# Speckle Reduction in SAR Imagery

Mark Gebhardt

University of Cape Town

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

# Speckle Reduction in SAR Imagery

Mark Gebhardt, B.Sc.(Eng)

30 June 1995

Thesis submitted to the Department of Electrical Engineering of the University of Cape Town in partial fulfilment of the requirements for the Degree of Master of Science in Engineering.

The University of Cape Town has been given
the right to reproduce this thesis in whole

# Acknowledgements

I would like to thank Mike Inggs, my supervisor, for suggesting the thesis topic, for providing valuable support and guidance and for showing me the way ahead.

I would also like to thank the following friends and colleagues for their advice: Jasper Horrell, Norman Ballard, George Tattersfield and all the members of the UCT Radar Remote Sensing Group.

# Synopsis

Synthetic Aperture Radar (SAR) is a popular tool for airborne and spaceborne remote sensing. Inherent to SAR imagery is a type of multiplicative noise known as *speckle*. There are a number of different approaches which may be taken in order to reduce the amount of speckle noise in SAR imagery. One of the approaches is termed *post image formation processing* and this is the main concern of this thesis.

Background theory relevant to the speckle reduction problem is presented. The physical processes which lead to the formation of speckle are investigated in order to understand the nature of speckle noise. Various statistical properties of speckle noise in different types of SAR images are presented. These include Probability Distribution Functions as well as means and standard deviations. Speckle is considered as a multiplicative noise and a general model is discussed. The last section of this chapter deals with the various approaches to speckle reduction.

Chapter three contains a review of the literature pertaining to speckle reduction. Multiple look methods are covered briefly and then the various classes of post image formation processing are reviewed. A number of non-adaptive, adaptive and segmentation-based techniques are reviewed. Other classes of technique which are reviewed include Morphological filtering, Homomorphic processing and Transform domain methods. From this review, insights can be gained as to the advantages and disadvantages of various methods. A number of filtering algorithms which are either promising, or are representative of a class of techniques, are chosen for implementation and analysis.

The chosen filters are implemented and a discussion of their algorithms is

presented. The theory and operation of each of the filters is explained. The filters which are presented are the Mean, Median, Lorentzian, K Nearest Neighbour (KNN), Hirosawa, *Maximum a Posteriori*, Frost and Maximum Homogeneous Region filters. The filters all operate on the principle of a two dimensional window which is shifted across the image, one pixel at a time. The pixels covered by the window are used to determine the new value of the pixel at the centre of the window. For certain of the filters the local mean and standard deviation (the local statistics) are used to modify the filter response in the presence of edges or point targets. Detailed listings of the source code for all of the filters is given in the appendices.

The chosen filters are used to filter three test images, *i.e.* one and four look ESAR images and a simulated four look image. After filtering, both qualitative and quantitative assessments of filter performance are made. In order to measure the trade-off between geometric and radiometric resolutions two quantities are calculated from the filtered and unfiltered images. These quantities are the Equivalent Number of Looks (an indication of radiometric resolution) and an edge measure, which represents the geometric resolution of the image. The local statistics filters (Frost and MAP) are found to produce the best geometric resolution, but only a slight reduction in the amount of speckle. Good edge preservation is also provided by the Median filter. The Mean filter is found to provide the best speckle reduction, but it causes degradation of the geometric resolution. Two filters which achieve a compromise between speckle reduction and edge preservation are the Hirosawa and KNN filters. The point is made that all filters are dependent on the selection of parameters. It is possible to change the performance of the filters by changing the number of iterations, the window size, or other parameters. The results presented in this section are therefore not absolute and merely serve to provide information on the typical performances of different filters.

The choice of filtering algorithm and its parameters is seen to be closely related to the purpose for which the final image will be used. Filters should be chosen according to whether large or small scale features are of interest. The work presented in this thesis provides valuable insights into the potential of

post image formation speckle reduction methods. These algorithms can be used in addition to, or in lieu of, multiple look methods in order to reduce the speckle in SAR images.

Further research into post image formation techniques, as well as multiple frequency and multiple polarization methods, is suggested. This further comparison could provide valuable information about the potential for further reducing speckle in SAR imagery.

# Contents

**Acknowledgements** i

**Synopsis** ii

**Contents** v

**List of Figures** ix

**List of Tables** x

**Glossary** xi

**1 Introduction** 1

**2 Background Theory** 4
- 2.1 Speckle Physics . . . . . . . . . . . . . . . . . . . . . . . 4
- 2.2 Speckle Statistics . . . . . . . . . . . . . . . . . . . . . . 8
  - 2.2.1 First-order Statistics . . . . . . . . . . . . . . . . . . 9
  - 2.2.2 Second-order Statistics . . . . . . . . . . . . . . . . . 10
- 2.3 Modelling Speckle . . . . . . . . . . . . . . . . . . . . . . 11
- 2.4 Speckle Reduction . . . . . . . . . . . . . . . . . . . . . . 12

**3 Review of Despeckling Literature** 16
- 3.1 Multiple Look Processing . . . . . . . . . . . . . . . . . . 17
- 3.2 Post Image Formation Processing . . . . . . . . . . . . . . 19
  - 3.2.1 Basic Spatial Filtering . . . . . . . . . . . . . . . . . 19
  - 3.2.2 Local Statistics (Adaptive) Filtering . . . . . . . . . . 21
  - 3.2.3 Shape Adaptive Filtering . . . . . . . . . . . . . . . . 27

3.2.4 Segmentation Based Filtering . . . . . . . . . . . . . . 29
3.2.5 Morphological Filtering . . . . . . . . . . . . . . . . . . . . 31
3.2.6 Homomorphic Processing . . . . . . . . . . . . . . . . . . . 31
3.2.7 Transform Domain Filtering . . . . . . . . . . . . . . . . . 32
3.2.8 Other Filtering Techniques . . . . . . . . . . . . . . . . . . 33
3.3 Summary of Techniques . . . . . . . . . . . . . . . . . . . . . . 34

**4 Implementation of Filtering Algorithms** **36**
4.1 Mean Filter . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.2 Median Filter . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.3 Lorentzian Filter . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.4 K Nearest Neighbour Filter . . . . . . . . . . . . . . . . . . . 38
4.5 Hirosawa Filter . . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.6 Gamma-Gamma MAP Filter . . . . . . . . . . . . . . . . . . . 39
4.7 Frost Filter . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.8 MHR Filter . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

**5 Filter Evaluation** **45**
5.1 Radiometric Resolution . . . . . . . . . . . . . . . . . . . . . . 47
5.2 Geometric Resolution . . . . . . . . . . . . . . . . . . . . . . . 49
5.3 Summary of Results . . . . . . . . . . . . . . . . . . . . . . . 53

**6 Conclusions** **56**

**7 Recommendations** **58**

**Bibliography** **59**

**Appendices** **64**

**A Results** **65**
A.1 ENL Values . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
A.2 Edge Measure Values . . . . . . . . . . . . . . . . . . . . . . . 65

**B Mean Filter** **67**
B.1 Source Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
B.2 Parameter Investigation . . . . . . . . . . . . . . . . . . . . . 70

B.3 Theoretical Calculations . . . . . . . . . . . . . . . . . . . . . . 70

**C Median Filter** **72**
C.1 Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
C.2 Source Code . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
C.3 Parameter Investigation . . . . . . . . . . . . . . . . . . . . . 75

**D KNN Filter** **76**
D.1 Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
D.2 Source Code . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
D.3 Parameter Investigation . . . . . . . . . . . . . . . . . . . . . 81

**E Lorentzian Filter** **82**
E.1 Source Code . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
E.2 Parameter Investigation . . . . . . . . . . . . . . . . . . . . . 85

**F Hirosawa Filter** **86**
F.1 Source Code . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
F.2 Parameter Investigation . . . . . . . . . . . . . . . . . . . . . 89

**G MAP Filter** **90**
G.1 Source Code . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
G.2 Parameter Investigation . . . . . . . . . . . . . . . . . . . . . 93

**H Frost Filter** **94**
H.1 Source Code . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
H.2 Parameter Investigation . . . . . . . . . . . . . . . . . . . . . 97

**I MHR Filter** **98**
I.1 Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
I.2 Source Code . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
I.3 Parameter Investigation and Results . . . . . . . . . . . . . . . 108

**J Miscellaneous Code** **109**

**K Images** **111**

# List of Figures


2.1 Reflection at a smooth surface . . . . . . . . . . . . . . . . . . . 5
2.2 Vector addition of random-phase radar returns . . . . . . . . . 7
2.3 Multilook division of SAR reference function . . . . . . . . . . 14

4.1 Masks for calculating edge orientation . . . . . . . . . . . . . . 43

5.1 E-SAR test image for evaluation of filters . . . . . . . . . . . . 45
5.2 Simulated SAR image for evaluation of filters . . . . . . . . . . 46
5.3 Edge cut indicating peak and trough pixels . . . . . . . . . . . 51
5.4 Edge measure for filtered 1 look ESAR image . . . . . . . . . 51
5.5 Edge plot of ESAR 4 look image for various filters . . . . . . . 52
5.6 Edge measure for filtered 4 look ESAR image . . . . . . . . . 53
5.7 Edge measure for filtered simulated SAR image . . . . . . . . 54

I.1 Flow chart of the modified MHR algorithm . . . . . . . . . . . 99

K.1 Unfiltered 1 look ESAR image . . . . . . . . . . . . . . . . . . . 111
K.2 Mean filtered 1 look ESAR image . . . . . . . . . . . . . . . . 112
K.3 Median filtered 1 look ESAR image . . . . . . . . . . . . . . . 112
K.4 Lorentzian filtered 1 look ESAR image . . . . . . . . . . . . . 113
K.5 KNN filtered 1 look ESAR image . . . . . . . . . . . . . . . . 113
K.6 Hirosawa filtered 1 look ESAR image . . . . . . . . . . . . . . 114
K.7 MAP filtered 1 look ESAR image . . . . . . . . . . . . . . . . 114
K.8 Frost filtered 1 look ESAR image . . . . . . . . . . . . . . . . 115
K.9 MHR filtered 1 look ESAR image . . . . . . . . . . . . . . . . 115
K.10 Unfiltered 4 look ESAR image . . . . . . . . . . . . . . . . . . 116
K.11 Mean filtered 4 look ESAR image . . . . . . . . . . . . . . . . 116
K.12 Median filtered 4 look ESAR image . . . . . . . . . . . . . . . 117

K.13 Lorentzian filtered 4 look ESAR image . . . . . . . . . . . . . . 117
K.14 KNN filtered 4 look ESAR image . . . . . . . . . . . . . . . . . . 118
K.15 Hirosawa filtered 4 look ESAR image . . . . . . . . . . . . . . . . 118
K.16 MAP filtered 4 look ESAR image . . . . . . . . . . . . . . . . . . 119
K.17 Frost filtered 4 look ESAR image . . . . . . . . . . . . . . . . . . 119
K.18 MHR filtered 4 look ESAR image . . . . . . . . . . . . . . . . . . 120
K.19 Unfiltered simulated 4 look SAR image . . . . . . . . . . . . . . 120
K.20 Mean filtered simulated 4 look SAR image . . . . . . . . . . . . 121
K.21 Median filtered simulated 4 look SAR image . . . . . . . . . . . 121
K.22 Lorentzian filtered simulated 4 look SAR image . . . . . . . . . 122
K.23 KNN filtered simulated 4 look SAR image . . . . . . . . . . . . 122
K.24 Hirosawa filtered simulated 4 look SAR image . . . . . . . . . . 123
K.25 MAP filtered simulated 4 look SAR image . . . . . . . . . . . . 123
K.26 Frost filtered simulated 4 look SAR image . . . . . . . . . . . . 124
K.27 MHR filtered simulated 4 look SAR image . . . . . . . . . . . . 124

# List of Tables


5.1 ENL for different look/filter combinations . . . . . . . . . . . 47

A.1 ENL values for all image/filter combinations . . . . . . . . . . 66

A.2 Edge measure values for all image/filter combinations . . . . . 66

B.1 ENL and edge values from mean filtered image for various parameter values. . . . . . . . . . . . . . . . . . . . . . . . . 70

C.1 ENL and edge values from median filtered image for various parameter values. . . . . . . . . . . . . . . . . . . . . . . . . 75

D.1 ENL and edge values from KNN filtered image for various parameter values. . . . . . . . . . . . . . . . . . . . . . . . . 81

E.1 ENL and edge values from Lorentzian filtered image for various parameter values. . . . . . . . . . . . . . . . . . . . . . . . . 85

F.1 ENL and edge values from Hirosawa filtered image for various parameter values. . . . . . . . . . . . . . . . . . . . . . . . . 89

G.1 ENL and edge values from MAP filtered image for various parameter values. . . . . . . . . . . . . . . . . . . . . . . . . 93

H.1 ENL and edge values from Frost filtered image for various parameter values. . . . . . . . . . . . . . . . . . . . . . . . . 97

I.1 ENL and edge measure results for the MHR filter . . . . . . . 108

# Glossary

**backscatter** reflected radar energy

**CMI** Continuous Mixed Integrator

**coherent** dependent on phase

**DMI** Discrete Mixed Integrator

**ENL** Equivalent Number of Looks

**EPOS** Edge Preserving Optimised Speckle filter

**ERS** European Research Satellite

**ESAR** European Synthetic Aperture Radar campaign

**geometric resolution** the amount of detail contained in an image

**IDF** Image Domain Filtering

**IML** Improved Multiple Look

**KNN** K Nearest Neighbour

**local statistics** the statistics within a small window

**MAP** Maximum a posteriori

**MHR** Maximum Homogeneous Region

**multilook images** multiple look images which make use of excess radar beamwidth

**PDF** Probability distribution function

**radiometric resolution** the amount of speckle in an image

**SAR** Synthetic Aperture Radar

**sinc** $sin(x)/x$ function

**SLAR** Side Looking Airborne Radar

**speckle** multiplicative noise found in coherent imagery

# Chapter 1

# Introduction

In order to gain information about his changing environment, man has developed many ways to observe and measure the environment and any changes which it experiences. To this end, the field of remote sensing has been developed, enabling mankind to monitor its world from airborne or spaceborne platforms.

Imagery obtained from these platforms contains a great deal of information. These images enable man to monitor the size of rain forests, observe urban expansion and detect large-scale geological features. Valuable information can be obtained about all aspects of the world in which mankind lives.

Radar imagery can provide information which is not available from photographic imagery: this includes information about sub-surface structures in arid areas and specific information about crop-types in agricultural scenes. In addition, radar images can be obtained in cloudy conditions and at night. These advantages make radar imagery more versatile than its photographic counterpart.

The two most common types of radar imagery used for remote sensing are *Side Looking Airborne Radar (SLAR)* and *Synthetic Aperture Radar (SAR)*. The imagery examined in this thesis is obtained using a SAR system, due to its superior resolution and potential to provide polarimetric information.

In any kind of coherent imaging process, including SAR imaging, there is an inherent phenomenon called speckle. Speckle appears as 'Salt and Pepper' noise in the final image, causing image degradation and making both human interpretation and mechanical feature extraction difficult. In order to make optimum use of SAR images, steps must be taken to reduce the amount of speckle in the images.

There has been a large amount of work done on developing different speckle reduction techniques and there are many different techniques which have been documented. The objective of the research presented in this dissertation is to review and compare existing algorithms for speckle reduction, and, in so doing, present information which identifies the trends in the field of speckle reduction.

SAR data sets consisting of co-registered multiple frequency or multiple polarization images are not readily available and this limitation has restricted this study to techniques involving single frequencies and polarizations. In addition, the scope of this dissertation includes only post image formation speckle reduction techniques and does not include a comprehensive study of multilook algorithms.

Chapter 2 details the theory of speckle. The physics of speckle formation is explained and the statistical properties of speckle are detailed briefly. Different approaches to the reduction of speckle are also mentioned here.

This dissertation provides a comprehensive review of the various classes of post image formation techniques and identifies certain filters which are representative of their classes and which seem to provide efficient speckle reduction. Many of the reviewed papers compare existing speckle reduction methods, but few present a full quantitative comparison. Quantitative measures of the amount of speckle are quoted but few of the papers present a measure of the resolution degradation. The selected algorithms include the Mean filter and a number of its derivatives, as well as three filters which adapt to the image according to the local statistics.

The algorithms for the selected filters are described in order to give the reader an appreciation for the different approaches which may be taken in developing a filter for speckle reduction. The various algorithms have been implemented in 'C' code and are used to reduce the level of speckle in a number of test images.

A quantitative comparison of the various speckle-reduced images is made. Here, emphasis is given both to the amount of speckle reduction as well as the degree of edge preservation of the various algorithms. In this section of the dissertation, the merits and demerits of the various algorithms and classes of algorithm become clear.

Lastly, conclusions are drawn as to the efficiency of the various algorithms and their suitability for the filtering of different types of SAR images. Recommendations for future research are also made.

# Chapter 2

# Background Theory

When a scene is illuminated with a single frequency (coherent) source of electromagnetic radiation, the reflected radiation is subject to a process known as fading. Fading manifests itself on images as a type of multiplicative noise known as Speckle [13]. Strictly, speckle is not noise, as it is directly dependent on the structure of the imaged surface. However, the information contained in the speckle pattern is useless to most image users and it obstructs the identification of structures in the image by perturbing the useful information. The presence of speckle in an image makes both human and machine interpretation of images difficult. Speckled images have a false texture in homogeneous areas, and point targets and edges are obscured by the presence of speckle.

This chapter explains the basic theory of speckle, starting with the physical processes which cause it. Next, a brief overview of the statistics of speckled images is given. The final two sections are dedicated to the modelling of speckled images and techniques for speckle reduction.

## 2.1 Speckle Physics

Although speckle is present in all coherent imaging systems, the more specific case of speckled Synthetic Aperture Radar (SAR) images will be considered here. In SAR, as in all types of radar, information about a target (in this case, the earth's surface) is obtained by transmitting electromagnetic radia-

tion and then receiving the radiation which is reflected by the surface. The properties of the received radiation vary according to the characteristics of the imaged surface. The reflected radiation (known as backscatter) can vary in amplitude, phase and polarization, depending on the properties of the reflecting surface.

Radar backscatter is influenced by the following characteristics of the reflecting surface:

- orientation of the surface (or part thereof) relative to the antenna
- moisture content
- surface roughness relative to transmitted wavelength

In addition, the calibration of the antenna and the radar receiver are factors which determine the appearance of the final image.

When an electromagnetic wave encounters a surface (represented by a change in dielectric constant, $\epsilon$) a certain amount of reflection takes place. In the case of a perfectly smooth surface, the reflection obeys the laws of geometric optics, and the wave is reflected with the angle of incidence equal to the angle of reflection, as illustrated in Figure 2.1. However, perfectly flat surfaces do

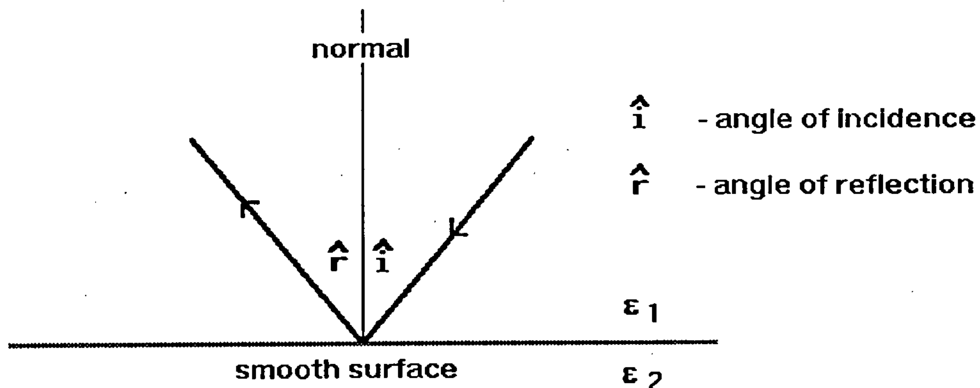


Figure 2.1: Reflection at a smooth surface

not occur in nature, as all surfaces have some degree of roughness [39]. A

surface is considered to be 'rough' if it has undulations or features which have sizes comparable to, or greater than, the radar wavelength. A rough surface causes the electromagnetic waves to be reflected in directions other than that dictated by geometric optics. The rougher the surface, the wider the range of angles through which the radiation is scattered. The above process is known as surface scattering. Another important scattering process is known as volume scattering: this occurs when the electromagnetic radiation penetrates the surface and is reflected from a number of different levels below. Volume scattering is well illustrated by the interaction of electromagnetic radiation with forest areas. In areas with large trees, such as forests, there is reflection of radiation from the topmost leaves (or canopy layer), from branches, trunks and the ground. In a real SAR system, the received radar returns are made up of components from both surface and volume scattering.

