text);
  sprintf( text, "Color model : %s",  cm_ptr);
  vsa_write_string( y++, x, TF_White, text);
  sprintf( text, "Bits/pixel  : %d",  bpp);
  vsa_write_string( y++, x, TF_White, text);
  vsa_write_string( ++y, x, TF_White, "Press a key to analyze image ..");
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


void save_config( analysis_cfg * fcfg )
{
  FILE * stream;

  stream = fopen( "ANALYSIS.CFG", "wb" );
  if ( stream == NULL ) {
    strcpy( error_string, "Error creating ANALYSIS.CFG" );
    exit( 1 );
  }

  fwrite( fcfg, sizeof( analysis_cfg ), 1, stream );

  fclose( stream );
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
void load_config( analysis_cfg * fcfg )
{
  FILE * stream;

  stream = fopen( "ANALYSIS.CFG", "rb" );
  if ( stream == NULL ) return;   // Not there so return

  fread( fcfg, sizeof( analysis_cfg ), 1, stream );

  fclose( stream );
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

char exists( char * name )
{
  FILE * stream;
  stream = fopen( name, "rb" );
  if ( stream == NULL ) return FALSE;
  fclose( stream );
  return TRUE;
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void clear_screen( void )
{
  vsa_set_color( TF_Black );
  vsa_move_to( 0, 0 );
  vsa_rect_fill( (unsigned) (XResolution), (unsigned) YResolution );
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void beeeeeep( void )
{
  sound( 800 );
  delay( 50 );
  sound( 1100 );
  delay( 25 );
  nosound();
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void display_help( void )
{
```

```
  char * help[] = { { " F1  \0Display this help screen" },
                   { "  D \0Display the 2 TIFF images being analysed" },
                   { "  H \0Recalculate and display TIFF image histograms" },
                   { "  O \0Overlay resulting histograms" },
                   { "  - \0Subtract resulting histograms" },
                   { "  = \0Subtract images" },
                   { "  1 \0Display histogram 1 full screen" },
                   { "  2 \0Display histogram 2 full screen" },
                   { "  S \0Save current screen as a TIFF image" },
                   { "ALT 1\0Dump histogram 1 data to a file" },
                   { "ALT 2\0Dump histogram 2 data to a file" },
                   { " ESC \0Exit from ANALYSIS" } };
  char i;

  clear_screen();
  vsa_write_string( 3, 32, TF_White, "HELP SCREEN" );
  for ( i = 0; i < 12; i++ ) {
    vsa_write_string( 5+i, 15, TF_White, help[i] );
    vsa_write_string( 5+i, 23, TF_White, help[i]+6 );
  }

}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void write_map( char * name, long * map )
{
  FILE * stream;
  char fname[80];
  char buffer[80];
  char * ptr;
  int i;

  strcpy( fname, name );
  ptr = strstr( fname, ".TIF" );
  if ( ptr == NULL ) {
    sprintf( error_string, "Invalid filename %s", fname );
    exit( 1 );
  }

  *ptr = 0;
  strcat( fname, ".DAT" );
  stream = fopen( fname, "wt" );
  if ( stream == NULL ) {
    sprintf( error_string, "Error creating file %s", fname );
    exit( 1 );
  }

  for ( i = 0; i < 256; i++ ) {
    sprintf( buffer, "%03d %08lu\n", i, map[i] );
    fwrite( buffer, strlen( buffer ), 1, stream );
  }
```

```
  fclose( stream );
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%
```