In SAR, the smallest image unit is known as a resolution cell. It is important to note that, in the final image, one pixel does not necessarily represent one resolution cell. Image sub-sampling can lead to the pixel spacing being less than the size of a resolution cell. The size of a resolution cell is typically in the range of 3m×3m to 50m×50m. Thus, the dimensions of a resolution cell are large relative to the typical radar wavelength, which is usually tens of centimeters. For each resolution cell there will be a large number of radar returns, each of which can be represented by the vector: $Ae^{j\phi}$. Each radar return has a characteristic magnitude, $A$ and phase, $\phi$. At the radar receiver, all the returns for a specific resolution cell are added vectorially, yielding:

$$v(x,y) = \sum_{i=0}^{n} A_i e^{j\phi_i} \tag{2.1}$$

Where $x$ and $y$ are the azimuth and range coordinates of the resolution cell and $n$ is the number of scatterers within the cell. The amplitude of the individual radar returns is determined by the size, orientation and roughness of the scatterer on the ground. The phase is determined by the two-way path length from the transmitter to the scatterer ($2R$) and is given by:

$$\phi = \frac{4\pi R}{\lambda} - k2\pi \tag{2.2}$$

where $k$ is an integer and $\lambda$ is the radar wavelength. Because the scatterers

within a resolution cell are at different (random) positions, the path length from each scatterer to the transmitter or receiver is different. These differing path lengths cause the returns from different scatterers to have different phases. Considering equation 2.1, it is evident that for high $n$ and random $\phi$, the value of the function is random. The vector addition of a number of radar returns with random phase is indicated in Figure 2.1. Each resolution cell is

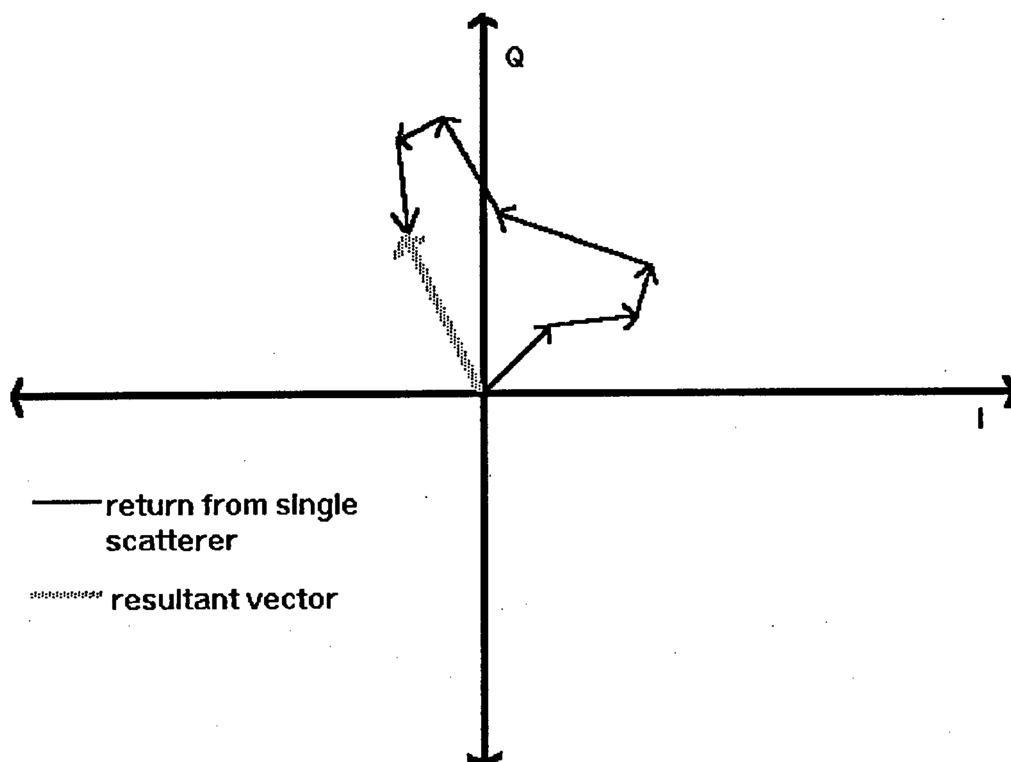


Figure 2.2: Vector addition of random-phase radar returns

represented by its resultant vector as shown in Figure 2.1. Raw SAR data comprises in-phase (I) and quadrature (Q) representations of these resultant vectors. In order to obtain SAR images from the raw data, the amplitude of each IQ element is taken. It is the randomness of the amplitude of each cell's resultant vector which leads to the speckled appearance of the final SAR image.

## 2.2 Speckle Statistics

The formation of speckle can be considered to be a statistical process and this approach can be used to derive statistical models for speckled images. The statistics relating to speckle at a particular point in an image are known as first-order statistics (PDF, mean, variance), while the statistics relating neighbouring pixels are known as second-order statistics (e.g. correlation).

SAR images can take a number of different forms, these are:

- *Intensity images*, which are the product of square-law detection and thus represent the received signal power. After detection, the $N$ look image is generated as follows:

$$P_{ij} = \frac{1}{N}\sum_{k=1}^{N} I_{ijk} \tag{2.3}$$

  where $P_{ij}$ represents the final pixel intensity in row $i$, pixel number $j$. The intensity image is the most common type of SAR image and is used throughout this dissertation.

- *Amplitude images*, which result from a linear detection law and represent the received voltage. Multilook amplitude images are generated as follows:

$$A_{ij} = \frac{1}{N}\sum_{k=1}^{N} [I_{ijk}]^{1/2} \tag{2.4}$$

- Another type of image which is sometimes encountered is the *Square-root Intensity* image. This image is formed by receiving the signal with a square-law detector, resulting in an intensity image and then taking the square-root of each pixel value in order to compress the dynamic range of the image. Multilook square-root intensity images are created according to the following:

$$S_{ij} = \left[\frac{1}{N}\sum_{k=1}^{N} I_{ijk}\right]^{1/2} \tag{2.5}$$

Because of the various detection laws and transfer functions employed, these images have different statistical properties. This section briefly outlines the

image statistics. For a more thorough treatment, see Ulaby *et al* [45] as well as [3, 20, 34, 35, 46].

### 2.2.1 First-order Statistics

The first-order image statistics relate to a particular point in an image and include the Probability Distribution Function (PDF), mean and variance of the image pixels.

- *Intensity Image.* The PDF for a single-look image of an homogeneous scene is a negative exponential distribution, while the PDF of an $N$-look image is simply the convolution of $N$ exponential distributions, the result of which is a *Gamma* distribution [46]. The PDF is given by the following:

$$p(I) = \frac{I^{N-1}\beta^{-N}e^{-I/\beta}}{\Gamma(N)} \tag{2.6}$$

where:

$$\beta = \frac{E[I]}{N} = < I > /N \tag{2.7}$$

$$\Gamma(N) = (N-1)! \tag{2.8}$$

where $I$ represents the image intensity.
This leads us to the multiplicative image model:

$$I = < I > \frac{z}{2N} \tag{2.9}$$

where $z$ represents the fading random variable and is characterized by a normalized chi-square distribution with $2N$ degrees of freedom. The chi-square PDF is as follows:

$$p_z(z) = \frac{z^{N-1}e^{-z/2}}{2^N\Gamma(N)} \tag{2.10}$$

with:

$$E[z] = 2N \tag{2.11}$$

$$var(z) = 4N \tag{2.12}$$

- *Amplitude Image.* For an Amplitude image, the PDF of the speckle in a single look image is *Rayleigh* distributed. The Rayleigh distribution is given as [36]:

$$p(x) = \frac{x}{\sigma^2} e^{-x^2/2\sigma^2}, \qquad x \geq 0 \tag{2.13}$$

with:

$$E[x] = \sqrt{\frac{\pi}{2}} \tag{2.14}$$

$$var(x) = (2 - \frac{\pi}{2})\sigma^2 \tag{2.15}$$

The $N$ look PDF of the speckle is obtained by convolving $N$ *Rayleigh* distributions [46].

- *Square-root Intensity Image.* The PDF of the fading random variable ($f_N$), in an $N$ look square-root intensity image is given by [46]:

$$p(f_N) = \frac{2 f_N^{N-1} N^N \exp^{-N f_N^2}}{(N-1)!}, \qquad f_N \geq 0 \tag{2.16}$$

with:

$$E[f_N] = \frac{\Gamma(N + 1/2)}{N^{1/2}\Gamma(N)} \tag{2.17}$$

$$var(f_N) = 1 - \mu_{f_N}^2 \tag{2.18}$$

where $\mu_{f_N} = E[f_N]$.

### 2.2.2 Second-order Statistics

Second-order image statistics measure the relationship between neighbouring pixels. A common way of measuring this relationship is by using the autocorrelation function (ACF).

SAR images can exhibit a number of different characteristic correlation lengths. In images with large, repeated features, such as agricultural fields, there will be a characteristic correlation length which is approximately the same as the length of the fields. The local mean value will exhibit this feature correlation length and this is an important consideration when selecting how many pixels to average when reducing speckle. The same image may also show other lengths at which there is a high autocorrelation, this can be caused by other

repetitive features such as trees or ocean waves.

SAR images can also have a high correlation between adjacent pixels because of speckle. If the image is sampled with the pixel spacing equal to or greater than the size of a resolution cell, the speckles will be restricted to a single pixel and will not cause correlation between adjacent pixels. Image sampling with pixel spacing less than the size of a resolution cell will cause speckles to span more than one pixel and a correlation length equal to the size of the speckles will be evident in the image. This correlation between adjacent pixels is important for speckle reduction, as it can reduce the effectiveness of simple pixel averaging.

These different scales of correlation length give an indication of the type of information contained in the image. In certain images, the feature correlation length will be greater than the correlation length of the speckle. In these images, it is possible to reduce the speckle while still retaining the majority of the information contained in the image. If the image has a feature correlation length which is approximately two or three pixels, it will be impossible to reduce the speckle appreciably without removing a large proportion of the information contained in the image.

## 2.3 Modelling Speckle

Speckle is often referred to as multiplicative noise. A common model for images containing speckle noise is presented by Frost *et al.* [11]. In this model, the returned radar power, $P_r$ is given by:

$$P_r(x,y) = r(x,y)n(x,y) \tag{2.19}$$

where $(x,y)$ is the imaged position, $r$ represents the unspeckled image and $n$ is the speckle noise. The suitability of this multiplicative model is confirmed by observing that for a given image, within homogeneous regions, the ratio of the standard deviation to the mean is constant. Within purely homogeneous regions, the standard deviation of a noise free image is equal to zero. So, in speckled images, the local standard deviation measured in an homogeneous

area is caused solely by the speckle and represents the speckle noise. Had the speckle noise been additive, the standard deviation would have been constant for all homogeneous areas. The fact that the ratio of standard deviation to mean is constant, is indicative of multiplicative noise.

Oliver [34] shows that, for a 1 look intensity image, the standard deviation is equal to the mean and thus the speckle may be referred to as 100% multiplicative noise.

The multiplicative noise model can be used in conjunction with statistical representations of speckle in order to derive various speckle reduction techniques.

## 2.4 Speckle Reduction

In order to make proper use of SAR images, steps must be taken to reduce the amount of speckle. The different approaches which have been taken when reducing speckle usually use some sort of averaging to combine different images or parts of an image. Because of the random nature of speckle, when a scene is imaged from a slightly different angle or using a different frequency, the speckle pattern is completely different. Two images obtained in such a way have identical *features* and this allows them to be averaged, resulting in a reduction of the amount of speckle.

It is necessary to quantify the amount of speckle in a SAR image in order to compare the effectiveness of different speckle reduction techniques. A useful measure of the amount of speckle is the *equivalent number of looks, (ENL)*, obtained in an homogeneous area, and given by [30]:

$$ENL = <I>^2 / \sigma_I^2 \tag{2.20}$$

where $\sigma_I$ is the local standard deviation of the intensity and $<I>$ is the local mean intensity value. Images characterized by large amounts of speckle will have a low *ENL*, while images with less speckle will have a higher *ENL*. When $N$ independent images of the same scene are averaged, the *ENL* is

increased by a factor of $N$.

Speckle reduction techniques fall into one of the following categories:

- *multiple frequency methods*
- *multiple polarization methods*
- *multiple look methods*
- *post image formation methods*

Techniques which use multiple frequencies require the SAR platform to be equipped with transceivers which operate at different frequencies. For successful speckle reduction the frequencies should be similar, in order to avoid vastly different returns from the same surface. If the radar frequencies are too dissimilar it is possible that, due to resonance or increased penetration, the images may be considerably different. The need for two or more transceivers of similar frequency on a single SAR platform has made this particular approach to speckle reduction impractical and there is therefore not a great deal of data available. The presence of only two distinct frequencies is a severe limitation in that it only allows the ENL to be increased by a factor of two.

Multiple polarization methods require data sets which consist of a number of images of the same surface, obtained using different polarizations. The different polarizations cause the images to have different speckle patterns and allow the speckle to be reduced by averaging a number of these images. The extent to which the speckle can be reduced using this method is limited, as conventional radar systems only produce four different types of polarization, namely horizontal-horizontal (HH), horizontal-vertical (HV), vertical-horizontal (VH) and vertical-vertical (VV). This means that the ENL value can only be increased by a factor of four. In addition, it is possible that the different polarization combinations will produce vastly different local returns because of factors such as scatterer orientation. It is possible that a specific orientation may produce larger returns for one polarization

combination, thereby resulting in a different final image. These image differences also reduce the effectiveness of polarimetric speckle reduction methods.

Multiple look methods make use of excess beamwidth in the SAR antenna (and therefore, excess data) in order to produce a number of independent (or semi-independent) looks. One method used in SAR processing divides the reference function into a number of parts in the spatial frequency domain. Each of these parts is known as a *look*. In practice there is a certain amount of overlap between looks. For example, the reference function could be divided into three looks (see Figure 2.4), with 50 % overlap. In this case, we have two independent (non-overlapping) looks. Each look will produce

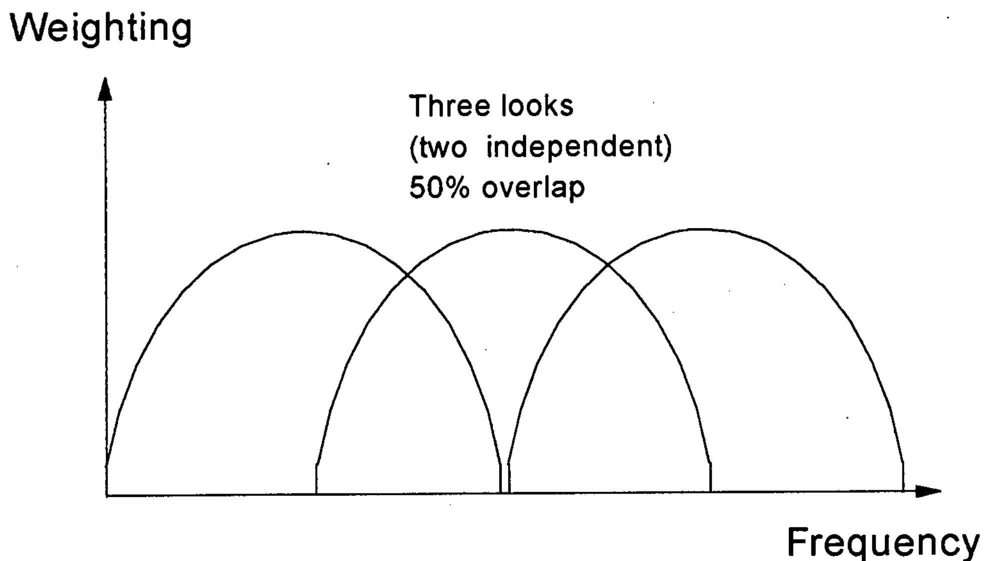


Figure 2.3: Multilook division of SAR reference function

a different version of the imaged scene and each of these versions will have a different speckle pattern. The looks are then averaged non-coherently in order to produce a reduction in the amount of speckle. For $N$ independent looks, the ENL value is increased by a factor of $N$. There is, however, a tradeoff between radiometric resolution (the amount of speckle) and geomet-

ric resolution (the imaged resolution). This tradeoff means that the imaged resolution is reduced by a factor of $N$ for $N$ independent looks if the SAR data is processed to its full bandwidth. A study of speckle in a single image pixel, and how it is affected by multiple look averaging, is presented by Tomiyasu [43].

Post image formation speckle reduction methods are characterized by spatial or frequency domain filtering of the SAR image. Initial attempts at removing the speckle noise (which has a high spatial frequency) by low pass filtering proved unsuccessful, as the high frequency component in edges and point targets was also removed, leading to edge blurring. Most techniques in this category operate in the spatial domain by convolving a filtering window with the SAR image. These filters also seek to achieve an optimal tradeoff between speckle reduction (simple smoothing) and edge preservation. A popular way of achieving this trade-off is by using adaptive filters which are able to adjust to the presence of edges and point targets.

By making use of the physical and statistical properties of speckle, it is possible to design efficient speckle reducing filters which maintain much of the information content of the image. Speckle statistics can be used to identify homogeneous and non-homogeneous areas and therefore can control the amount of speckle reduction which is performed. Factors such as feature correlation length can be used to determine the optimum filter window size. In Chapter 3 a review of literature pertaining to the despeckling of SAR images is presented. Many of the techniques presented make use of insight gained from consideration of the properties of speckle.

# Chapter 3

# Review of Despeckling Literature

This chapter contains a review of existing literature pertaining to speckle reduction. In the first section, a number of papers which propose new or improved multilook methods are reviewed. Following this is a comprehensive review of post image formation techniques which have been broken up into a number of classes. Comparisons are drawn between these algorithms and a number are selected for review. The selected algorithms are not necessarily the most efficient but are chosen as good representatives of a class of techniques, or because they illustrate important basic principles.

This literature review does not include certain other classes of speckle reduction techniques. No attempt has been made to review polarimetric and multiple frequency methods. A lack of readily available multiple polarization and multiple frequency data sets has caused the primary focus of this dissertation to be post image formation techniques.

In addition to the four classes of technique described in Section 2.4, certain other speckle reduction strategies have been developed. An example of an alternative technique is give in a paper by Mancini [26]. This technique involves modification of the SAR antenna in order to reduce the amount of speckle in the final image.

## 3.1 Multiple Look Processing

The basic principle behind multiple look processing of SAR images is the creation of independent images or 'looks' which can be averaged non-coherently. The non-coherent averaging leads to a reduction in speckle, but the generation of independent looks results in a reduction in geometric resolution.

There are many different multiple look techniques which use various methods to generate independent looks, but they all operate on the same underlying principle, that is, a trade-off between geometric and radiometric resolution. In theory, for $N$ independent looks, there will be a reduction in geometric resolution of $1/N$ and the speckle standard deviation will be decreased by a factor of $\sqrt{N}$.

In a paper by Tomiyasu [43], the amount of speckle in a single pixel containing various numbers of scatterers, is investigated. Pixel values are seen to vary by up to 28 dB due to speckle. This pixel variation is seen to be greatly reduced by averaging a number of independent azimuth looks.

Multiple looks are created by dividing up either the azimuth or range frequency spectrum with filters known as 'look' filters. By dividing up the reference function frequency spectrum, performing the multiplication with the frequency domain data, and then taking the inverse FFT and obtaining the image magnitude, a number of different spatial domain versions of the image are produced. These can now be averaged non-coherently, producing a final speckle-reduced image. The looks are overlapped so as to use the bandwidth more efficiently. Forte [9] suggests methods for determining both the optimum amount of overlapping as well as the optimum weighting of each filter. Forte notes that the optimum weighting leads to a 0.125 dB improvement in the radiometric resolution, while optimizing the overlapping yields a 0.33 dB improvement.

Two improved multilook (IML) techniques are proposed by Moreira [30]. The first IML technique involves creating two frequency domain reference functions, each split into looks with different bandwidths. For example, the

first reference function is divided into three 'large' looks, with a 50% overlap. The second reference function is divided into seven 'small' looks, also with 50% overlap. Each reference function is now used to generate a multilook image in the conventional way. In order to achieve the optimum compromise between radiometric and geometric resolution, the 'small' and 'large' looks are weighted and then averaged non-coherently. The second IML technique involves removing the 'small' look overlap in order to improve the equivalent number of looks (ENL). Here, the bandwidth of each look is increased by 20% in order to compensate for the reduced resolution of the small looks. Both IML techniques, when compared with traditional multilook methods, yield improved ENL values for a given resolution.

Scivier and Corr [41] propose a new method for reducing speckle based on the adaptive enhancement of multiple looks. The conventional multilook process is followed, until $N$ independent images are generated. Instead of simply averaging these images non-coherently, the local statistics within a specified window in all $N$ images are used to modify pixel values in the *preferred look*. This method of speckle reduction is claimed to improve the radiometric resolution of the image by the same amount as conventional multiple look processing. Scivier's method should provide better quality images than conventional multilook methods, as it compensates for scene variations from look to look.

Li *et al.* [12] categorize speckle reduction techniques into three broad groups. DMI (Discrete Mixed Integrator) or conventional multi-looking, CMI (Continuous Mixed Integrator) which is multilooking with overlapping segments and IDF (Image Domain Filtering). Comparisons are drawn between DMI, CMI and various IDF techniques using the equivalent number of looks (ENL). Results indicated that better speckle reduction was achieved by using certain IDF techniques than when the DMI or CMI methods were used. No measurement of resolution degradation was made and so a full comparison of algorithm efficiency is not possible from their results.

## 3.2 Post Image Formation Processing

A number of image filtering techniques are reviewed and compared. The aim of this comparison is to identify filters which are able to suppress as much speckle as possible whilst preserving the information content of the original image. This preservation of information content is very important, as it allows both human interpretation and machine feature extraction.

The filters which are reviewed are divided into a number of logical groupings depending on their method of operation. Comparisons are drawn between filters in all groups.

### 3.2.1 Basic Spatial Filtering

The simplest form of spatial filtering is to simply run a small window across the surface of the image, replacing the centre pixel of the window with the mean value of the pixels within the window. This is referred to as a *Box* or *Mean filter*. The Box filter is investigated by Lopes [23] and is observed to produce a final image which is smoothed, with all fine detail removed. This technique is unsatisfactory as it does not maintain edge sharpness or any fine detail.

The *Spatial Averaging Filter* reviewed by Sadjadi [40] is simply a slightly improved version of the Box filter, with different weightings. This filter also leads to a large amount of blur and smearing of fine detail.

A better technique is the *Median filter*. This operates in the same way as the Box filter, except that the centre pixel is replaced with the median of the pixels in the window. Sadjadi [40] found the Median filter to be almost as effective as the spatial averaging filter in removing speckle, and better at maintaining fine detail in the image. Median filtering, however, has the potential to partially or totally reduce the amplitude of point targets.

Masuoka [29] compares the effectiveness of a number of spatial filters at reducing speckle in satellite SAR images. The following filters are compared:

- *Mean filter* as described above.
- *Median filter* as described above.
- *K Nearest Neighbour Average filter.* This filter has a similar form to the Mean filter except that the centre pixel is replaced by the average of a number of neighbouring pixels with intensity values closest to that of the centre pixel.
- *Selective Average Filter.* The centre pixel of the window is only replaced with the average of the neighbouring pixels if there are enough pixels which differ from the centre pixel by a certain threshold value.
- *Spatial Domain Convolution.* In this technique, a window is convolved with the image and the centre pixel of the window is replaced with the sum of the convolved pixels.

Masuoka states that the K Nearest Neighbour (KNN) filter was preferred to the other techniques because some speckle was reduced, while image resolution was maintained. All the other techniques reduced more speckle, but they caused blurred edges and a loss of fine detail. An important conclusion reached by Masuoka was that the success of speckle reduction techniques is dependent on the type of terrain being imaged. Because of this, a filter which performs well on one type of terrain may not be as successful on other surfaces.

In a paper by Abdelhamid *et al.* [1], averaging over $N$ uncorrelated pixels is taken to be equivalent to the averaging of $N$ independent images. In other words, the effect of the Mean filter is taken to be the same as that of multiple look processing. The frequency domain representation of the Mean filter has a $sin(x)/x$ or *sinc* shape. It is suggested that this is not the optimum spectrum, and an improved filter is proposed. This new filter has an impulse response which has a *sinc* shape and is limited with a Hanning window, yielding a spectrum with a flat passband and low sidelobe levels. The results produced by the new filter are an improvement on the Mean filter in that the loss in resolution is the same, but the speckle reduction is greatly enhanced.

A number of variations on the Box filter are implemented by Li *et al.* [12], these are merely Mean filters with various weighting functions applied. The weighting functions are:

- Triangle
- Sinc Squared
- Exponential
- Lorentzian

The equivalent number of looks (ENL) produced by filtering with the Lorentzian weighting function is found to be greater than the other weighting functions for both agricultural and ocean scenes.

The filters described in this section on basic spatial filtering are the Mean and Median filters, with all others simply being variations on the Mean filter. By modifying the weightings applied to the Mean filter, the amount of edge blurring can be decreased. Of the various weighting functions, the Lorentzian function seems to produce the best results. Another promising method is the KNN filter, which reduces edge blurring by only using a specified number of the pixels in the surrounding window which are closest in intensity value to the central pixel. The Mean filter is a useful filter for purposes of comparison as it provides a rough upper bound on the amount of speckle reduction which can be achieved for a given window size.

### 3.2.2 Local Statistics (Adaptive) Filtering

Adaptive filtering techniques attempt to vary the extent of smoothing according to the statistics of the area covered by the filtering window. This adaptivity allows for more smoothing in homogeneous regions, while still maintaining edges, point targets and image texture.

The *Sigma filter* [14, 20, 21, 27] is a well documented filter which works by adapting to the local statistics of the image. The central pixel is thought of as being the mean of a Gaussian distribution and is then replaced with the

average of all surrounding pixels which lie within two standard deviations of the mean.

The *Frost Filter* [11, 14, 23] makes use of a simplified image model and uses a Minimum Mean Squared Error (MMSE) filter in order to estimate the ideal terrain reflectivity from the degraded image. Frost makes both qualitative and quantitative comparisons with Median and Box filters. Due to its ability to adapt to the presence of edges in the filtering window, the Frost filter produces superior results to the filters with which it is compared.

The *Weighting Filter* [14, 27] is a method proposed by Martin and Turner in which pixels within the window ($\mu_i$) are only included in a weighted replacement of the central pixel ($z_c$) if a Gaussian distribution centred on the pixel $\mu_i$ has $z_c$ within its two-sigma bounds. Both Martin and Gordon conclude that, despite slightly inferior speckle reduction, the Weighting filter may be superior to the Sigma Filter due to the fact that it retains more of the intrinsic detail of the image. In addition, Gordon [14] concludes that the Weight filter gives better results than Median, Sigma, Frost and Modified Frost (see below) filters for large scale images.

Kuan *et al.* [19, 23, 49] model speckle according to the coherent process of image formation, instead of assuming a multiplicative model. A linear minimum mean-square error filter is derived, based on the speckle model. This is an adaptive filter which is implemented as a two dimensional recursive algorithm. The Kuan filter takes into account the speckle correlation and this is used in order to produce improved speckle reduction.

Lee [20] proposes a local statistics method of speckle suppression which is very similar to the Kuan filter. This is an adaptive filter which operates without a statistical model for the image and which adapts itself to the statistics of a local 5×5 or 7×7 window. Lee finds the filter to be effective, since it adapts itself to edge areas, thus maintaining the integrity of edges, while still smoothing uniform areas of the image.

Adair and Guindon [2] compare the Frost and Lee Sigma filters. The comparisons are made on a quantitative basis, involving the preservation of existing edges and the generation of spurious ones. The Frost filter is found to have a better performance in both respects and thus is seen to preserve the information content of the original image better.

Lopes *et al.* [23] consider a number of well known filters, including the Box, Lee and Frost Filters. They find the Lee and Frost filtering methods to be based on the local coefficient of variation, which is a measure of image homogeneity. The local coefficient of variation within the filtering window is considered and two thresholds are imposed. Modified forms of the Lee and Frost [14] filters are now developed and compared with the originals. For homogeneous areas, the Modified Frost filter is found to be as effective at suppressing speckle as the Box Filter, while the Modified Lee, although quicker, is slightly less effective. The modified Frost and Lee filters are both found to suppress speckle while still adequately maintaining point targets and edges. In general, the modified filters are found to be better at preserving edges and textural information than their original counterparts. The modified Frost and Lee Filters are found to have very similar performances, with the modified Lee filter being more computationally efficient for smaller windows.

A theoretical analysis of three adaptive filtering methods is provided by Oliver [34] these methods are:

- *Adaptive Linear Despeckling.* This is a Minimum Mean Squared Error (MMSE) method which smooths low contrast areas and preserves high contrast areas. The performance of this filter can be improved by iterating it, leading to a non-linear filtering process.

- *Analytic Bayesian Speckle Reduction.* A non-linear approach is adopted from the start and this leads to a non-linear *maximum a posteriori* (MAP) filter. In this method, the underlying cross-sectional area can be represented by either Gaussian or Gamma-distributed PDFs. The Gamma-distributed PDF is found to preserve bright objects better than the Gaussian distribution.

- *Bayesian Reconstruction by Stochastic Relaxation with Annealing.* This method takes into account the correlations between neighbouring pixels. The success of this method depends on the quality of the prior knowledge used in the filtering process. For real SAR images it is almost impossible to predict, with any accuracy, the speckle-free form of the imaged area.

The best results seem to be produced by the second method, that is, Bayesian MAP reconstruction using a Gamma-distributed PDF. The speckle reduction is not as effective as the last method, but edges and fine detail are better preserved.

An adaptive filter which takes into account lines and edges contained within the filtering window is proposed by Lopes *et al.* [24]. The filter is referred to as the Gamma-Gamma MAP (*Maximum a posteriori*) filter and it takes into account the local coefficient of variation in the filtered window, assuming both the speckle and the imaged surface to have Gamma distributed probability distribution functions. Comparisons are drawn between the following filters:

- *Box filter* as described in the previous subsection
- *Lee filter* as described above
- *Refined Lee filter* which is a version of the Lee filter modified to take into account the local statistics around edges, by means of gradient operators
- *Frost filter* as described above
- *Gamma-Gamma MAP filter*

The Gamma-Gamma MAP filter is found to outperform all other filters in terms of its edge and isolated target preservation. Only the Box filter was found to deliver better speckle reduction, at the cost of greater edge smearing.

An improved version of the Gamma-Gamma MAP filter is proposed by Nezry *et al.* [33]. The local mean value required in the filter algorithm is calculated using the image ACF. This method takes care of the correlation between

adjacent pixels and causes the final image to be closer in appearance to a photographic image than products of the original filter.

A local linear minimum mean square error (LLMMSE) filter is proposed by Harvey and April [16]. This filter works in the intensity domain by comparing a windowed portion of the image with a noise model and then minimizing the mean square error. The LLMMSE filter is found to be more efficient than the Mean filter, in that it is better able to preserve edge information.

An Adaptive Block Kalman Filter (ABKF) is proposed by Azimi-Sadjadi and Bannour [4]. The filter reduces the effect of speckle as well as additive receiver thermal noise and blur. It is shown that this filter can successfully reduce speckle while maintaining edge clarity.

An adaptive filter dependent on the local statistics of an image is proposed by Nathan and Curlander [32], this filter is known as the *Local Adaptive Filter* (LAF). Comparisons were drawn with a number of other speckle reduction techniques, including Lee, Median and Box filters, as well as multilook processing. Use was made of a quantitative measure of speckle reduction(F) consisting of the ratio of equivalent number of looks (ENL) to the resolution degradation (R), as follows:

$$F = \frac{ENL}{R} \tag{3.1}$$

where R is measured by filtering simulated point targets and measuring the width of the filtered and unfiltered point targets in order to quantify the resolution degradation. The LAF method was found to give better results than all others when applied to the test data.

A speckle reduction method for application to one look images is proposed by Hirosawa and Kimura [17]. It is an adaptive method in which the central pixel in a window is changed according to the local statistics of the window. Comparison is made with a three look image with no post image-formation processing, showing the new method to be more effective due to the higher resolution of the final image. The improvement in radiometric resolution is roughly equal to that of a three or four look image, with no apparent loss of

geometric resolution.

A comprehensive comparison of six common speckle reducing filters is drawn by Shi and Fung [42]. The six filters investigated are:

- *The Lee filter*
- *The Kuan filter*
- *The Frost filter*
- *The Enhanced Lee and Frost filters*
- *The Gamma Map filter*

The experimental results show that different filters give better results in different aspects of the filtering process, for example, the Kuan filter is better than the other filters at preserving point targets while still reducing surrounding speckle. It is seen that the values of the filter parameters are important in determining the filter's effectiveness. Overall, the best results seem to be produced by the Frost filter, as well as by the two enhanced filters.

Durand *et al.* [7] compare the effectiveness of ten different spatial domain filters, some adaptive and some not. They conclude that adaptive filters are better suited to speckle reduction as they are able to distinguish between useful information and speckle. The filtering method which is chosen as being the best compromise between maintaining geometric resolution and enhancing radiometric resolution was a variant of the Lee filter [20, 21].

A filter which takes into account the correlation between pixels has been proposed by Lopes and Sery [22]. This is the Vector LMMSE (Linear Minimum Mean Square Error) filter. The filter is adaptive in that it performs less speckle reduction in textured areas, thereby preserving the original texture of the scene. In theory, the filter does not apply in the vicinity of edges or lines, but speckle reduction is also achieved in these regions. The authors do not comment on the ability of the filter to preserve point targets or edges.

A filter which adapts to the presence of an edge within the window area is proposed by Ueno and Hirosawa [44]. If a change in intensity level (ie. an edge) is detected within the window, then the pixels are sorted into two groups, representing the high and low intensity areas. The value of the central pixel is now determined as the mean of the larger group. In this way, the filter compensates for and preserves edge clarity, but no such attempt is made for point targets. Ueno's filter is compared with Mean and Median filters, an adaptive filter and a Wiener filtering (Transform domain) method. The new filter was found to preserve edges relatively well and also to be superior to the other filters in the smoothing of homogeneous areas.

Comparison of various speckle filters is complicated by the fact that certain filters are more efficient than others on different types of terrain. Thus, a filter which performs well on sea surfaces may be outperformed in forested or urban areas. This, coupled with vast differences in performance due to variations in parameter values, makes filter comparison difficult.

Two adaptive filters which seem to produce promising results are the Frost and Gamma-Gamma MAP filters, both of which are typical of the local statistics type of filtering. Another interesting filter is the Hirosawa filter which seems to produce promising results when filtering one look images.

A disadvantage associated with local statistics filtering is the failure to reduce speckle in areas adjacent to edges or strong point targets.

### 3.2.3 Shape Adaptive Filtering

In addition to adapting to the local statistics of an image, it is possible to design filters which include some measure of shape adaptivity. This additional adaptivity allows features such as edges to be better represented within the filtering window. Wu and Maître [49] suggest a generalized filter which exhibits shape adaptivity. This filter is referred to as the Maximum Homogenous Region (MHR) filter. The basic operation of the filter involves starting with a small window and enlarging it for as long as the area within the window is determined to be homogeneous. If an edge is detected, the

window expansion continues by means of a semiwindow which includes the largest homogeneous part of the window. Once an homogeneous area has been identified by the MHR filter, any known adaptive filtering method can be applied in order to reduce the speckle within this homogeneous area. The local statistics smoothing method proposed by Lee [20] is used in the MHR filter, and the results produced by the MHR filter are compared with results from Median, Gaussian and Kuan filtering. The MHR filter was found to have better performance than any of the other filters due to its shape adaptivity characteristic.

A filter which adapts itself to the orientation of edges is proposed by Nagao and Matsuyama [31]. The filter works by calculating the variance of the image under different orientations of a bar-shaped mask. The mask orientation which yields the smallest variance indicates the most homogeneous area. By averaging over the most homogeneous area, edge integrity is preserved (or even enhanced) and efficient smoothing is performed. The authors suggest this as a general smoothing filter and not specifically for SAR images. However, as the smoothing is simply performed by taking the mean, this filter is suitable for removing speckle from SAR images.

An Edge Preserving Optimized Speckle filter (EPOS) is proposed by Hagg and Sties [15]. This is a filtering method whereby the largest region containing no edges is used for averaging. The presence of an edge is detected using the relative standard deviation ($R$) within a region, this is given by:

$$R = \sigma_I / < I > \tag{3.2}$$

Where $\sigma_I$ is the standard deviation and $< I >$ is the mean. The relative standard deviation should remain constant within homogeneous areas but increase when an edge is present. This property is used to identify the largest possible homogeneous area surrounding a pixel. Once this area has been identified, the grey-scale value of the chosen pixel is changed to the average value of the chosen area. When EPOS was applied to an ERS-1 image, the amount of speckle was reduced and edges seemed to be preserved. The performance of EPOS on a synthetically generated image was compared with that of the Sigma filter. The EPOS algorithm produced better speckle re-

duction and also maintained more edge clarity than the Sigma filter.

Shape adaptive filtering is a promising area in speckle reduction, and offers the possibility of better speckle reduction and edge preservation than normal local statistics methods. Shape adaptivity allows filters to accurately maintain edge clarity and position which is important for image interpretation. However, these advantages are offset by increased computation time due to window sizing and shaping. A filter which seems to be representative of this class of algorithms is the MHR filter proposed by Wu and Maître. It should be noted that accurate adaption of window shape is made difficult because of the image speckle, which presents itself as false edges.

### 3.2.4 Segmentation Based Filtering

Spatial filtering techniques are inherently limited in that they require a large window size for optimum filtering, and at the same time need to keep the window size small enough to only cover homogeneous areas. This problem seems to suggest that some sort of image segmentation prior to filtering would produce the best results. Successful image segmentation, however, requires a high degree of prior knowledge which is usually not available.

White [48] proposes an algorithm which performs radar cross-section estimation on SAR images, using a technique called simulated annealing. Conventional annealing techniques attempt to categorize each pixel as belonging to one of a fixed number of states. In order to avoid the restrictions imposed by forcing the pixels into a fixed number of states, an edge detection stage is included. This allows the algorithm to have a real-valued output. The annealing algorithm is equivalent to adaptive filtering and results indicate that speckle is significantly reduced, while a high degree of edge clarity is maintained.

Ellis *et al.* [8] suggest that neural networks should be used in order to provide the capability to learn the segmentation task by example. A neural network was trained to reduce speckle using the results of White's simulated annealing technique [48]. The results produced by the neural network approach

are close to those produced by the simulated annealing technique. It is suggested that good results should be produced if the neural network is trained by using single and multiple look images.

Another segmentation based approach to speckle reduction is proposed by Mascarenhas and Frery [28]. The image segmentation is achieved through the use of the Iterated Conditional Modes (ICM) algorithm, which is an iterative technique for parameter estimation. The ICM algorithm is shown to be effective at noise reduction and to perform better than the Nagao-Matsuyama algorithm [31].

A Structure Detection Filter (SDF) originally presented by Nezry has been modified by Baraldi and Parmiggiani [5]. The modified SDF adapts to both local statistics and geometric properties of an image. The shape and orientation of the adaptive window is chosen using a statistical edge detector and a thin linear structure detector. Segmentation is performed using edges detected in the image and a statistically adaptive filter (such as the Gamma-Gamma MAP filter) is applied to the segmented image. Experimental results show that the modified SDF reduces speckle and is better at retaining fine image detail than a plain adaptive filter with no image segmentation step.

Image segmentation, when used as a precursor to image filtering, allows more efficient and more accurate filtering of speckled images. Effective segmentation allows for better speckle reduction in the resulting homogeneous areas and also contributes to maintaining the integrity of edges and of the areas which they separate. The presence of speckle, however, makes successful image segmentation a difficult task as the speckle obscures real edges and can cause the detection algorithm to find false edges. These two factors can lead to an image being incorrectly segmented.

Image segmentation can be very time-consuming and often requires a high degree of prior knowledge about the imaged scene. It is possible to use images from optical sensors in order to segment SAR images[1]. This method requires

[1]see: Schoenmakers*et al*, "Results of a Hybrid Segmentation Method", Image and Sig-

SAR and optical coverage of the same region and overlays the segmentation information from the optical image on the SAR image. The lack of availability of prior knowledge is limitation in the image segmentation approach to speckle reduction.

### 3.2.5 Morphological Filtering

The basic binary morphological operators are known as *dilation* and *erosion*. The action of these operators is illustrated by running a structuring element around the inside (for erosion) or outside (for dilation) of an image feature. The size of the feature is now increased or decreased by including or rejecting the pixels covered by the structuring element. These binary operators can also be extended to the grey-scale case, where differing grey-scale values are used instead of binary values.

From these two basic operators, further operators can be derived, for example, *opening* (erosion followed by dilation) and *closing* (dilation followed by erosion). Kher and Mitra [18] derive new operators and develop processes known as bounded opening and closing. They also propose a sampling method which reduces computation time, at a slight loss of image clarity. The proposed morphological filtering technique reduces speckle and, at the same time, preserves some fine boundary details and fractal features.

### 3.2.6 Homomorphic Processing

Speckle noise can be assumed to be multiplicative. In order to convert this multiplicative noise to additive noise, the natural logarithm of the image can be taken, as follows:

$$log(g(x,y)) = log(f(x,y)) + log(I(x,y)) \tag{3.3}$$

where **g** is the speckled image, **f** is the uncorrupted image and **I** is the speckle noise. Now any technique for removing additive noise can be used to filter the image. This filtering in the logarithmic domain is known as homomorphic

nal Processing for Remote Sensing, Rome, SPIE Proceedings, vol.2315, pp 123-125,1994.

filtering. Sadjadi [40] applied Homomorphic Median and Homomorphic spatial average filters to images and found that Homomorphic filtering showed little or no improvement over normal filtering.

The LLMMSE filter proposed by Harvey [16] was also applied to the log intensity domain representation of the image and this was found to produce similar results to those obtained in the intensity domain (see Section 3.2.3).

In a recent paper by Franceschetti *et al.* [10], an iterative homomorphic technique using Wiener filtering is presented. The intensity image is transformed into the log intensity domain and then into the frequency domain for Wiener filtering. Due to the lack of prior knowledge relating to the unspeckled scene, an iterative technique is implemented in order to approximate the power spectrum of the speckle-free image. This filtering method, when applied to ESAR single look data, produced a final image ENL of 19. The filtering method is also shown to maintain good edge clarity.

Homomorphic processing has been found to be no more effective than Intensity domain filtering when applied in conjunction with filters which reduce additive or multiplicative noise. Improvements over Intensity domain techniques seem unlikely to result from this area of speckle reduction. However, this method of filtering is useful in that it allows noise reduction techniques which have been developed for additive noise to be applied to speckled images.

### 3.2.7 Transform Domain Filtering

Early approaches to speckle reduction consisted of a two dimensional Fourier Transform, followed by 2-D low pass filtering of the resulting spectrum. Due to the fact that a large proportion of the speckle noise power is situated in the high-frequency area, this technique reduced the speckle noise favourably. The chief limitation of this technique is that high-frequency image components were not taken into account and this resulted in drastic blurring of edges and point targets as well as the eradication of certain scene texture variations.

MacKinnon [25] proposes a method of noise reduction which involves minimization of cross-entropy. The cross-entropy function for a set of equally spaced intensities, $x_k > 0, k = 0, 1, 2, ..., N-1$, is given by:

$$H = \sum_{k=0}^{N-1} x_k ln(x_k/b_k) \tag{3.4}$$

where $b_k$ is a set of values which represents the prior knowledge of the intensities $x_k$. MacKinnon showed the sample entropy to be a function of phase in the Fourier domain. The cross-entropy method was shown to be an iterative method which can be computationally expensive if provided with incorrect prior knowledge.

Ranchin and Cauneau [38] present a method based on the *Wavelet Transform (WT)* and Wiener filtering. An improvement is achieved over methods based on the Fourier transform and Wiener filtering. Ranchin investigates the choice of WT algorithm and mother wavelet. In addition to reducing the speckle noise, it is claimed that the WT method only reduces the geometric resolution of the filtered image slightly.

### 3.2.8 Other Filtering Techniques

A *Geometric Filter* is implemented by Crimmins [6]. A *convex hulling* algorithm is applied alternately to the image and its inverse in horizontal, vertical and the two diagonal directions. The *convex hull* of a set of points $S$ is defined as the boundary of the smallest convex domain containing $S$. For further details of this geometric technique, see Preparata and Shamos [37, pp. 95-225]. Comparisons are drawn between the performance of this filter and multiple look processing. Results seem to indicate that the geometric filter outperforms look averaging. From the presented results it seems as though this filter could be used on single look images in order to produce high resolution, speckle reduced images. However Lee [21] states that this filter will blur and possibly wipe out features such as roads and rivers which are only one or two pixels wide.

A simplified and improved version of Crimmins' filter, called ABSEIL, is proposed by Whatmough [47]. The performance of ABSEIL was found to be superior to the Mean and Median filters in that more detail was preserved. Unlike local statistics methods, it removed noise close to edges, and it also smoothed speckle in areas of both weak and strong background, unlike Crimmins' Geometric filter.

## 3.3 Summary of Techniques

In order to compare different speckle reduction techniques, it is necessary to take a number of factors into account. The trade off between geometric and radiometric resolutions is the most important factor. Computational efficiency can also be very important, as SAR images can contain a large number of data points, making the filtering process a lengthy one. Other important factors include the applicability of a particular filter to the type of terrain being imaged and the purpose for which the final image will be used.

Few of the reviewed papers give full quantitative results for the speckle filters. Qualitative assesment is useful for comparing images which differ markedly, but an accurate comparison can only be conducted with the aid of quantitative results for both the radiometric and geometric resolutions.

The non-adaptive filtering techniques are the most computationally efficient and tend to reduce speckle significantly. These techniques, in general, do not preserve edge clarity as well as the other forms of technique which have been presented. The two basic techniques are the Mean and Median filters and two promising algorithms are the KNN and Lorentzian filters, which seem to offer the best edge preservation of this class of technique. These four filters have been chosen for implementation.

The adaptive filtering techniques are less computationally efficient and do not result in as great a degree of speckle reduction as the non-adaptive techniques. This class of technique, however, provides good edge preservation and can maintain the original information content of the image. Two algorithms

which are typical of this class of technique are the Frost and Gamma-Gamma MAP filters. Another interesting algorithm is the Hirosawa filter for single look images, which, together with the Frost and MAP filters, has been chosen for implementation and evaluation.

Shape adaptive filtering techniques are often heuristic in nature and it is difficult to predict their performance compared to ordinary adaptive filters. An interesting filter is the MHR filter which has been selected for implementation and evaluation.

Segmentation based filtering is a promising field. The segmentation task is made difficult by the presence of speckle in the image and this, together with the increased complexity of filtering algorithms creates severe limitations to this type of filtering. Due to time constraints, no filters from this class of technique have been implemented.

There are few promising speckle reduction algorithms in the Geometric, Homomorphic and Transform domain classes. The focus of this dissertation has been the larger, more promising classes which give an indication of the general trends in the field of speckle reduction. In keeping with this, no filter from these smaller, less promising classes has been implemented.

# Chapter 4

# Implementation of Filtering Algorithms

This chapter details the various filtering approaches and algorithms which have been implemented as part of this dissertation. The various filters described in this section are not necessarily the best or most efficient, but they have been chosen in order to demonstrate either basic principles or otherwise promising techniques. Certain of the filters have been chosen as they seem to be representative of a class of techniques which the author believes to be efficient or promising in some way.

The parameters for each filter have been investigated and this is presented in appendices B to I. A single set of parameters has been chosen for each filter. These parameters are used in the filtering of all test images, but it should be noted that, for optimal filtering, parameters should be chosen for each specific image.

A number of test images, both filtered and unfiltered, are presented in Appendix K. Each image has been processed with the filters described in this section. It should be noted that the displayed images have been Gamma corrected in order to produce an acceptable range of intensity values. This process alters the speckle statistics slightly, but is necessary for viewing processed SAR images.

## 4.1 Mean Filter

The Mean (Box) filter [23] is a basic, non-adaptive filter which smooths speckle. The algorithm involves running an $N \times N$ filter across the speckled image. The filter is moved one pixel at a time and at each location, the central pixel is replaced by the mean value of the $N^2$ pixels contained within the window. The filter is run left to right, top to bottom of the image, however the two-dimensional nature of the filter makes the order of operation arbitrary. The *C* code for the mean filter is given in Appendix B.

The Mean filter reduces speckle by simply smoothing the image. In this way the speckle amplitude is reduced, but it is expected that point targets and edges will also be blurred or removed. The parameters for the Mean filter have been chosen to yield a large reduction in the amount of speckle, at the price of greatly reduced geometric resolution (see Appendix B).

## 4.2 Median Filter

The Median filter [40] is similar in operation to the Mean filter (see previous section). The only difference in the two algorithms is that the Median filter requires the centremost pixel of the filtering window to be replaced with the median (not the mean) of the pixel values within the window. The *C* code for this filter is given in Appendix C.

The Median filter is not expected to reduce speckle as significantly as the Mean filter, but good edge preservation is expected. In addition, it is also possible that point targets could be removed from the image. The parameters for the Median filter (see Appendix C) have been chosen to yield the best edge preservation possible.

## 4.3 Lorentzian Filter

The Lorentzian filter [12] is a derivative of the Mean filter and seeks to improve on its performance by introducing a weighting function onto the

window. Thus the window shape is changed from a rectangular function to the Lorentzian function. The Lorentzian function is expected to result in improved edge clarity after filtering, while still reducing the level of speckle noise. The Lorentzian function is applied according to the distance from the centre of the window to the pixel in question. The Lorentzian weighting function is given by the following [12]:

$$w(x) = \frac{1}{1 + \pi^2 x^2} \tag{4.1}$$

where $x$ is the distance from the centre of the filtering window. The *C* code which implements the Lorentzian filter can be found in Appendix E.

The parameters for the Lorentzian filter have been chosen to yield the best possible filter performance (see Appendix E).

## 4.4 K Nearest Neighbour Filter

The K Nearest Neighbour filter [29] is also a derivative of the mean filter. This filter seeks to improve on the performance of the mean filter by averaging only a certain number of pixels ($K$) which are closest in intensity value to the pixel which is to be replaced.

The algorithm works by first dividing the pixels within the window into two groups according to their intensity values, *i.e.* those greater or less than the central pixel. The pixels within these groups are now arranged in order of ascending intensity value. Pixels which are equal in intensity to the central pixel are placed straight into an array of size $K$, for averaging. This array is now filled with the pixels from the two groups which are closest in value to the central window pixel. When the array has been filled, the values are averaged and the central window pixel is replaced with this value. This provides a certain amount of adaptivity to edges, in that the majority of the pixels which are averaged will lie on the same side of the edge as the central pixel, as they have similar intensity values. The code listing and parameter details for the KNN filter are given in Appendix D. The parameters for the KNN filter have been chosen to reduce speckle significantly, while still maintaining some edge clarity.

## 4.5 Hirosawa Filter

This filter is proposed by Hirosawa and Kimura [17]. It is designed for use on single look images and is suggested as a substitute for conventional frequency domain multiple look techniques. The code for the Hirosawa filter can be found in Appendix F.

The filter operates on the principle that, in an homogeneous area, one may adjust individual pixel values ($P$) without losing information, as long as the mean pixel value is unchanged. The filter algorithm is as follows:

- A threshold value, $k$ is set in order to identify homogeneous and non-homogeneous areas.
- The local image mean, $\mu$ and standard deviation, $\sigma$ are calculated for an n×n window with central pixel value equal to $P_i$.
- If $\sigma/\mu > k$ then $P = P_i$.
- If $\sigma/\mu \leq k$ then $P = \mu + g(P_i - \mu)$ where $g$ is a parameter which specifies the amount of speckle reduction performed. A value of $g = 0.5$ corresponds roughly to 4 looks, while $g = 0.6$ corresponds to 3 looks.

Despite the fact that this filter is suggested for use on 1 look images only, its design does not restrict it to only single look images. This filter has therefore been applied to both single and multiple look images.

The filter parameters (see Appendix F) have been chosen to maintain good edge clarity, while still reducing the amount of image speckle.

## 4.6 Gamma-Gamma MAP Filter

The *Maximum a Posteriori (MAP)* filter is presented by Lopes *et al.* [24]. The presented filter assumes a Gamma speckle model and a Gamma reflectivity model as well as $K$-distributed multilook data. Two MAP filters are presented by Lopes, a single point MAP filter for multiple look images and a multiple points filter for the filtering of single look SAR images. The single

point filter has been implemented and the source code can be found in Appendix G.

This filter involves running an $N \times N$ window over the image and replacing the central pixel. The pixels within the filtering window are used to calculate the local statistics for the pixel which is to be replaced. For a filtering window containing $N^2$ pixels, let $I$ be the $N^2$ dimensional observation vector and $R$ be the $N^2$ dimensional scene vector. Now, the MAP equation is developed by maximizing the *a posteriori* conditional pdf $p_c(R/I)$, *i.e.* $R_{MAP} = arg_R[max\ p_c(R/I)]$. Where $arg_R[]$ indicates the argument of the PDF. According to the Bayes rule,

$$p_c(R/I) = p_R(R)p_c(I/R)/p_I(R) \tag{4.2}$$

and it is therefore equivalent to maximize the right hand side of equation 4.2. Here, $p_c(I/R)$ is the maximum likelihood term, describing the detected intensity, given the noise model. The first and second order speckle statistics are introduced through $p_c(I/R)$, while $p_R(R)$ is the scene model. Assuming a $K$ distributed pdf for image intensity, $p_R(R)$ is given by:

$$p_R(R) = \frac{\alpha^\alpha}{< R >^\alpha} \frac{1}{\Gamma(\alpha)} exp(-\alpha \frac{R}{< R >}) R^{\alpha-1} \tag{4.3}$$

where,

- $\alpha = 1/ < C_R >^2$ is the heterogeneity parameter
- $C_R = \sigma_R / < R >$
- $\sigma_R$ is the local scene standard deviation
- $< R >$ is the local mean value of the scene

Now, the single point MAP solution for the new value of the window's central pixel is:

$$R_{MAP} = \frac{(\alpha - L - 1) < R > + \sqrt{< R >^2 (\alpha - L - 1)^2 + 4\alpha L I < R >}}{2\alpha} \tag{4.4}$$

where $L$ is the number of looks, $I$ is the observed intensity value and $R_{MAP}$ is the value to be calculated.

The parameters for the MAP filter have been chosen so as to provide the best possible edge preservation (see Appendix G). The single point Gamma-Gamma MAP filter is not designed for 1 look images, and good filter performance is therefore only expected for multiple look images.

## 4.7 Frost Filter

Frost *et al* [11] develop a statistical model for radar noise and an image enhancement technique which is based on this model. The image enhancement technique makes use of a minimum mean square error (MMSE) filter which attempts to estimate the terrain backscatter from the image data.

In designing an MMSE filter, it is necessary to assume stationarity of both the signal and noise. In radar images, it is valid to assume that the noise is stationary, but invalid to assume a stationary model for the signal, as the mean backscatter varies according to the type of surface being imaged. It is therefore necessary to adapt the filter to local changes in the amount of backscatter. The local mean and standard deviation are used to adapt the filter so that MMSE estimates are produced inside locally homogeneous areas.

The image model gives $I$, the image intensity, as:

$$I(x,y) = [r(x,y)n(x,y)] * h(x,y) \tag{4.5}$$

where $r$ is the radar backscatter, $n$ is the speckle noise, $h$ is the system impulse response and $*$ denotes convolution. In developing the filter, use is made of $\overline{r}$ and $\sigma_r$, the backscatter mean and standard deviation, as well as $\overline{n}$ and $\sigma_n$ the speckle noise mean and standard deviation.

An MMSE estimate of $r(x,y)$ is now generated from $I(x,y)$ and from this, the filter impulse response $(m(x,y))$ is obtained. The mean square error $\epsilon$, is given by:

$$\epsilon^2 = E[(r(t) - I(t) * m(t))^2] \tag{4.6}$$

where $t = (x, y)$ is the spatial coordinate. From this, it can be shown [11] that the impulse response of the filter is given by:

$$m(t) = K_1 \alpha e^{-\alpha|t|} \tag{4.7}$$

where

$$\alpha = \sqrt{2a\left[\frac{\overline{n}}{\sigma_n}\right]^2 \frac{1}{1+[\frac{\overline{r}}{\sigma_r}]^2} + a} \tag{4.8}$$

and $K_1$ is a normalizing constant. Frost goes on to show that

$$\alpha^2 \propto \sigma_I^2 / \overline{I}^2 \tag{4.9}$$

The Frost filter is an adaptive filter which is expected to preserve edges well whilst reducing the amount of speckle. The algorithm for this filter is based on the mean filter algorithm, but the central window pixel is replaced with the average of the surrounding pixels, weighted according to equation 4.7. The code for the Frost filter can be found in Appendix H. The filter parameters for the Frost filter have been chosen to provide a compromise between edge preservation and speckle reduction (see Appendix H).

## 4.8 MHR Filter

A promising filter which adapts itself to the largest possible homogeneous region is presented by Wu and Maître [49]. The algorithm first calculates the standard deviation for the filtering window and the change in local standard deviation when the window size is increased. These two quantities are used in order to ascertain whether the window contains a pike, an edge or is homogeneous. If the window covers an homogeneous region, the window size is increased and the process is repeated. Windows containing pikes are filtered using local statistics filtering and windows containing edges are subjected to the shape adaptation process. In this way, the window size is increased until an edge enters the window, or a maximum window size is reached.

Windows which have reached the preset threshold size are subjected to local statistics filtering. Windows in which an edge is detected are assigned one of eight configurations, according to the orientation of the edge. Once an edge

configuration has been assigned, the window (now called a semi-window) is grown in the direction away from the edge, resulting in a large homogeneous region which is filtered using local statistics methods.

The paper by Wu and Maître provides only a brief explanation of the semi-window configuration process. It is stated that the configuration process is driven by grey level gradient detection, followed by appropriate quantization into one of eight edge configurations. Because of a lack of information regarding to the original edge detection process, an algorithm was developed in order to determine the final edge quantization.

The new edge detection algorithm involved multiplying the window by two orthogonal binary level masks, containing the values 1 and $-1$. These masks are shown in Figure 4.1. The masks are used to calculate edge gradient in

| | | |
|---|---|---|
| 1 | 1 | 1 |
| | | |
| -1 | -1 | -1 |

| | | |
|---|---|---|
| -1 | | 1 |
| -1 | | 1 |
| -1 | | 1 |

Figure 4.1: Masks for calculating edge orientation

the x and y directions and these gradients are then used to assign an edge configuration number.

The developed edge quantization process was found to be not robust enough to detect edges reliably in the presence of speckle. This caused the detection of false edges and the introduction of artifacts into the final image. As a result the final results were not as ideal as those which Wu and Maître claim to have achieved. Development of a more robust edge detection technique was deemed to be beyond the scope of this thesis and the results for the MHR filter have been included in a separate appendix, along with details of the

filtering algorithm. The MHR filtered test images have also been included in Appendix K.

# Chapter 5

# Filter Evaluation

This chapter details the results produced by the filters described in Chapter 4. The filters are run over the test image shown in Figure 5.1. The imaged scene

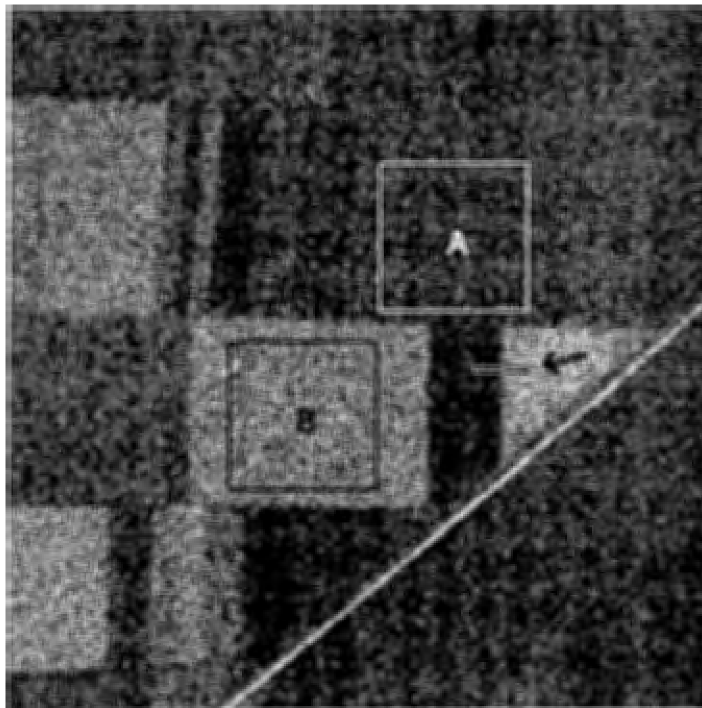


Figure 5.1: ESAR test image for evaluation of filters

contains a number of fields with varying intensities as well as a railway line. The raw imagery was taken from a flight of the DLR's C-band sensor of the airborne

sensor operated at C-Band (5.3 $GHz$). Processing at UCT has generated both single and multiple look images from the raw data and these images are used in testing the efficiency of the spatial filters.

A simulated SAR image is also used for evaluation of the filters. This image, which is shown in Figure 5.2, contains simulated fields at various intensities, as well as linear features and point targets. The simulated image has an ENL

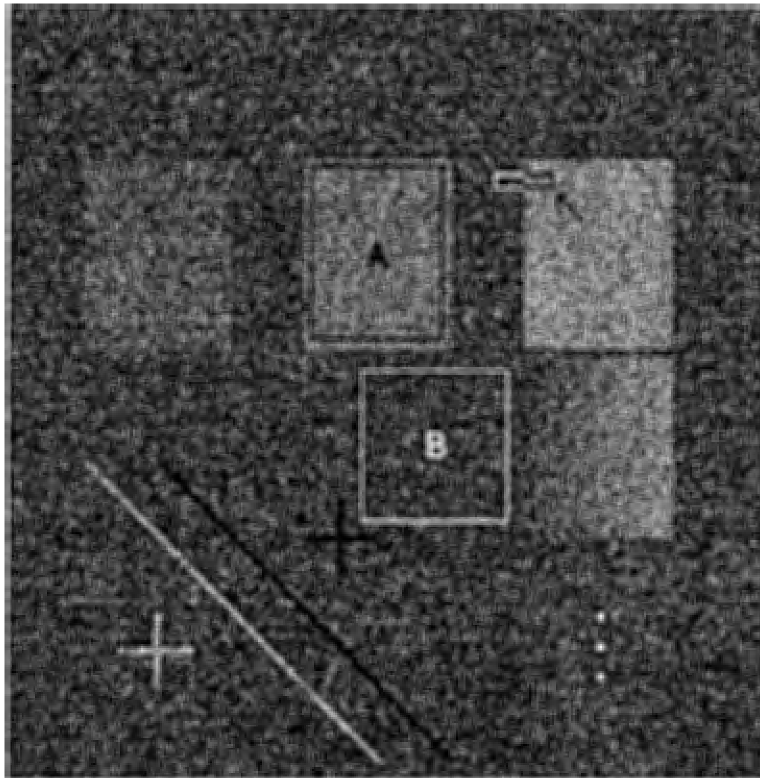


Figure 5.2: Simulated SAR image for evaluation of filters

value of four and the speckle has a normal distribution. Normally distributed speckle was chosen as a convenient approximation to the more complicated Chi-squared distribution.

The amount of speckle in an image is referred to as the radiometric resolution of the image. A measure of the amount of speckle in the filtered images is obtained by considering the local statistics in the homogeneous areas marked $A$ and $B$ in Figure 5.1 and Figure 5.2. The term geometric resolution refers to the amount of fine detail that is represented in the image. A useful indication

of the geometric resolution of an image can be obtained by observing the steepness of edges. The marked areas, covering edges in Figure 5.1 and Figure 5.2 are used to observe the geometric resolution of the test image after filtering. In order to reduce the effect of speckle on the edge measurement, a number of adjacent parallel cuts through the edges are averaged.

## 5.1 Radiometric Resolution

In order to obtain a quantitative measure of the amount of speckle in the test image, use was made of the *equivalent number of looks (ENL)*. The *ENL* was calculated using the local statistics of the homogeneous areas marked in Figures 5.1 and 5.2. The results given in Table 5.1 are for the areas labelled *B*. For full results, refer to Appendix A.

Table 5.1 gives ENL values for various combinations of look number and filter. The ENL values given for the three unfiltered images correspond well to the

Table 5.1: ENL for different look/filter combinations

| | ESAR data | | simulated data |
|---|---|---|---|
| | 1 look | 4 look | 4 look |
| **Frost** | 1.04 | 6.04 | 8.17 |
| **Hirosawa** | 2.43 | 10.88 | 14.32 |
| **KNN** | 2.02 | 9.96 | 12.99 |
| **Lorentzian** | 1.27 | 5.08 | 19.49 |
| **MAP** | 1.34 | 7.70 | 7.67 |
| **mean** | 10.62 | 53.82 | 201.42 |
| **median** | 1.99 | 11.26 | 23.77 |
| **unfiltered** | 1.01 | 4.10 | 4.07 |

actual number of looks for each image. These unfiltered ENL values provide a starting point for the various filters, and all improvements should be measured relative to them. It is, however, important to note that these results are highly dependent on the choice of filter parameters. The performance of these filters, relative to each other, can be influenced by adjusting the various filter parameters and this should be borne in mind when comparing

the various filters with each other. As an example of this, the ENL value for a mean filtered, three look image is seen to vary from 9.69 to 37.67 when the window size is varied from 3 to 7 pixels on a side (see Appendix B, Table B.1).

As expected, the mean filter provides the largest ENL value for all test images. This filter is able to reduce the amount of speckle substantially.

The Frost and MAP filters, which both fall into the Adaptive Filters category, produce the lowest ENL values for all images. The Frost filter does not affect the ENL value of the one look image, but a noticeable improvement in ENL is evident for both of the four look images. This indicates that this filter is not suitable for speckle reduction of one look images. The MAP filter, as expected, also produces better results for the multiple look images than for the single look case. Both of these adaptive filters are thus more suited to speckle reduction in multiple look images.

The Hirosawa and KNN filters produce ENL values which are higher than the Frost and MAP filters, but in most cases less than both the mean and median filters. The Hirosawa filter performs well for all three test images, supporting the assumption that its algorithm is valid for single and multiple look images. Excluding the mean and median filters, the Hirosawa filter produces the best ENL values for both ESAR images.

The median filter produces ENL values which are similar to those of the Hirosawa and KNN filters for all three images. For the single look image, the performance of the median filter is inferior, while for the four look images, the median filter provides greater speckle reduction than the Hirosawa and KNN filters.

The Lorentzian filter, which has little effect on the ENL value of the 1 look image also performs poorly when used to filter the 4 look ESAR image. This filter, however, performs well on the simulated 4 look image, suggesting that it may be suitable for filtering images which have little or no correlation between adjacent pixels due to speckle size.

While certain filters (see above) do not reduce speckle in the one look image, all of the algorithms examined are able to increase the ENL noticeably for the 4 look images. The lower bound on ENL improvement is provided by the Lorentzian and Frost filters which, in the case of the 4 look ESAR image, increase the ENL to approximately 5 and 6 looks respectively. The majority of the filters provide larger ENL gains for the simulated 4 look image than for the ESAR 4 look image. This is due to the speckles in the simulated image only covering one pixel, whilst, in the ESAR image, the speckles are larger. Smaller speckles mean that fewer pixels are needed in order to increase the ENL by the same amount and this leads to the higher values in the right-hand column of Table 5.1.

## 5.2 Geometric Resolution

In addition to removing speckle, spatial filters tend to blur edges and point targets and remove texture, thereby removing fine detail from an image. This degradation corresponds to a decrease in geometric resolution. Reduction in geometric resolution can be measured by observing an edge before and after filtering.

The edge areas (marked with arrows) in Figures 5.1 and 5.2 are used to obtain a quantitative indication of image resolution. After averaging a number of adjacent cuts through the edge, an edge measure was calculated. Various methods of obtaining an edge measure were tried and some of the more conventional techniques proved to be imprecise in differentiating between various filters. The methods tried were:

- Measuring the slope between the points one third and two thirds of the way up the edge. Because of the small number of pixels over which the edge lies (between 2 and 6), this method was unable to distinguish between edges which were obviously very different.

- Measuring the slope between the mean values at the upper and lower sides of the edge. This produced results which were very similar to

those produced above. This quantitative edge measure also failed to agree with the qualitative decisions which were obtained from viewing a cut through the edge.

- Measuring the slope between the maximum and minimum edge values. Despite being relatively unorthodox, this method produced the best agreement with qualitative results obtained from observation of the edge. Smoothing tends to affect the top and bottom of an edge more than the central portion. It is because of this that the previous two methods failed to distinguish sufficiently between various edges.

Following from the above discussion of techniques, the last method was chosen for calculating the edge measure. According to this, the edge measure is given by:

$$\textit{edge measure} = \frac{\Delta Y}{\Delta X \times < I >} \tag{5.1}$$

where $\Delta Y$ is the difference in intensity value between the peak and trough pixels, $\Delta X$ is the pixel difference between the peak and trough pixels and $< I >$ is the image mean. The factor of $< I >$ is included in order to allow comparison of images filtered with different algorithms, which might have different image means. An edge cut in which the peak and trough pixels are identified is given in figure 5.3. Visual comparison of the two curves reveals that the unfiltered image (high geometric resolution) has a sharper edge than the image after mean filtering (low geometric resolution). The small difference between the two curves in the central edge region supports the fact that the first two attempts at an edge measure method were inconclusive.

Figure 5.4 is a bar graph of the edge measure values for the one look ESAR image, filtered with all of the chosen filters. It is expected that the unfiltered image should have the highest edge measure and that the filtered images should all have slightly degraded edges, as none of the filters included any sort of edge enhancement algorithm. Figure 5.4 shows that the Frost, Lorentzian,MAP and median filters all have edge measure values which are comparable with the value for the unfiltered image. These results indicate that the Frost, Lorentzian, MAP and median filters do not degrade the edge appreciably. It is interesting to note that the Lorentzian filter (see

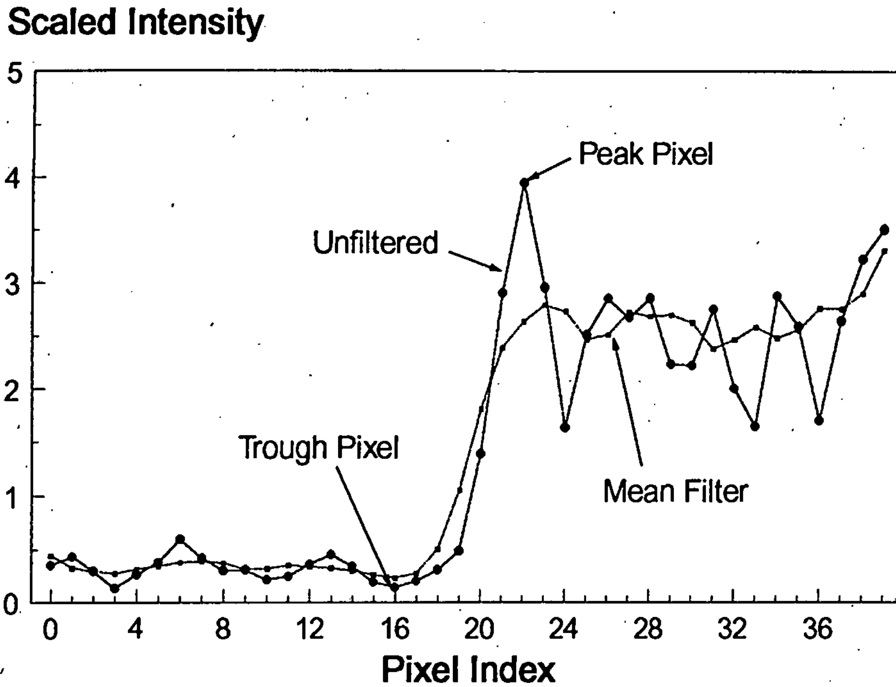


Figure 5.3: Edge cut indicating peak and trough pixels

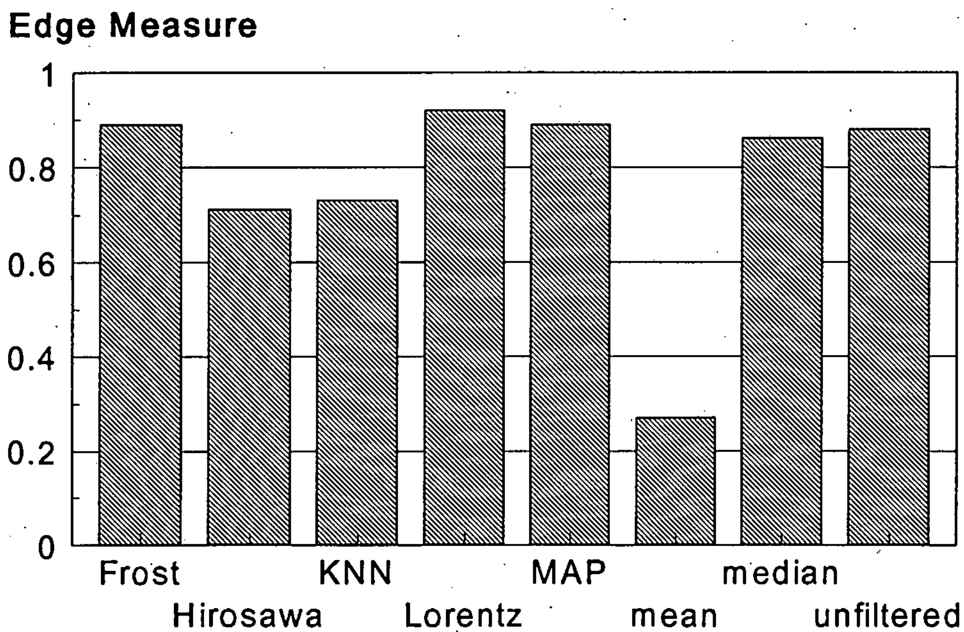


Figure 5.4: Edge measure for filtered 1 look ESAR image

Appendix A, Table A.2) provides edge measure values which are slightly better than the unfiltered case for all three test images. The Lorentzian filter modifies the mean values of the areas surrounding the edge, causing the intensity variation across the edge and therefore the edge measure value to be increased. This effect can be seen clearly in Figure 5.5. In the areas ad-

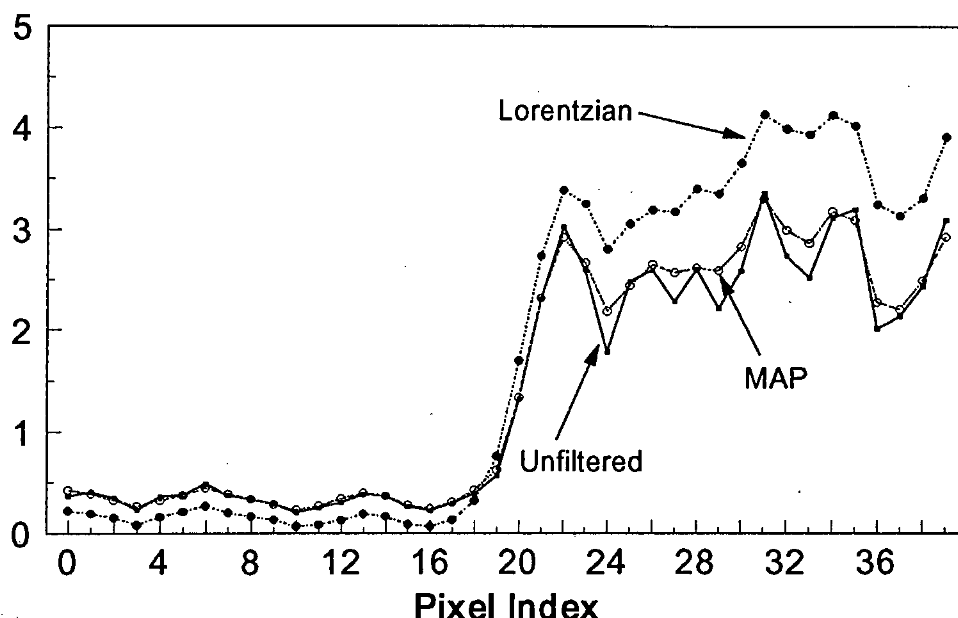


Figure 5.5: Edge plot of ESAR 4 look image for various filters

jacent to the edge, the Lorentzian filtered image has intensity values which are appreciably different from the unfiltered and MAP filtered images. The alteration of mean values relative to each other perturbs the real data and may cause problems in image interpretation.

The mean filter produces the lowest edge measure values for all three test images (see Figures 5.4, 5.6 and 5.7). Visually, this edge degradation presents itself as severe blurring in the filtered image.

The Hirosawa and KNN filters have similar edge preserving characteristics and for all test images produce edge measure values which lie below those of the other adaptive filters. For the 4 look images, however, the Hirosawa and KNN filters cause less edge degradation than for the 1 look case.

The median filter performs well on the one look image, but causes edge degradation on the four look images. The results on the ESAR four look image are comparable to those of the KNN and Hirosawa filters. When used to filter the simulated four look image, the loss in geometric resolution was greater than that caused by all other filters, except for the mean filter.

Figures 5.6 and 5.7 are bar graphs of the edge measure values for the four look ESAR image and simulated four look image.

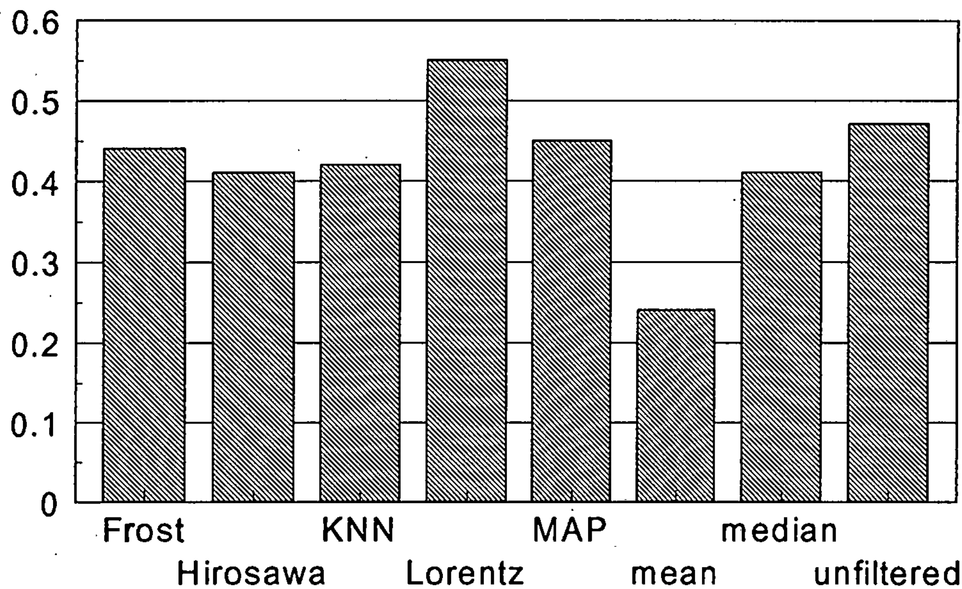


Figure 5.6: Edge measure for filtered 4 look ESAR image

## 5.3 Summary of Results

The results presented in the previous two sections serve to quantify the tradeoff between geometric and radiometric resolution which is associated with any speckle reduction algorithm. It is important to note that these results can be changed by varying the parameters of each filter.

The edge measure values given above clearly show that images filtered with local statistics based methods have the highest geometric resolution. These

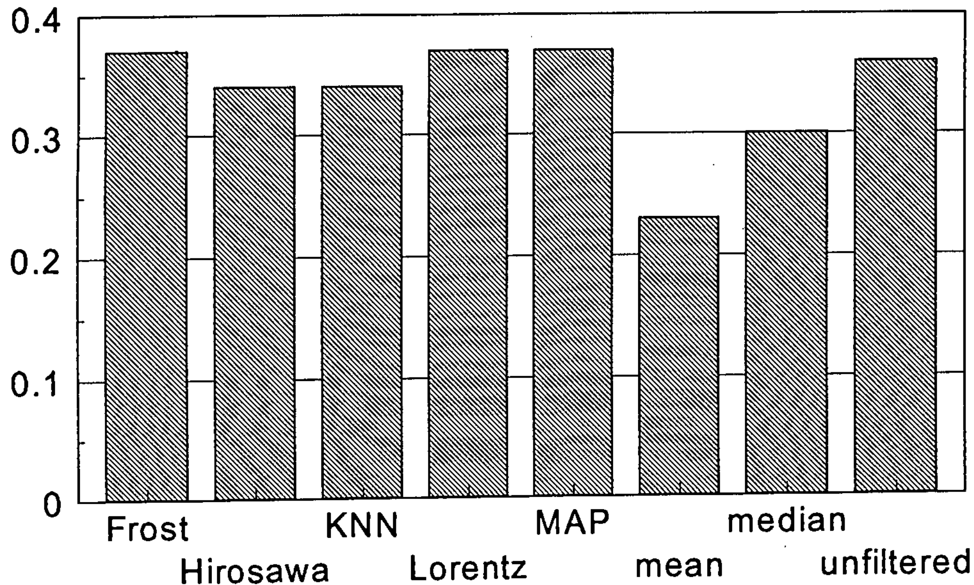


Figure 5.7: Edge measure for filtered simulated SAR image

local statistics methods (the Frost and MAP filters) do not, however, produce greatly improved radiometric resolution. These filters are ideally suited to the filtering of images which contain a large amount of fine detail, such as urban scenes.

The mean filter produces images with low geometric resolution, but is able to improve the radiometric resolution more than the other filters. This filter is useful only for reducing speckle in images which contain large-scale detail such as large fields or geographical features.

The Lorentzian filter, which alters local mean values in the image (as discussed in Section 5.2), produces high edge measure results, but low ENL values. This filter is only successful when applied to the simulated SAR image. It is thought that the lack of correlation between adjacent speckles in the simulated image makes the Lorentzian filter more effective.

The Hirosawa filter, which is a less elegant adaptive filter represents a good trade off between geometric and radiometric resolution. This filter and the KNN filter both yield considerably improved ENL values and are able to

maintain edge clarity in the test images. These filters lend themselves to general-purpose despeckling in images which do not contain large amounts of fine detail. The median filter performs as well as the Hirosawa and KNN filters on the ESAR images, but not as well when used to filter the simulated image. This is because the median filter does not derive any benefit from the lack of inter pixel correlation. It is important to note that a major limitation of the median filter is that it has the potential to wipe out small point targets.

# Chapter 6

# Conclusions

An analysis of the results presented in chapter 5 shows that post image formation techniques present a practical and efficient method of reducing speckle in SAR images. These methods require only image data and do not need complex in-phase and quadrature data as their input.

The process of speckle reduction involves a trade-off between the radiometric and geometric resolutions of the image. Increased speckle reduction leads to increased edge degradation and a compromise acceptable to the end user must be reached. In order to select the correct speckle reduction algorithm, it is important to consider the application for which the final image will be used. Applications which make use of fine image detail will require filtering which preserves edges and other small-scale image features. Applications which only concentrate on large-scale features do not have edge preservation requirements which are as stringent.

Certain classes of speckle filter (*i.e.* local statistics filtering) are able to maintain excellent edge clarity, but only a slight reduction in speckle noise is achieved. Other types of filter (*i.e.* the Mean filter) are able to reduce the speckle noise substantially, but severe edge degradation is brought about. A good compromise between radiometric and geometric resolution is achieved by the KNN and Hirosawa filters as these provide good speckle reduction, whilst still maintaining some edge clarity.

The speckle reduction algorithms have a number of parameters which influence the performance of the filters. The filters can be highly dependent on the choice of input parameter. It is important that filter parameters are optimized for the particular task at hand in order to give the best possible results.

The filters which have been investigated provide an indication of the potential of post image formation speckle reduction techniques. It is possible to select filters or groups of filters which can filter SAR images according to the needs of different end users. Post image formation techniques are easy to apply and are capable of further reducing the speckle in multiple look SAR images. Certain of these techniques can also be used as a substitute for multiple look processing in order to provide speckle reduced one look images.

# Chapter 7

# Recommendations

In order to gain complete insight into the potential of various speckle reduction algorithms, it will be necessary to investigate fully all classes of post image formation algorithm. Particularly, an investigation of the performance of segmentation based methods is suggested. Further work on the edge detection stage of the MHR filter is also required in order to evaluate its potential as a viable filter.

A recent paper by Franceschetti *et al.* (see section 3.2.6) presents an efficient speckle reduction algorithm. This work was discovered in the final stages of this thesis and was not implemented due to time constraints. Implementation and evaluation of this algorithm is suggested as a future work.

Further research into other areas of speckle reduction, such as multiple polarization and multiple frequency methods, could provide valuable information about how these techniques compare with the post image formation methods reviewed in this dissertation.

# Bibliography


[1] Gamal H. Abdelhamid, Hany M. Assal, and Moustafa N. Fahmy. A New Filter Method for Speckle Reduction in SAR Images . In *Antennas and Propogation Society Symposium* , volume 2, pages 1036–1039, 1991.

[2] M. Adair and Bert Guindon. Methods for evaluating speckle-suppressing filters based on edge detection performance. *Canadian Journal of Remote Sensing*, 15(2):100–108, September 1989.

[3] Gilbert V. April and E. R. Harvey. Speckle statistics in four-look synthetic aperture radar imagery. *OPTICAL ENGINEERING*, 30(4):375–381, April 1991.

[4] M.R. Azimi-Sadjadi and Sami Bannour. Two-Dimensional Adaptive Block Kalman Filtering of SAR Imagery . *IEEE Trans on Geosc. and Remote Sensing*, 29(5):742–753, September 1991.

[5] Andrea Baraldi and Flavio Parmiggiani. A Modified Version of the SAR Speckle Filter Based on Structure Detection. In *IGARSS '94*, volume 4, pages 2168–2172. IEEE, 1994.

[6] Thomas R. Crimmins. Geometric filter for reducing Speckle . *OPTICAL ENGINEERING*, 25(5):651–654, May 1986.

[7] Jean M. Durand, Bernard J. Gimonet, and Jacqueline R. Perbos. SAR Data Filtering for Classification . *IEEE Trans on Geosc. and Remote Sensing*, GE-25(5):629–637, September 1987.

[8] John Ellis, Richard G. White, and Martin Warner. Smoothing SAR Images with Neural Networks . In *IGARSS '94*, volume 4, pages 1883–1885. IEEE, 1994.

[9] Alessandra Forte. Optimised Techniques to Reduce Speckle in SAR Images . In *Proceedings of the 18th European Microwave Conference.* , pages 699–704, 1988.

[10] G Franceschetti, G Schirinzi, and Vito Pascazio. Iterative homomorphic technique for speckle reduction in synthetic-aperture radar imaging. *Journal of the Optical Society of America, A.*, 12(4):686–694, April 1995.

[11] Victor S. Frost, Julian C Holtzman, Josephine Abbott Stiles, and K. S. Shanmugan. A Model for Radar Images and its Application to Adaptive Digital Filtering of Multiplicative Noise. *IEEE Trans on Pattern Anal and Machine Intelligence*, PAMI-4(2):157–166, March 1982.

[12] Daniel Held Fuk-Kwok Li and Cheryl Croft. Comparison of Several Techniques to Obtain Multiple-look SAR Imagery. *IEEE Trans on Geosc. and Remote Sensing*, GE-21(3):370–375, July 1983.

[13] Joseph W. Goodman. A random walk through the field of speckle . *OPTICAL ENGINEERING*, 25(5):610–612, May 1986.

[14] Christopher Gordon. Speckle Reduction in Synthetic Aperture Radar Images Anglo American Report, February 1994.

[15] Wilhelm Hagg and Manfred Sties. Efficient Speckle Filtering of SAR Images . In *IGARSS 94* , volume 4, pages 2140–2142. IEEE, August 1994.

[16] E.R. Harvey and Gilbert V. April. Speckle Reduction in Synthetic-Aperture-Radar Imagery . *Optics Letters*, 15(13):740–742, July 1990.

[17] Haruto Hirosawa and Hiroshi Kimura. Generation of Speckle-Reduced One-Look SAR Images . In *PROCEEDINGS OF THE 1989 INTERNATIONAL SYMPOSIUM OF NOISE AND CLUTTER REJECTION IN RADARS AND IMAGING SENSORS*, pages 705–710, 1990.

[18] Alok Kher and Sunanda Mitra. Optimum Morphological Filtering to Remove Speckle Noise from SAR Images. In *Proceedings of the SPIE*

, volume 2030, pages 97–108. Soc. Photo. Instrumentation Engineers, January 1993.

[19] D.T. Kuan, A.A. Sawchuk, T.C. Strand, and P. Chavel. Nonstationary 2-d recursive filter for speckle reduction. In *ICASSP 82*, volume 3, pages 1561–1564, 1982.

[20] Jong-Sen Lee. Speckle suppression and analysis for synthetic aperture radar images. *OPTICAL ENGINEERING*, 25(5):636–643, May 1986.

[21] Jong-Sen Lee. Statistical Modelling and Suppression of Speckle in Synthetic Aperture Radar Images. In *IGARSS '87*, volume 2, pages 1331–1339. IEEE, 1987.

[22] Armand Lopes and Franck Sery. The LMMSE Polarimetric Wishart Vector Speckle Filter for M/L data and the LMMSE Spatial Vector Filter for correlated pixels in SAR. In *IGARSS '94*, volume 4, pages 2143–2145. IEEE, 1994.

[23] Armand Lopes, Ridha Touzi, and E Nezry. Adaptive Speckle Filters and Scene Heterogeneity . *IEEE Trans on Geosc. and Remote Sensing*, 28(6):992–1000, November 1990.

[24] Armand Lopes, Ridha Touzi, E Nezry, and H Laur. Structure Detection and Statistical Adaptive Speckle in SAR Images. *International Journal of Remote Sensing*, 14(9):1735–1758, June 1993.

[25] R.F. Mackinnon. Minimum cross-entropy noise reduction in images . *Journal of the Optical Society of America, A.*, 6(6):739–747, June 1989.

[26] P. Mancini and H.D. Griffiths. Speckle reduction by spatial filtering. In *IEE Colloquium on Synthetic Aperture Radar*, volume 143, pages 81–86, 1989.

[27] F.J. Martin and R.W. Turner. SAR speckle reduction by weighted filtering . *Int. J. Remote Sensing*, 14(9):1759–1774, 1993.

[28] Nelson D. A. Mascarenhas and Alejandro C. Frery. SAR image filtering with the ICM algorithm . In *IGARSS '94*, volume 4, pages 2185–2187. IEEE, 1994.

, volume 2030, pages 97–108. Soc. Photo. Instrumentation Engineers, January 1993.

[19] D.T. Kuan, A.A. Sawchuk, T.C. Strand, and P. Chavel. Nonstationary 2-d recursive filter for speckle reduction. In *ICASSP 82*, volume 3, pages 1561–1564, 1982.

[20] Jong-Sen Lee. Speckle suppression and analysis for synthetic aperture radar images. *OPTICAL ENGINEERING*, 25(5):636–643, May 1986.

[21] Jong-Sen Lee. Statistical Modelling and Suppression of Speckle in Synthetic Aperture Radar Images. In *IGARSS '87*, volume 2, pages 1331–1339. IEEE, 1987.

[22] Armand Lopes and Franck Sery. The LMMSE Polarimetric Wishart Vector Speckle Filter for M/L data and the LMMSE Spatial Vector Filter for correlated pixels in SAR. In *IGARSS '94*, volume 4, pages 2143–2145. IEEE, 1994.

[23] Armand Lopes, Ridha Touzi, and E Nezry. Adaptive Speckle Filters and Scene Heterogeneity . *IEEE Trans on Geosc. and Remote Sensing*, 28(6):992–1000, November 1990.

[24] Armand Lopes, Ridha Touzi, E Nezry, and H Laur. Structure Detection and Statistical Adaptive Speckle in SAR Images. *International Journal of Remote Sensing*, 14(9):1735–1758, June 1993.

[25] R.F. Mackinnon. Minimum cross-entropy noise reduction in images . *Journal of the Optical Society of America, A.*, 6(6):739–747, June 1989.

[26] P. Mancini and H.D. Griffiths. Speckle reduction by spatial filtering. In *IEE Colloquium on Synthetic Aperture Radar*, volume 143, pages 81–86, 1989.

[27] F.J. Martin and R.W. Turner. SAR speckle reduction by weighted filtering . *Int. J. Remote Sensing*, 14(9):1759–1774, 1993.

[28] Nelson D. A. Mascarenhas and Alejandro C. Frery. SAR image filtering with the ICM algorithm . In *IGARSS '94*, volume 4, pages 2185–2187. IEEE, 1994.

[29] Penny M. Masuoka. Digital Processing of Orbital Radar Data to Enhance Geologic Structure: Examples from the Canadian Shield. *PHOTOGRAMMETRIC ENGINEERING AND REMOTE SENSING*, 54(5):621–632, May 1988.

[30] Alberto Moreira. Improved Multilook Techniques Applied to SAR and SCANSAR Imagery . *IEEE Trans on Geosc. and Remote Sensing*, 29(4):529–534, July 1991.

[31] Makoto Nagao and Takashi Matsuyama. Edge preserving smoothing. *Computer Graphics and Image Processing*, 9(4):394–407, April 1979.

[32] Krishna S. Nathan and John C. Curlander. Speckle Noise Reduction of 1-Look SAR Imagery . In *IGARSS '87*, volume 2, pages 1457–1462. IEEE, 1987.

[33] E Nezry, H G Kohl, and H De Groof. Restoration and Enhancement of Textural Properties in SAR Images Using Second Order Statistics. In *Proceedings of the 1st European Symp. on Satellite Remote Sensing*, volume 2316, September 1994.

[34] C.J. Oliver. Information from SAR images . *Journal of Physics D:Applied Physics*, 24:1493–1514, January 1991.

[35] Kazuo Ouchi, Shahram Tajbakhsh, and Ronald E. Burge. Dependance of Speckle Statistics on Backscatter Cross-Section Fluctuations in Synthetic Aperture Radar Images of Rough Surfaces. *IEEE Trans on Geosc. and Remote Sensing*, GE-25(5):623–628, September 1987.

[36] A. Papoulis. *Probability, Random Variables and Stochastic Processes.* McGraw Hill, New York, 1965.

[37] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry.* Springer-Verlag, New York, USA, 1985.

[38] Thierry Ranchin and Francois Cauneau. Speckle reduction in Synthetic Aperture Radar Imagery using Wavelets. *Soc. Photo. Instrumentation Engineers*, 2034:432–441, 1993.

[39] J.P. Rossi and D. Maystre. Rigorous Numerical Study of Speckle Patterns for Two-dimensional, Random Microrough Surfaces. *OPTICAL ENGINEERING*, 25(5):613–617, May 1986.

[40] Firooz A. Sadjadi. Perspective on Techniques for Enhancing Speckled Imagery . *OPTICAL ENGINEERING*, 29(1):25–30, January 1990.

[41] M. S. Scivier and D. G. Corr. Adaptive Enhancement of Multi-look Synthetic Aperture Radar Images. In *IGARSS '86*, volume 3, pages 1635–1638. IEEE, 1986.

[42] Zhenghao Shi and Ko B. Fung. A Comparison of Digital Speckle Filters. In *IGARSS '94*, volume 4, pages 2129–2133. IEEE, 1994.

[43] Kiyo Tomiyasu. Computer Simulation of Speckle in a Synthetic Aperture Radar Image Pixel. *IEEE Trans on Geosc. and Remote Sensing*, GE-21(3):357–363, July 1983.

[44] Hideyuki Ueno and Haruto Hirosawa. A Filter for Reducing Speckle of Synthetic-Aperture Radar Imagery . *Electronics & Communications in Japan, Part 1*, 71(3):92–100, 1988.

[45] Fawwaz T. Ulaby, Richard K. Moore, and Adrian K. Fung. *Microwave Remote Sensing, Active and Passive*, volume III. Artech House, Norwood, MA, USA, 1986.

[46] F.T. Ulaby. Textural Information in SAR Images . *IEEE Trans on Geosc. and Remote Sensing*, GE-24(2):235–245, March 1986.

[47] R.J. Whatmough. ABSEIL - A Feature-Preserving Filter for SAR Imagery . In *Proceedings of RADARCON '90. First Australian Radar Conf.* , volume 2, pages 579–585, April 1990.

[48] Richard G. White. Cross-section Estimation by Simulated Annealing . In *IGARSS '94*, volume 4, pages 2188–2190. IEEE, 1994.

[49] Yifeng Wu and Henri Maitre. Smoothing Speckled Synthetic Aperture Radar Images by Using Maximum Homogeneous Region Filters. *OPTICAL ENGINEERING*, 31(8):1785–1792, August 1992.

# Appendices

# Appendix A

# Results

This appendix contains tables of all results produced during this thesis. First tables of ENL values and then tables containing edge measure values are presented.

## A.1 ENL Values

In Table A.1, ENL values are given for each of the three test images. Each image has two sample areas (A and B), where the ENL has been measured. Values are given for each filter as well as for the unfiltered case.

## A.2 Edge Measure Values

In Table A.2, edge measure values are given for each of the three test images. Each image has a designated sample edge. A number of adjacent cuts through the edge have been averaged in order to reduce the effect of the speckle on the edge measurement. Values are given for each filter as well as for the unfiltered case.

Table A.1: ENL values for all image/filter combinations

| | **ESAR 1 look** | | **ESAR 4 look** | | **sim. 4 look** | |
|---|---|---|---|---|---|---|
| | *area A* | *area B* | *area A* | *area B* | *area A* | *area B* |
| **Frost** | 1.14 | 1.04 | 5.81 | 6.04 | 8.20 | 8.17 |
| **Hirosawa** | 2.73 | 2.43 | 10.59 | 10.88 | 14.69 | 14.32 |
| **KNN** | 2.08 | 2.02 | 9.65 | 9.96 | 12.64 | 12.99 |
| **Lorentzian** | 2.27 | 1.27 | 8.99 | 5.08 | 28.33 | 19.49 |
| **MAP** | 1.47 | 1.34 | 7.36 | 7.70 | 7.67 | 7.67 |
| **mean** | 11.51 | 10.62 | 50.48 | 53.82 | 196.19 | 201.42 |
| **median** | 2.06 | 1.99 | 10.70 | 11.26 | 22.32 | 23.77 |
| **unfiltered** | 1.10 | 1.01 | 3.95 | 4.10 | 4.15 | 4.07 |

Table A.2: Edge measure values for all image/filter combinations

| | **ESAR 1 look** | **ESAR 4 look** | **sim. 4 look** |
|---|---|---|---|
| **Frost** | 0.89 | 0.44 | 0.37 |
| **Hirosawa** | 0.71 | 0.41 | 0.34 |
| **KNN** | 0.73 | 0.42 | 0.34 |
| **Lorentzian** | 0.92 | 0.55 | 0.37 |
| **MAP** | 0.89 | 0.45 | 0.37 |
| **mean** | 0.27 | 0.24 | 0.23 |
| **median** | 0.86 | 0.41 | 0.30 |
| **unfiltered** | 0.88 | 0.47 | 0.36 |

# Appendix B

# Mean Filter

The mean filter has a very simple algorithm which is briefly described in section 4.1. This Appendix contains a listing of the source code for the mean filter algorithm, as well as an investigation into the effect of varying the filter parameters.

## B.1 Source Code

The following 2 pages contain the source code for the mean filter.

```c
/*===================================================================

 program to implement MEAN FILTER in C

 by: Mark Gebhardt, (c) University of Cape Town (March 1995)

 ===================================================================*/

/*-------------------------------------------------------------------

CLASS:  Speckle Filters

MODULE: mean

INFO: The Mean (Box) filter {lopes:90} is a basic, non-adaptive algorithm
which smoothes speckle.
The algorithm involves running an NXN filter across the speckled
image. The filter is moved one pixel at a time and at each location, the
central pixel is replaced by the mean value of the N^2 pixels contained
within the window. The filter is run left to right, top to bottom of the
image, however the two-dimensional nature of the filter should make the
order of operation arbitrary.
The Mean filter reduces speckle by simply smoothing the image. In this way
the speckle amplitude is reduced, but it is expected that point targets and
edges will also be blurred or removed.


PARAMETERS:     I_DATA - name of input file
                O_DATA - name of output file
                IMGLINES - y dimension of input block
                LINESIZE - x dimension of input block
    WINSIZE - specifies side of square filtering window (must be odd)
                ITER - the number of ITERations
                (default values assigned in code)

---------------------------------------------------------------------*/


#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "iofunct.h"

/*----------GLOBAL VARIABLE DEFINITIONS---------------*/

FILE *fp1,*fp2;
FILE *outfile,*specfile,*infile;
char  specname[80],inname[80],outname[80],ext[4];
int linesize,imglines,winsize,iter;
int x,y,i,count,win_off,linepos,linenum,imgpos,size,size_i,half_wsize;
int *filt_ptr,*filtstart,*filt_line_ptr,sum;
unsigned char *arrstart,*arrayptr,*arr_line_ptr,*out_arr,*out_arr_st;
unsigned long imgsize;
unsigned char lsb,msb;

/*----------PROCEDURE DEFINITIONS--------------------*/

void convert_array(void);
void assignments(void);
void get_image(void);
void processing(void);
void write_image(void);

/*----------PROC. TO CONVERT ARRAY TO CHAR-----------*/

void convert_array(void)
{
   if ((out_arr = (unsigned char *) calloc(imgsize,size)) == NULL )
      puts("can't make array");
   out_arr_st = out_arr;
   filt_ptr = filtstart;
   imgpos = 0;
   while (imgpos<imgsize)
       {
          if ((*filt_ptr<=255) && (*filt_ptr>=0))
              *out_arr = (unsigned char)(*filt_ptr);
          else if (*filt_ptr<0)
              *out_arr = 0;
          else if (*filt_ptr>255)
              *out_arr = 255;
          out_arr++;
          filt_ptr++;
          imgpos++;
       }
}

/*-----------------ASSIGNMENTS--------------------*/

void assignments(void)
{
     /* Open spec file */
     printf ("\nExtension for spec file (mean.???) - max 3 characters : ");
     scanf ("%s", ext);
     sprintf(specname, "mean.%s", ext);
     if ( (specfile = fopen (specname, "rt") ) == NULL )
       {
         printf ("ERROR: Specfile %s not found/opened\n", specname);
         exit(0);
       }

   /* Read processing specs *
    *************************/

     skipchar(specfile,'>');
     fscanf (specfile, "%d", &linesize);
     skipchar(specfile,'>');
     fscanf (specfile, "%d", &imglines);
     skipchar(specfile,'>');
     fscanf (specfile, "%d", &winsize);
     skipchar(specfile,'>');
     fscanf (specfile, "%d", &iter);
     skipchar(specfile,'>');
     GetString(specfile,inname);           /* Space taken care of */
     skipchar(specfile,'>');
     GetString(specfile,outname);          /* Space taken care of */

     fclose(specfile);

     size = sizeof(unsigned char);
     size_i = sizeof(int);

     imgsize = imglines*linesize;
     if ((filt_ptr = (int *) calloc(imgsize,size_i)) == NULL )
        puts("can't make array");
     if ((arrayptr = (unsigned char *) calloc(imgsize,size)) == NULL )
        puts("can't make array");
     arrstart = arrayptr;
     filtstart = filt_ptr;

     /* OPEN INPUT AND OUTPUT IMAGE FILES */

     if ((fp1 = fopen(inname,"rb"))==NULL)
       {
```

ICI14PRIN

```c
            printf("cannot open input file\n");
            exit(1);
        }
    if ((fp2 = fopen(outname,"wb"))==NULL)
        {
            printf("cannot open output file\n");
            exit(1);
        }
    puts("Filtering SAR image, using MEAN filter");
}

/*------------------PROC. TO READ IMAGE----------------*/

void get_image(void)
{
    /* Read in the image */
    arrayptr = arrstart;

    for (y=0;y<imglines;y++)   /* read a line */
        {
            fread(arrayptr,linesize*size,1,fp1);
            arrayptr = arrayptr+linesize;
        }
    puts("Image read.");
}

/*--------------PROC TO DO PROCESSING----------------*/

void processing(void)
{
    /*  THE PROCESSING GETS DONE HERE !!!!!!! */

    win_off = ((int)(((float)winsize/2)+0.5));
    half_wsize = (winsize+1)/2-1;

    puts("Busy filtering ...");
    printf("Window size = %d pixels\n",winsize*winsize);

    printf("The number of iterations = %d\n",iter);
    for (count = 1; count <=iter ; count++)
        {
            linenum = win_off;

            while (linenum < (imglines - (win_off - 1)))
                {
                    filt_line_ptr = filtstart+(linenum*linesize);
                    arr_line_ptr = arrstart+(linenum*linesize);
                    linepos =(win_off);
                    filt_ptr = filt_line_ptr + linepos;
                    arrayptr = arr_line_ptr + linepos;

                    while (linepos < (linesize - (win_off) - 1))
                        {
                            /* APPLY THE RELEVANT FILTER */
                            /* TRY A MEAN FILTER */
                            sum = 0;
                            for (y=-half_wsize; y<=half_wsize ; y++)
                                for (x=-half_wsize ; x<=half_wsize ; x++)
                                    {
                                        sum = sum + (*(arrayptr+y*linesize+x));

                                    }
                            *filt_ptr = sum/((float)(winsize*winsize));
                            linepos++;
                            filt_ptr++;
                            arrayptr++;
                        }
                    linenum++;
                }
    /* now copy the filtered array back into the original
     for next iteration*/
            if (iter > 1)
                {
                    arrayptr = arrstart;
                    filt_ptr = filtstart;
                    for (x=0 ; x<imglines*linesize ; x++)
                        {
                            *arrayptr = *filt_ptr;
                            arrayptr++;
                            filt_ptr++;
                        }
                }
        }
    convert_array();

    free(filt_ptr);
}

/*-------------PROC TO WRITE IMAGE-----------------*/

void write_image(void)
{
    /* NOW  WRITE TO THE FILE */

    out_arr = out_arr_st;

    for (y=0; y<imglines; y++)
        {
            fwrite(out_arr,linesize*size,1,fp2);
            out_arr = out_arr + linesize;
        }
    puts("Written to file, program complete.");
}

/*---------------MAIN PROGRAM--------------------*/

int main()
{
    assignments();
    get_image();
    processing();
    write_image();

    free(out_arr);
    fclose(fp2);
    fclose(fp1);
    return 0;
}
```

## B.2 Parameter Investigation

The mean filter has two input parameters, *i.e.* window size and number of iterations. Table B.1 shows the variation in ENL and edge measure for various input parameters. The window size parameter is the length of one side of the square filtering window. The image used was a three look version of the ESAR image used throughout this thesis. The measurements were taken at the locations specified in Chapter 5. The mean filter is characterized by

Table B.1: ENL and edge values from mean filtered image for various parameter values.

| iterations | window size | edge measure | ENL |
|---|---|---|---|
| 1 | 3 | 0.26 | 9.69 |
| 1 | 5 | 0.16 | 21.69 |
| 1 | 7 | 0.15 | 37.67 |
| 2 | 5 | 0.16 | 36.75 |

high ENL values and low edge measure values for all combinations of the input parameters. The ENL value is seen to increase markedly for window sizes greater than 3. In order to obtain a high ENL value, the following parameters were chosen: $iterations = 1, window\ size = 7$. It should be noted that the ENL value increases, without bound, as the window size and number of iterations are increased.

## B.3 Theoretical Calculations

The following calculations are presented as a theoretical check of the performance of the mean filter.

For a speckled SAR image, the local signal to noise ratio may be defined as:

$$SNR = \frac{\sigma_I}{< I >}$$

where $\sigma_I$ and $< I >$ are the local standard deviation and mean values. For a single look image, $\sigma_I / < I > = 1$ and for a multiple look image, with $L$ equal

to the number of independent looks,

$$SNR_{ml} = \frac{\sigma_I}{< I > \sqrt{L}} = \frac{1}{\sqrt{L}}$$

similarly, for an image which has speckles that are restricted to a single pixel, after mean filtering with window size equal to $N \times N$,

$$SNR_{filt} = \frac{\sigma_I}{< I > \sqrt{N^2}}$$

so, for a mean filtered multiple look image,

$$\frac{\sigma_I}{< I >} = \frac{1}{N\sqrt{L}}$$

now, the equivalent number of looks is defined as

$$ENL = \frac{< I >^2}{\sigma_I^2} = N^2 L$$

Consulting Table A.1, for the simulated 4 look image, $L = 4$ and $N = 7$, giving:

$$ENL = 7^2 \times 4 = 196$$

which agrees favourably with the experimental values of 196.19 and 201.42 given in Table A.1.

# Appendix C

# Median Filter

The median filter has a simple algorithm which is explained in section 4.2. This Appendix contains an elaboration on the algorithm, a listing of the source code and an investigation of the effect of varying the filter parameters.

## C.1 Algorithm

The median filter algorithm is very similar to the algorithm for the mean filter. In order to obtain the median intensity value contained in the filtering window, it is necessary to sort the pixels according to their intensities.

The function *sort* takes an array which is the size of the filtering window and performs a bubble sort, creating an array which is ordered according to intensity value. The central (median) value can now be used to replace the relevant image pixel.

## C.2 Source Code

The following two pages contain the source code for the Median filter.

```
/*=========================================================================

 program to implement MEDIAN FILTER in C

 by: Mark Gebhardt, (c) University of Cape Town (March 1995)

 =========================================================================*/

/*-------------------------------------------------------------------------

CLASS:  Speckle Filters

MODULE: median

INFO:   The Median filter {sadjadi:90} is very similar in operation to the
Mean filter. The only difference in the two algorithms is that the Median filter
requires the centermost pixel of the filtering window to be replaced with the
median (not the mean) of the pixel values within the window.
The median filter is not expected to reduce speckle as significantly as the
mean filter, but excellent edge preservation is expected. In addition, it is
also possible that point targets could be reduced in amplitude or even wiped
out.

PARAMETERS: inname - name of input file
    outname - name of output file
    imglines - y dimension of input block
    linesize - x dimension of input block
    winsize - specifies size of square filtering window (winsizeXwinsize) (must
  => be odd)
    iter - the number of iterations
              (default values assigned in code)

---------------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "iofunct.h"

/*-----------GLOBAL VARIABLE DECLARATIONS------------------*/

FILE *fp1,*fp2;
FILE *outfile,*specfile,*infile;
char  specname[80],inname[80],outname[80],ext[4];
int linesize,imglines,winsize,iter;
int x,y,i,count,win_off,linepos,linenum,imgpos,size,half_wsize;
unsigned char *window,sum;
unsigned char *arrayptr,*filt_ptr,*filt_line_ptr;
unsigned char *arrstart,*filtstart,*arr_line_ptr;
unsigned char *head_array,line_hd_arr[12][200];
unsigned long imgsize;
unsigned char lsb,msb;


/*------------FUNCTION DECLARATIONS-----------------------*/

void sort (unsigned char *arr_ptr, int length);

/*--------------PROCEDURE DECLARATIONS------------------*/

void assignments(void);
void get_image(void);
void processing(void);
void write_image(void);

/*-----------------ASSIGNMENTS--------------------*/

void assignments(void)
{
     /* Open spec file */
     printf ("\nExtension for spec file (median.???) - max 3 characters : ");
     scanf ("%s", ext);
     sprintf(specname, "median.%s", ext);
     if ( (specfile = fopen (specname, "rt") ) == NULL )
       {
         printf ("ERROR: Specfile %s not found/opened\n", specname);
         exit(0);
       }

   /* Read processing specs *
    *************************/

    skipchar(specfile,'>');
    fscanf (specfile, "%d", &linesize);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &imglines);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &winsize);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &iter);
    skipchar(specfile,'>');
    GetString(specfile,inname);           /* Space taken care of */
    skipchar(specfile,'>');
    GetString(specfile,outname);          /* Space taken care of */

    fclose(specfile);

    size = sizeof(unsigned char);
    imgsize = imglines*linesize;
    if ((filt_ptr = (unsigned char *) calloc(imgsize,size)) == NULL )
        puts("can't make array");
    if ((arrayptr = (unsigned char *) calloc(imgsize,size)) == NULL )
        puts("can't make array");

    arrstart = arrayptr;
    filtstart = filt_ptr;

    /* OPEN INPUT AND OUTPUT IMAGE FILES */

      if ((fp1 = fopen(inname,"rb"))==NULL)
         {
            printf("cannot open input file\n");
            exit(1);
         }
    if ((fp2 = fopen(outname,"wb"))==NULL)
       {
          printf("cannot open output file\n");
          exit(1);
       }
    puts("Filtering SAR image, using MEDIAN filter");
}

/*-----------------PROC. TO READ IMAGE---------------*/

void get_image(void)
{
    /* Read in the image */
    arrayptr = arrstart;

    for (y=0;y<imglines;y++)   /* read a line */
       {
      fread(arrayptr,linesize*size,1,fp1);
```

ICI 4PRINT

```
      arrayptr = arrayptr+linesize;
        }
    puts("Image read.");
}

/*--------------PROC TO DO PROCESSING----------------*/

void processing(void)
{
    /*  THE PROCESSING GETS DONE HERE !!!!!!! */

    half_wsize = (winsize+1)/2 -1;
    win_off = ((int)(((float)winsize/2)+0.5));
    linenum = win_off;
    if ((window = (unsigned char *) calloc(winsize*winsize,sizeof(int))) == NULL
  =>  )
        puts("can't make array");

    window = window + half_wsize*(1+winsize);

    printf("Window size = %d pixels\n",winsize*winsize);

    printf("The number of iterations = %d\n",iter);
    for (count = 1; count <=iter ; count++)
        {
            linenum = win_off;

            puts("Busy filtering ...");

      while (linenum < (imglines - (win_off - 1)))
            {
        filt_line_ptr = filtstart+(linenum*linesize);
        arr_line_ptr = arrstart+(linenum*linesize);
                    linepos =(win_off);
                    filt_ptr = filt_line_ptr + linepos;
                    arrayptr = arr_line_ptr + linepos;

        while (linepos < (linesize - (win_off) - 1))
                    {
                        /* APPLY THE RELEVANT FILTER */
                        /* TRY A MEDIAN FILTER */

                        for (y=-half_wsize; y<=half_wsize ; y++)
                            for (x=-half_wsize ; x<=half_wsize ; x++)
                                {
          *(window+y*winsize+x) = *(arrayptr + y*linesize + x);
                                }
          sort(window, winsize);
                        *filt_ptr = *(window);

                        /*  if ((sum<=255) && (sum>=0))
                            *filt_ptr = (unsigned char)(sum);
                        else if (sum<0)
                            *filt_ptr = 0;
                         else if (sum>255)
                            *filt_ptr = 255;*/

                        linepos++;
                        filt_ptr++;
                        arrayptr++;
                    }
                linenum++;
            }

            /* now copy the filtered array back into the original for next itera
  => tion*/
        if (iter > 1)
                {
                        arrayptr = arrstart;
                        filt_ptr = filtstart;
            for (x=0 ; x<imglines*linesize ; x++)
                        {
                            *arrayptr = *filt_ptr;
                            arrayptr++;
                            filt_ptr++;
                        }
                }

        }
}

/*--------------PROC TO WRITE IMAGE------------------*/

void write_image(void)
{
    /* NOW  WRITE TO THE FILE */

    filt_ptr = filtstart;
    arrayptr = arrstart;

    for (y=0; y<imglines; y++)
        {
      fwrite(filt_ptr,linesize*size,1,fp2);
      filt_ptr = filt_ptr + linesize;
        }
    puts("Written to file, program complete.");
}

/*------------MAIN PROGRAM-------------------*/

int main()
{
    assignments();
    get_image();
    processing();
    write_image();

    free(filt_ptr);
    free(arrayptr);
    fclose(fp2);
    fclose(fp1);
    return 0;
}

/*----------FUNCTION TO DO SORTING----------------*/

void sort (unsigned char *arr_ptr, int wsize)
{
    int half_wsize;
    unsigned char *start,temp;
    int SORTED, MADE_SWAP, i, false, true;

    half_wsize = (int)(wsize+1)/2 -1;
    false = 0;
    true = 1;
    SORTED = false;
    start = arr_ptr - half_wsize*(1+winsize);
    while (SORTED == false)
    {
        arr_ptr = start;
        MADE_SWAP = false;
        for (i=1; i<wsize*wsize; i++)
```

```
            {
                if (*arr_ptr > *(arr_ptr+1))
                    {
                        temp = *arr_ptr;
                        *arr_ptr = *(arr_ptr+1);
                        *(arr_ptr+1) = temp;
                        MADE_SWAP = true;
                    }
                arr_ptr++;
            }
        if (MADE_SWAP == false)
            SORTED = true;
    }
    arr_ptr = start;
    return;
}
```

IC/14PRIN

## C.3 Parameter Investigation

The median filter has two input parameters, *i.e.* window size and number of iterations. Table C.1 shows the variation in ENL and edge measure for various input parameters. The window size parameter is the length of one side of the square filtering window. The image used was a three look version of the ESAR image used throughout this thesis. The measurements were taken at the locations specified in Chapter 5. The median filter represents

Table C.1: ENL and edge values from median filtered image for various parameter values.

| iterations | window size | edge measure | ENL |
|---|---|---|---|
| 1 | 3 | 0.60 | 7.91 |
| 1 | 5 | 0.40 | 17.35 |
| 1 | 7 | 0.35 | 30.75 |
| 2 | 5 | 0.30 | 28.59 |

the classical trade off between ENL and edge measure. In order to obtain the best possible edge clarity in the filtered image, the following parameters were chosen: $iterations = 1, window\ size = 3$.

# Appendix D

# KNN Filter

The algorithm for the K Nearest Neighbour (KNN) filter is described in Section 4.4. This appendix contains an elaboration on the algorithm, a listing of the source code and an investigation of the effect of varying the filter parameters.

## D.1 Algorithm

The KNN filter operates by running a square window across the image and using the pixels covered by the window to replace the central pixel. The central pixel is replaced by the mean of the $K$ pixels which are nearest in intensity value to the central window pixel. The algorithm which follows gives an indication of how the algorithm determines the $K$ pixels to be averaged. The variables in italics appear in the source code and the various steps are indicated within the code.

1. identify pixels covered by the filtering window using a pointer to the central pixel (*arrayptr*) and the window size (*winsize*).

2. extract the intensity values which are less than **arrayptr* to array (*lt_array*).

3. extract the intensity values which are greater than **arrayptr* to array (*gt_array*).

4. extract the intensity values which are equal to **arrayptr* to *ave_array*.

5. sort *lt_array* and *gt_array*.

6. add smallest member of *gt_array* to *ave_array* and set *latest* equal to the difference between **arrayptr* and the most recent member of *ave_array*.

7. while (**arrayptr* − **lt_array*) <= *latest*, add **lt_array* to *ave_array* and increment *lt_array* and *ave_array*.

8. add next member of *gt_array* to *ave_array*. If *ave_array* is not full, goto 7.

9. find mean of *ave_array* and use it to replace **arrayptr*.

## D.2 Source Code

The following three pages contain the source code for the KNN filter.

```c
/*=====================================================================

 program to implement K NEAREST NEIGHBOUR FILTER in C

 by: Mark Gebhardt, (c) University of Cape Town (March 1995)

 ======================================================================*/

/*---------------------------------------------------------------------

CLASS: Speckle Filters

MODULE: knn

INFO: The K Nearest Neighbour filter {masuoka:88} is also a derivative of the
Mean filter. This filter seeks to improve on the performance of the Mean filter
by averaging only a certain number of pixels (K) which are closest in value to t
  => he
pixel which is to be replaced.

PARAMETERS: inname - name of input file
    outname - name of output file
    imglines - y dimension of input block
    linesize - x dimension of input block
    winsize - specifies size of square filtering window (must be odd)
    iter - the number of iterations
               K - the number of nearest neighvours
               (default values assigned in code)

-------------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "iofunct.h"

/*------------GLOBAL VARIABLE DECLARATIONS----------*/

FILE *fp1,*fp2;
FILE *outfile,*specfile,*infile;
char  specname[80],inname[80],outname[80],ext[4];
int linesize,imglines,winsize,iter,K;
int x,y,i,lt_len,gt_len,latest,count;
int win_off,linepos,linenum,imgpos,size,half_wsize,sum;
unsigned char *arrayptr,*filt_ptr,*filt_line_ptr,*lt_array,*gt_array,*ave_array;
unsigned char *arrstart,*filtstart,*arr_line_ptr,*lt_start,*gt_start,*ave_start;
unsigned long imgsize;
unsigned char lsb,msb;

/*-------FUNCTION DECLARATIONS-----------*/

void sort (unsigned char *arr_ptr, int length);

/*--------PROCEDURE DECLARATIONS-----------*/

void assignments(void);
void get_image(void);
void processing(void);
void write_image(void);

/*-------------ASSIGNMENTS----------------*/

void assignments(void)
{
     /* Open spec file */
     printf ("\nExtension for spec file (knn.???) - max 3 characters : ");
      scanf ("%s", ext);
      sprintf(specname, "knn.%s", ext);
      if ( (specfile = fopen (specname, "rt") ) == NULL )
        {
          printf ("ERROR: Specfile %s not found/opened\n", specname);
          exit(0);
        }

    /* Read processing specs *
     *************************/

     skipchar(specfile,'>');
     fscanf (specfile, "%d", &linesize);
     skipchar(specfile,'>');
     fscanf (specfile, "%d", &imglines);
     skipchar(specfile,'>');
     fscanf (specfile, "%d", &winsize);
     skipchar(specfile,'>');
     fscanf (specfile, "%d", &iter);
     skipchar(specfile,'>');
     fscanf (specfile, "%d", &K);
     skipchar(specfile,'>');
     GetString(specfile,inname);            /* Space taken care of */
     skipchar(specfile,'>');
     GetString(specfile,outname);           /* Space taken care of */

     fclose(specfile);


     size = sizeof(unsigned char);
     imgsize = imglines*linesize;
     if ((filt_ptr = (unsigned char *) calloc(imgsize,size)) == NULL )
         puts("can't make array");
     if ((arrayptr = (unsigned char *) calloc(imgsize,size)) == NULL )
         puts("can't make array");
     if ((ave_array = (unsigned char *) calloc(K,size)) == NULL )
         puts("can't make array");
     ave_start = ave_array;
     arrstart = arrayptr;
     filtstart = filt_ptr;

     /* OPEN INPUT AND OUTPUT IMAGE FILES */

     if ((fp1 = fopen(inname,"rb"))==NULL)
         {
             printf("cannot open input file\n");
             exit(1);
         }
     if ((fp2 = fopen(outname,"wb"))==NULL)
         {
             printf("cannot open output file\n");
             exit(1);
         }
     puts("Filtering SAR image, using KNN filter");
}

/*--------------PROC. TO GET IMAGE----------------*/

void get_image(void)
{
       /* Read in the image */
    arrayptr = arrstart;

    for (y=0;y<imglines;y++)   /* read a line */
      {
      fread(arrayptr,linesize*size,1,fp1);
```

```
    arrayptr = arrayptr+linesize;
      }
   puts("Image read.");
}

/*--------PROC. TO DO PROCESSING-----------------*/

void processing(void)
{
        /*  THE PROCESSING GETS DONE HERE !!!!!!! */

   win_off = ((int)(((float)winsize/2)+0.5));
   linenum = win_off;
   puts("Busy filtering ...");
   printf("Window size = %d pixels\n",winsize*winsize);
   half_wsize = (winsize+1)/2-1;

   printf("The number of iterations = %d\n",iter);
   puts("busy with iteration number ...");
   for (count = 1; count <=iter ; count++)
   {
       printf("%d\n",count);
       linenum = win_off;

       while (linenum < (imglines - (win_off - 1)))
         {
         filt_line_ptr = filtstart+(linenum*linesize);
         arr_line_ptr = arrstart+(linenum*linesize);
               linepos =(win_off);
               filt_ptr = filt_line_ptr + linepos;
               arrayptr = arr_line_ptr + linepos;

         while (linepos < (linesize - (win_off) - 1))
                    {
                    /* APPLY THE RELEVANT FILTER */
                    /* TRY A KNN FILTER */
                    if ((lt_array = (unsigned char *) calloc(48,size)) == NUL
 => L )
                        puts("can't make array");
                    if ((gt_array = (unsigned char *) calloc(48,size)) == NUL
 => L )
                        puts("can't make array");

                    lt_start = lt_array;
                    gt_start = gt_array;

                    sum = 0;
                    lt_len = 0;
                    gt_len = 0;
                    lt_array = lt_start;
                    gt_array = gt_start;
                    ave_array = ave_start;

                    for (y=-half_wsize; y<=half_wsize ; y++)
                    for (x=-half_wsize; x<=half_wsize ; x++)
                         {
                              /* first sort the pixels into <,> and = arrays */
                              if (!((y == 0) && (x == 0)))
                                  {
             if (*(arrayptr+y*linesize+x) < *(arrayptr))
                                                   {
                                                        /* STEP 2 */
               *lt_array = *(arrayptr+y*linesize+x);
                                                        lt_array++;
                                                        lt_len++;
                                                   }
             if (*(arrayptr+y*linesize+x) > *(arrayptr))
                                                   {
                                                        /* STEP 3 */
               *gt_array = *(arrayptr+y*linesize+x);
                                                        gt_array++;
                                                        gt_len++;
                                                   }
             if (*(arrayptr+y*linesize+x) == *(arrayptr))
                                                   {
                                                        /* STEP 4 */
                                                        *ave_array = *(arrayptr);
                                                        ave_array++;
                                                   }

                                  }
                         }
                    /* now sort the arrays into ascending order - STEP 5 */
                    sort(lt_start,lt_len);
                    sort(gt_start,gt_len);

                    lt_array = lt_start+lt_len-1;
                    gt_array = gt_start;

                    if (*gt_array != 0)
                         {
                              /* STEP 6*/
                              *ave_array = *gt_array;
                              latest = *gt_array-*arrayptr;
                              ave_array++;
                              gt_array++;
                         }
                    else latest = 1;

                    while (ave_array < (ave_start + K))
                         {
                              while (((*arrayptr-*lt_array)<=latest)&&(ave_arra
 => y<(ave_start+K))&&(*lt_array!=0))
                                   {
                                        /* STEP 7 */
                                        *ave_array = *lt_array;
                                        ave_array++;
                                        lt_array--;
                                   }
                              if ((*gt_array != 0)&&(ave_array<(ave_start+K)))
                                   {
                                        /* STEP 8 */
                                        *ave_array = *gt_array;
                                        latest = *gt_array-*arrayptr;
                                        ave_array++;
                                        gt_array++;
                                   }
                              if ((*gt_array == 0)&&(*lt_array == 0))
                                   {
                                        *ave_array = *arrayptr;
                                        ave_array++;
                                        gt_array++;
                                        lt_array++;
                                   }
                              latest++;
                         }

                    sum = 0;
                    ave_array = ave_start;
                    /* STEP 9 */
                    for (i=0; i<K; i++)
```

```
                        {
                            sum = sum + *ave_array;
                            ave_array++;
                        }
                    sum = (1.1)*sum/K;

                     if ((sum<=255) && (sum>=0))
                         *filt_ptr = (unsigned char)(sum);
                     else if (sum<0)
                         *filt_ptr = 0;
                     else if (sum>255)
                         *filt_ptr = 255;

                    linepos++;
                    filt_ptr++;
                    arrayptr++;
                    lt_array = lt_start;
                    free((void*)lt_array);
                    gt_array = gt_start;
                    free((void*)gt_array);

                }
            linenum++;
        }
    /* now copy the filtered array back into the original
        for next iteration*/
    if (iter > 1)
        {
            arrayptr = arrstart;
            filt_ptr = filtstart;
      for (x=0 ; x<imglines*linesize ; x++)
                {
                    *arrayptr = *filt_ptr;
                    arrayptr++;
                    filt_ptr++;
                }
        }
    }
}

/*---------PROC. TO WRITE THE IMAGE------------------*/

void write_image(void)
{
      /* NOW  WRITE TO THE FILE */

   filt_ptr = filtstart;
   arrayptr = arrstart;

   for (y=0; y<imglines; y++)
     {
     fwrite(filt_ptr,linesize*size,1,fp2);
     filt_ptr = filt_ptr + linesize;
     }

   puts("Written to file, program complete.");
}

/*-----------MAIN PROGRAM--------------*/

int main()
{
    assignments();
    get_image();
    processing();
    write_image();
```

```
    free((void*)filt_ptr);
    free((void*)arrayptr);
    free((void*)ave_array);
    fclose(fp2);
    fclose(fp1);
    return 0;
}

/*--------FUNCTION TO SORT---------------- */

void sort (unsigned char *arr_ptr, int length)

{
    char *start, temp;
    int SORTED, MADE_SWAP, i, false, true;

    false = 0;
    true = 1;
    SORTED = false;
    start = arr_ptr;
    while (SORTED == false)
        {
            arr_ptr = start;
            MADE_SWAP = false;
            for (i=1; i<length; i++)
                {
                    if (*arr_ptr > *(arr_ptr+1))
                        {
                            temp = *arr_ptr;
                            *arr_ptr = *(arr_ptr+1);
                            *(arr_ptr+1) = temp;
                            MADE_SWAP = true;
                        }
                    arr_ptr++;
                }
            if (MADE_SWAP == false)
                SORTED = true;
        }
    arr_ptr = start;
    return;
}
```

## D.3 Parameter Investigation

The KNN filter has three input parameters, *ie.* window size, number of iterations and $K$, the number of pixels to average. Table D.1 shows the variation in ENL and edge measure for various input parameters. The window size parameter is the length of one side of the square filtering window. The image used was a three look version of the ESAR image used throughout this thesis. The measurements were taken at the locations specified in Chapter 5. A $K$ value corresponding to approximately half of the pixels in the filtering

Table D.1: ENL and edge values from KNN filtered image for various parameter values.

| iterations | window size | K | edge measure | ENL |
|---|---|---|---|---|
| 1 | 3 | 4 | 0.44 | 4.76 |
| 1 | 5 | 12 | 0.43 | 6.27 |
| 1 | 7 | 24 | 0.41 | 7.12 |
| 2 | 7 | 24 | 0.39 | 9.99 |

window was found to give the best results. Slight variations in $K$ had almost no effect on the edge and ENL values, while values of $K$ which are too small or too large produced images which looked inferior to those obtained from filtering with $K \approx \frac{(window\ size)^2}{2}$. An increase in number of iterations and window size resulted in an increase in ENL and a decrease in edge measure. In order to obtain good speckle reduction but still retain edge clarity, the following parameters were chosen: $iterations = 1, window\ size = 7, K = 24$.

# Appendix E

# Lorentzian Filter

The Lorentzian filter is based on the mean filter and its algorithm is described in Section 4.3. This Appendix contains a listing of the source code and an investigation of the effect of varying the various filter parameters.

The Lorentzian weighting function is implemented by means of the function *get_weight* and is applied to the pixels in the filtering window according to their distance from the central pixel.

## E.1 Source Code

The following two pages contain the source code for the Lorentzian filter.

```
/*============================================================================

 program to implement Lorentzian filtering in C
 filter for 8-bit images!

 by: Mark Gebhardt, (c) University of Cape Town (March 1995)

 ==========================================================================*/

/*-------------------------------------------------------------------------

CLASS: Speckle filters

MODULE: lorentz

INFO: The Lorentzian filter {li:83} is a derivative of the mean filter and
seeks to improve on its performance by introducing a weighting function onto
the window. Thus the window shape is changed from a rectangular function to
the Lorentzian function. It is hoped that the Lorentzian function will result
in improved edge clarity after filtering, while still reducing the level of spec
  => kle noise.
The Lorentzian function is applied according to the distance from the center of
the window to the pixel in question.

PARAMETERS: inname - name of input file
    outname - name of output file
    imglines - y dimension of input block
    linesize - x dimension of input block
    winsize - specifies size of square filtering window (winsizeXwinsize) (must
  => be odd)
    iter - the number of iterations
            (default values assigned in code)

---------------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>
#include "iofunct.h"

/*---------GLOBAL VARIABLE DECLARATIONS-------------------*/

FILE *fp1,*fp2;
FILE *outfile,*specfile,*infile;
char  specname[80],inname[80],outname[80],ext[4];
int linesize,imglines,winsize,iter;
int x,y,i,half_wsize,sum,sum1;
int win_off,linepos,linenum,imgpos,size,count;
unsigned char *arrayptr,*filt_ptr,*arrstart,*filtstart;
float norm;
unsigned long imgsize;
unsigned char lsb,msb;

/*-----------PROCEDURE DECLARATIONS-----------------------*/

void assignments(void);
void get_image(void);
void processing(void);
void write_image(void);

/*------------FUNCTION DECLARATIONS---------------------*/

float get_weight(int x,int y);

/*------------------ASSIGNMENTS-----------------*/

void assignments(void)
{
     /* Open spec file */
     printf ("\nExtension for spec file (lorentz.???) - max 3 characters : ");
     scanf ("%s", ext);
     sprintf(specname, "lorentz.%s", ext);
     if ( (specfile = fopen (specname, "rt") ) == NULL )
       {
         printf ("ERROR: Specfile %s not found/opened\n", specname);
         exit(0);
       }

   /* Read processing specs *
    *************************/

    skipchar(specfile,'>');
    fscanf (specfile, "%d", &linesize);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &imglines);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &winsize);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &iter);
    skipchar(specfile,'>');
    GetString(specfile,inname);            /* Space taken care of */
    skipchar(specfile,'>');
    GetString(specfile,outname);           /* Space taken care of */

    fclose(specfile);

    imgsize = imglines*linesize;
    size = sizeof(unsigned char);
    if ((filt_ptr = (unsigned char *) calloc(imgsize,size)) == NULL )
        puts("can't make array");
    if ((arrayptr = (unsigned char *) calloc(imgsize,size)) == NULL )
        puts("can't make array");
    arrstart = arrayptr;
    filtstart = filt_ptr;
    /* OPEN INPUT AND OUTPUT IMAGE FILES */

    if ((fp1 = fopen(inname,"rb"))==NULL)
        {
          printf("cannot open input file\n");
          exit(1);
        }
    if ((fp2 = fopen(outname,"wb"))==NULL)
        {
          printf("cannot open output file\n");
          exit(1);
        }
    puts("Filtering SAR image, using Lorentzian filter");
}

/*--------PROC TO GET IMAGE---------------------*/

void get_image(void)
{
    /* Read in the image */
    arrayptr = arrstart;

    for (y=0;y<imglines;y++)   /* read a line */
        {
     fread(arrayptr,linesize*size,1,fp1);
     arrayptr = arrayptr+linesize;
        }
```

```
    puts("Image read.");
}

/*-----------PROC TO DO THE PROCESSING---------*/
void processing(void)
{
    /*  THE PROCESSING GETS DONE HERE !!!!!!! */

    half_wsize = (winsize+1)/2 -1;
    win_off = ((int)(((float)winsize/2)+0.5));
    printf("The number of iterations = %d\n",iter);
    puts("filtering the image ...");
    i = (winsize*winsize);
    printf("block size = %d\n",i);

    for (count = 1; count <=iter ; count++)
        {
      printf("busy with iteration number %d\n",count);

            linenum = win_off;

      while (linenum < (imglines - (win_off - 1)))
              {
                linepos =(win_off);
        filt_ptr = filtstart+(linenum*linesize) + linepos;
        arrayptr = arrstart+(linenum*linesize) + linepos;

        while (linepos < (linesize - (win_off) - 1))
                {
                    /* A Lorentzian FILTER */
                    norm = 0;
                    for (y=-half_wsize; y<=half_wsize ; y++)
                        for (x=-half_wsize ; x<=half_wsize ; x++)
                            {
                                norm = norm + 255*get_weight(x,y);
                            }
                    norm = norm/(float)i;

                    /* multiply window pixels by their weighting factors
 =>  */
                    sum1 = 0;
                    for (y=-half_wsize; y<=half_wsize ; y++)
                        for (x=-half_wsize ; x<=half_wsize ; x++)
                            {
          sum1=sum1+(int)((*(arrayptr+y*linesize+x))*get_weight(x,y));
                            }
                    sum = ((sum1/(float)i)*255/norm);
                    if ((sum<=255) && (sum>=0))
                        *filt_ptr = (unsigned char)((float)sum*1.2);
                    else if (sum<0)
                        *filt_ptr = 0;
                    else if (sum>255)
                        *filt_ptr = 255;

                    linepos++;
                    filt_ptr++;
                    arrayptr++;
                }
                linenum++;
              }

      /* now copy the filtered array back into the original
          for next iteration*/
            arrayptr = arrstart;
            filt_ptr = filtstart;
      for (x=0 ; x<imglines*linesize ; x++)
                {
                    *arrayptr = *filt_ptr;
                    arrayptr++;
                    filt_ptr++;
                }
        }
}

/*-----------PROC TO WRITE IMAGE-------------*/

void write_image()
{
      /* NOW  WRITE TO THE FILE */

   filt_ptr = filtstart;
   arrayptr = arrstart;

   for (y=0; y<imglines; y++)
       {
     fwrite(filt_ptr,linesize*size,1,fp2);
     filt_ptr = filt_ptr + linesize;
       }
   puts("Written to file, program complete.");
}

/*------------MAIN PROGRAM----------------------*/

int main()
{
    assignments();
    get_image();
    processing();
    write_image();

    free(filt_ptr);
    free(arrayptr);
    free(filtstart);
    free(arrstart);
    fclose(fp2);
    fclose(fp1);
    return 0;
}

/*------------FUNCTION TO IMPLEMENT LORENTZIAN WEIGHTING------*/

float get_weight(int x,int y)
{
    float m, pi;

    pi = 3.1415927;
    m = 1/(1+pi*pi*(float)(x*x+y*y));

    return m;
}
```

ICT4PRINT

## E.2 Parameter Investigation

The Lorentzian filter has two input parameters, *i.e.* window size and number of iterations. Table E.1 shows the variation in ENL and edge measure for various input parameters. The window size parameter is the length of one side of the square filtering window. The image used was a three look version of the ESAR image used throughout this thesis. The measurements were taken at the locations specified in Chapter 5. The values for the Lorentzian

Table E.1: ENL and edge values from Lorentzian filtered image for various parameter values.

| iterations | window size | edge measure | ENL |
|---|---|---|---|
| 1 | 3 | 0.48 | 4.06 |
| 1 | 5 | 0.49 | 4.37 |
| 1 | 7 | 0.50 | 4.38 |
| 2 | 5 | 0.58 | 4.69 |

filter seem to indicate that both edge measure and ENL increase with increasing iterations and window size. The Lorentzian weighting function has a certain fixed width which compares favourably with a window size of 5 or 7 pixels and any further increase in window size yields no improvement in filter performance. In addition, when iterated more than twice, the filter introduces artifacts into the final image. The optimum parameters for the Lorentzian filter were chosen as: $iterations = 2, window\ size = 5$.

# Appendix F

# Hirosawa Filter

The filtering algorithm for the Hirosawa filter is described in section 4.5. This appendix contains a listing of the source code and an investigation of the various filter parameters.

## F.1 Source Code

The following 2 pages contain the source code for the Hirosawa filter.

```c
/*=========================================================================

 program to implement filtering of 1-look images in C
 filter for 8-bit images!

 by: Mark Gebhardt, (c) University of Cape Town (March 1995)

 ==========================================================================*/

/*-------------------------------------------------------------------------
CLASS:  Speckle filters

MODULE: hirosawa

INFO: This filter is proposed by Hirosawa and Kimura {hirosawa:89}. It is
designed for use on 1-look images and is suggested as a substitute for
conventional frequency-domain multi-look techniques.
The filter operates on the principle that, in an homogeneous area, one may
adjust individual pixel values without losing information, as long as the
mean pixel value is unchanged. The filter algorithm is as follows:
 - A threshold value, 'k' is set in order to identify homogeneous and
non-homogeneous areas.
 - The image mean (sigma) and standard deviation (mu) are calculated
for an NXN window with central pixel value equal to P_i.
 - If (sigma / mu) > k then P = P_i.
 - If (sigma / mu) <= k then P = mu + g(P_i - mu)
where 'g' is a parameter which specifies the amount of speckle reduction
performed. A value of g = 0.5 corresponds roughly to 4 looks, while
g = 0.6 corresponds to 3 looks.

PARAMETERS: inname - name of input file
    outname - name of output file
    imglines - y dimension of input block
    linesize - x dimension of input block
    winsize - specifies size of square filtering window (winsizeXwinsize) (must
  => be odd)
    iter - the number of iterations
                K - parameter which indicates the amont of speckle in the unfilt
  => ered image
                G - indicates the required amount of despeckling eg: g=.5 gives
  => 4-look, g=.6 gives 3-look
                (default values assigned in code)

---------------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>
#include "iofunct.h"

/*--------GLOBAL VARIABLE DECLARATIONS------------------*/

FILE *fp1,*fp2;
FILE *outfile,*specfile,*infile;
char  specname[80],inname[80],outname[80],ext[4];
int linesize,imglines,winsize,iter,K;
int x,y,i,half_wsize;
int win_off,linepos,linenum,imgpos,size,count;
unsigned char *arrayptr,*filt_ptr,*arrstart,*filtstart;
float R,sigma,sum1,sum2,G;
unsigned long imgsize;
unsigned char lsb,msb;

/*---------PROCEDURE DECLARATIONS-----------------*/

void assignments(void);
void get_image(void);
void processing(void);
void write_imnage(void);

/*---------ASSIGNMENTS-----------------*/

void assignments(void)
{
     /* Open spec file */
     printf ("\nExtension for spec file (hiro.???) - max 3 characters : ");
     scanf ("%s", ext);
     sprintf(specname, "hiro.%s", ext);
     if ( (specfile = fopen (specname, "rt") ) == NULL )
       {
         printf ("ERROR: Specfile %s not found/opened\n", specname);
         exit(0);
       }

   /* Read processing specs *
    *************************/

    skipchar(specfile,'>');
    fscanf (specfile, "%d", &linesize);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &imglines);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &winsize);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &iter);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &K);
    skipchar(specfile,'>');
    fscanf (specfile, "%f", &G);
    skipchar(specfile,'>');
    GetString(specfile,inname);          /* Space taken care of */
    skipchar(specfile,'>');
    GetString(specfile,outname);         /* Space taken care of */

    fclose(specfile);

    imgsize = imglines*linesize;
    size = sizeof(unsigned char);
    if ((filt_ptr = (unsigned char *) calloc(imgsize,size)) == NULL )
        puts("can't make array");
    if ((arrayptr = (unsigned char *) calloc(imgsize,size)) == NULL )
        puts("can't make array");
    arrstart = arrayptr;
    filtstart = filt_ptr;
    /* OPEN INPUT AND OUTPUT IMAGE FILES */

    if ((fp1 = fopen(inname,"rb"))==NULL)
        {
            printf("cannot open input file\n");
            exit(1);
        }
    if ((fp2 = fopen(outname,"wb"))==NULL)
        {
            printf("cannot open output file\n");
            exit(1);
        }
    puts("Filtering SAR image, using Hirosawa filter");
}

/*----------PROC. TO GET IMAGE-------------*/
```

```
void get_image(void)
{
    /* Read in the image */
    arrayptr = arrstart;

    for (y=0;y<imglines;y++)   /* read a line */
        {
      fread(arrayptr,linesize*size,1,fp1);
      arrayptr = arrayptr+linesize;
        }
    puts("Image read.");
}

/*-----------PROC. TO DO PROCESSING ---------------*/

void processing (void)
{
    /*  THE PROCESSING GETS DONE HERE !!!!!!! */

    printf("The number of iterations = %d\n",iter);
    for (count = 1; count <=iter ; count++)
        {
      printf("busy with iteration number %d\n",count);
      half_wsize = (winsize+1)/2 -1;
      win_off = ((int)(((float)winsize/2)+0.5));
            linenum = win_off;
            puts("filtering the image ...");
      i = (winsize*winsize);
            printf("block size = %d\n",i);

      while (linenum < (imglines - (win_off - 1)))
              {
                linepos =(win_off);
        filt_ptr = filtstart+(linenum*linesize) + linepos;
        arrayptr = arrstart+(linenum*linesize) + linepos;

        while (linepos < (linesize - (win_off) - 1))
                {
                    /* Hirosawa FILTER */
                    sum1 = 0;                /*calculate the mean and s
 => td dev */
                    sum2 = 0;
                    for (y=-half_wsize; y<=half_wsize ; y++)
                        for (x=-half_wsize ; x<=half_wsize ; x++)
                            {
          sum1 = sum1 + *(arrayptr + y*linesize + x);
          sum2=sum2+(*(arrayptr+y*linesize+x)-R)*(*(arrayptr+y*linesize+x)-R);
                            }
          R =(sum1/((float)(winsize*winsize)));
                    sigma = (unsigned char)sqrt((double)(sum2/(float)i))
 => ;

                    /* the hirosawa alg. */
                    if ((sigma/R) <= K)
                        *filt_ptr = R + G*(*arrayptr - R);
                    else
                        *filt_ptr = *arrayptr;

                    linepos++;
                    filt_ptr++;
                    arrayptr++;
                }
                linenum++;
              }

      /* now copy the filtered array back into the original
            for next iteration*/
             arrayptr = arrstart;
             filt_ptr = filtstart;
        for (x=0 ; x<imglines*linesize ; x++)
                {
                    *arrayptr = *filt_ptr;
                    arrayptr++;
                    filt_ptr++;
                }
        }
}

/* ----------PROC. TO WRITE IMAGE-----------*/

void write_image(void)
{
    /* NOW  WRITE TO THE FILE */

    filt_ptr = filtstart;
    arrayptr = arrstart;

    for (y=0; y<imglines; y++)
        {
      fwrite(filt_ptr,linesize*size,1,fp2);
      filt_ptr = filt_ptr + linesize;
        }
}

/*------------MAIN PROGRAM----------*/

int main()
{
    assignments();
    get_image();
    processing();
    write_image();

    puts("Written to file, program complete.");
    free(filt_ptr);
    free(arrayptr);
    free(filtstart);
    free(arrstart);
    fclose(fp2);
    fclose(fp1);
    return 0;
}
```

## F.2 Parameter Investigation

The parameters for the Hirosawa filter are window size, number of iterations, a threshold value ($K$) and a parameter which specifies the amount of speckle reduction ($g$). In the paper by Hirosawa and Kimura, the suggested values for $g$ are (when filtering a 1 look image) $g = 0.3$ or $g = 0.5$. Hirosawa states that these values yield 3 and 4 look images respectively. In order to obtain the maximum amount of speckle reduction, $g$ has been set equal to 0.5. Table F.1 gives ENL and edge measure values for the Hirosawa filtered image for various combinations of the input parameters. It was found that

Table F.1: ENL and edge values from Hirosawa filtered image for various parameter values.

| iterations | window size | K | edge measure | ENL |
|---|---|---|---|---|
| 1 | 3 | 5 | 0.42 | 5.54 |
| 1 | 5 | 5 | 0.38 | 8.01 |
| 1 | 7 | 5 | 0.36 | 9.36 |
| 1 | 9 | 5 | 0.35 | 10.12 |
| 1 | 7 | 1 | 0.37 | 9.36 |
| 1 | 7 | 2 | 0.36 | 9.36 |

more than 1 iteration caused the Hirosawa filter to blur edges excessively. Small variations in the $K$ parameter were found to have little effect on the performance of the filter. An increase in window size yields an increase in ENL and a decrease in edge measure. As a compromise between edge preservation and speckle reduction, the following parameters were chosen: $iterations = 1, window\ size = 5, K = 5$.

# Appendix G

# MAP Filter

The algorithm for the MAP filter is given in section 4.6. This appendix contains the listing of the source code for the filter as well as an investigation into the effects of varying the filter parameters.

## G.1 Source Code

The following two pages contain the source code for the MAP filter.

```c
/*=====================================================================

 program to implement Gamma-Gamma MAP filtering in C
 Single point filter for 8-bit multiple-look images!

 by: Mark Gebhardt, (c) University of Cape Town (March 1995)

 =====================================================================*/

/*---------------------------------------------------------------------

CLASS: Speckle filters

MODULE: map

INFO: The Maximum a Posteriori (MAP) filter is presented by Lopes et al
(lopes:93). The filter assumes a Gamma speckle model and a
Gamma reflectivity model as well as K-distributed multilook data.
This filter involves running an NXN window over the image and replacing
the central pixel according to the MAP model.

PARAMETERS: inname - name of input file
    outname - name of output file
    imglines - y dimension of input block
    linesize - x dimension of input block
    winsize - specifies size of square filtering window (winsizeXwinsize)(must b
  => e odd)
    iter - the number of iterations
            L - the equivalent number of looks in the unfiltered image (L >=
  =>  1)
            (default values assigned in code)

---------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>
#include "iofunct.h"

/*---------GLOBAL VARIABLE DECLARATIONS--------------*/

FILE *fp1,*fp2;
FILE *outfile,*specfile,*infile;
char  specname[80],inname[80],outname[80],ext[4];
int linesize,imglines,winsize,iter,L;
int x,y,i,half_wsize,win_off,linepos,linenum,imgpos,size,count;
unsigned char *filt_ptr,*filtstart;
unsigned char *arrayptr,*arrstart;
double R,sigma,alpha,fact,sum,temp;
unsigned long imgsize;
unsigned char lsb,msb;

/*--------PROCEDURE DECLARATIONS-----------------*/

void assignments(void);
void get_image(void);
void processing(void);
void write_image(void);

/*------------PROC. ASSIGNMENTS--------------------*/

void assignments(void)
{
     /* Open spec file */
     printf ("\nExtension for spec file (map.???) - max 3 characters : ");
     scanf ("%s", ext);
     sprintf(specname, "map.%s", ext);
     if ( (specfile = fopen (specname, "rt") ) == NULL )
       {
         printf ("ERROR: Specfile %s not found/opened\n", specname);
         exit(0);
       }

   /* Read processing specs *
    *************************/

    skipchar(specfile,'>');
    fscanf (specfile, "%d", &linesize);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &imglines);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &winsize);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &iter);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &L);
    skipchar(specfile,'>');
    GetString(specfile,inname);            /* Space taken care of */
    skipchar(specfile,'>');
    GetString(specfile,outname);           /* Space taken care of */

    fclose(specfile);

    imgsize = imglines*linesize;
    size = sizeof(unsigned char);
    if ((filt_ptr = (unsigned char *) calloc(imgsize,size)) == NULL )
        puts("can't make array");
    if ((arrayptr = (unsigned char *) calloc(imgsize,size)) == NULL )
        puts("can't make array");
    arrstart = arrayptr;
    filtstart = filt_ptr;

    /* OPEN INPUT AND OUTPUT IMAGE FILES */

    if ((fp1 = fopen(inname,"rb"))==NULL)
        {
            printf("cannot open input file\n");
            exit(1);
        }
    if ((fp2 = fopen(outname,"wb"))==NULL)
        {
            printf("cannot open output file\n");
            exit(1);
        }
    puts("Filtering SAR image, using single point Gamma-Gamma MAP filter");
}

/*------------PROC. TO READ IMAGE-----------------*/

void get_image(void)
{
    /* Read in the image */
    arrayptr = arrstart;

    for (y=0;y<imglines;y++)   /* read a line */
        {
      fread(arrayptr,linesize*size,1,fp1);
      arrayptr = arrayptr+linesize;
        }
    puts("Image read.");
}
```

```
/*------------PROC. TO DO PROCESSING---------------*/

void processing(void)
{
     printf("The effective number of looks, L = %d\n",L);

     /*  THE PROCESSING GETS DONE HERE !!!!!!! */

     printf("The number of iterations = %d\n",iter);
     for (count = 1; count <=iter ; count++)
         {
            printf("busy with iteration number %d\n",count);
       half_wsize = (winsize+1)/2-1;
       win_off = ((int)(((float)winsize/2)+0.5));
            linenum = win_off;
            puts("Busy filtering ...");
       i = (winsize*winsize);
            printf("block size = %d\n",i);

       while (linenum < (imglines - (win_off - 1)))
              {
                    linepos =(win_off);
         filt_ptr = filtstart+(linenum*linesize) + linepos;
         arrayptr = arrstart+(linenum*linesize) + linepos;

         while (linepos < (linesize - (win_off) - 1))
                    {
                          /* APPLY THE RELEVANT FILTER */
                          /* A SINGLE POINT GAMMA-GAMMA MAP FILTER */
                          sum = 0;    /*calculate the mean*/
                          for (y=-half_wsize; y<=half_wsize ; y++)
                              for (x=-half_wsize ; x<=half_wsize ; x++)
                                  {
          sum = sum + *(arrayptr + y*linesize + x);
                                  }
          R =(sum/((float)(winsize*winsize)));
                          sum = 0;     /*now find std. deviation*/
                          for (y=-half_wsize; y<=half_wsize ; y++)
                              for (x=-half_wsize ; x<=half_wsize ; x++)
                                  {
          sum=sum+(*(arrayptr+y*linesize+x)-R)*(*(arrayptr+y*linesize+x)-R);
                                  }
                          sigma = sqrt(sum/(float)i);
                          alpha = 1/((sigma/R)*(sigma/R));
                          fact = alpha-L-1;
                          temp=(1.5)*(fact*R+sqrt((R*R)*(fact*fact)+4*alpha*
 => L*(*arrayptr)*R))/(2*alpha);

                          if (temp>255.0)
                            *filt_ptr=255;
                          else if (temp<0.0)
                            *filt_ptr=0;
                          else if ((temp>0.0)&&(temp<255.0))
                            *filt_ptr=(unsigned char)temp;

                          linepos++;
                          filt_ptr++;
                          arrayptr++;
                    }
                linenum++;
              }

            /* now copy the filtered array back into the original for next ite
 => ration*/
            arrayptr = arrstart;
                 filt_ptr = filtstart;
            for (x=0 ; x<imglines*linesize ; x++)
                 {
                      *arrayptr = *filt_ptr;
                      arrayptr++;
                      filt_ptr++;
                 }
         }
}

/*---------------PROCEDURE TO WRITE IMAGE-------------*/

void write_image(void)
{
    /* NOW  WRITE TO THE FILE */

    filt_ptr = filtstart;
    arrayptr = arrstart;

    for (y=0; y<imglines; y++)
        {
      fwrite(filt_ptr,linesize*size,1,fp2);
      filt_ptr = filt_ptr + linesize;
        }
    puts("Written to file, program complete.");
}

/*------------MAIN PROGRAM-----------------------*/

int main()
{
    assignments();
    get_image();
    processing();
    write_image();

    free(filt_ptr);
    free(arrayptr);
    free(filtstart);
    free(arrstart);
    fclose(fp2);
    fclose(fp1);
    return 0;
}
```

## G.2 Parameter Investigation

The MAP filter has three parameters which must be specified. They are the window size, number of iterations and a parameter, $L$ which must be set equal to the number of looks of the unfiltered image. For the purposes of this investigation, a 3 look image has been used and $L$ has been set to three.

Table G.1: ENL and edge values from MAP filtered image for various parameter values.

| iterations | window size | edge measure | ENL |
|---|---|---|---|
| 1 | 3 | 0.42 | 5.52 |
| 1 | 5 | 0.40 | 7.19 |
| 1 | 7 | 0.38 | 7.93 |
| 1 | 9 | 0.37 | 8.31 |
| 2 | 7 | 0.29 | 27.49 |

The MAP filtering algorithm represents the classic trade off between geometric and radiometric resolutions. As the number of iterations and the window size increase, the ENL value increases and the edge measure decreases. It is worth noting that by iterating the filter twice, it is possible to increase the ENL from 7.93 to 27.49, this large increase is, however, accompanied by a sharp decrease in the edge measure value. In order to obtain the best possible edge preservation, the following filter parameters were chosen: $iterations = 1, window\ size = 3$.

# Appendix H

# Frost Filter

The algorithm for the Frost filter is described in Section 4.7. This appendix contains the source code as well as an investigation into the effects produced by varying the parameters of the Frost filter. The Frost weighting function of equation 4.7 is implemented in the function *get_weight* and is detailed in the next section.

## H.1 Source Code

The following 2 pages contain the source code for the Frost filter.

```
/*=====================================================================

 program to implement Frost filtering in C
 filter for 8-bit images!

 by: Mark Gebhardt, (c) University of Cape Town (March 1995)

 ====================================================================*/

/*---------------------------------------------------------------------

CLASS: Speckle Filters

MODULE: frost

INFO: Frost et al {frost:82} develop a statistical model for radar noise
and an image enhancement technique which is based on this model. The image
enhancement technique makes use of a minimum mean square error (MMSE) filter
which attempts to estimate the terrain backscatter from the image data.
The local mean and standard deviation are used to adapt the filter
so that MMSE estimates are produced inside locally homogeneous areas.

PARAMETERS: inname - name of input file
    outname - name of output file
    imglines - y dimension of input block
    linesize - x dimension of input block
    winsize - specifies size of square filtering window (must be odd)
    iter - the number of iterations
            K1 - normalising constant
            K2 - the filter parameter
            (default values assigned in code)

----------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>
#include "iofunct.h"

/*----------GLOBAL VARIABLE DECLARATIONS-------------*/

FILE *fp1,*fp2;
FILE *outfile,*specfile,*infile;
char  specname[80],inname[80],outname[80],ext[4];
int linesize,imglines,winsize,iter,K1,K2;
int x,y,i,half_wsize;
int win_off,linepos,linenum,imgpos,size,count;
unsigned char *arrayptr,*filt_ptr,*arrstart,*filtstart;
float R,sigma,sum1,sum2;
unsigned long imgsize;
unsigned char lsb,msb;

/*------------ FUNCTION DECLARATIONS ------------*/

float get_weight(int x,int y,float sigma,float mean);

/*----------PROCEDURE DEFINITIONS-------------------*/

void assignments(void);
void get_image(void);

void processing(void);
void write_image(void);

/*---------------ASSIGNMENTS----------------*/

void assignments(void)
{
        /* Open spec file */
    printf ("\nExtension for spec file (frost.???) - max 3 characters : ");
    scanf ("%s", ext);
    sprintf(specname, "frost.%s", ext);
    if ( (specfile = fopen (specname, "rt") ) == NULL )
      {
        printf ("ERROR: Specfile %s not found/opened\n", specname);
        exit(0);
      }

   /* Read processing specs *
    **************************/

    skipchar(specfile,'>');
    fscanf (specfile, "%d", &linesize);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &imglines);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &winsize);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &iter);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &K1);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &K2);
    skipchar(specfile,'>');
    GetString(specfile,inname);          /* Space taken care of */
    skipchar(specfile,'>');
    GetString(specfile,outname);         /* Space taken care of */

    fclose(specfile);

    /* initialise arrays */
    imgsize = imglines*linesize;
    size = sizeof(unsigned char);
    if ((filt_ptr = (unsigned char *) calloc(imgsize,size)) == NULL )
        puts("can't make array");
    if ((arrayptr = (unsigned char *) calloc(imgsize,size)) == NULL )
        puts("can't make array");
    arrstart = arrayptr;
    filtstart = filt_ptr;

    /* OPEN INPUT AND OUTPUT IMAGE FILES */

    if ((fp1 = fopen(inname,"rb"))==NULL)
        {
            printf("cannot open input file\n");
            exit(1);
        }
    if ((fp2 = fopen(outname,"wb"))==NULL)
        {
            printf("cannot open output file\n");
            exit(1);
        }
}

/*-----------PROC. TO GET IMAGE----------------*/

void get_image(void)
{
    /* Read in the image */
    arrayptr = arrstart;
```

```
    for (y=0;y<imglines;y++)   /* read a line */
        {
      fread(arrayptr,linesize*size,1,fp1);
      arrayptr = arrayptr+linesize;
        }
    puts("Image read.");
}

/*------------PROC. PROCESSING------------------*/

void processing(void)
{
  /*  THE PROCESSING GETS DONE HERE !!!!!!! */
    printf("The number of iterations = %d\n",iter);
    for (count = 1; count <=iter ; count++)
        {
      printf("busy with iteration number %d\n",count);
      half_wsize = (winsize+1)/2 -1;
      win_off = ((int)(((float)winsize/2)+0.5));
            linenum = win_off;
            puts("filtering the image ...");
    i = (winsize*winsize);
            printf("block size = %d\n",i);

      while (linenum < (imglines - (win_off - 1)))
                {
                    linepos =(win_off);
        filt_ptr = filtstart+(linenum*linesize) + linepos;
        arrayptr = arrstart+(linenum*linesize) + linepos;

        while (linepos < (linesize - (win_off) - 1))
            {
                            sum1 = 0.0;
                            sum2 = 0.0;

          for (y=-half_wsize; y<=half_wsize ; y++) /* find local mean and s.dev.
 => */
                                for (x=-half_wsize ; x<=half_wsize ; x++)
                                    {
          sum1 = sum1 + *(arrayptr + y*linesize + x);
          sum2=sum2+(*(arrayptr+y*linesize+x)-R)*(*(arrayptr+y*linesize+x)-R);
                                    }

          R =(sum1/((float)(winsize*winsize)));

                            sigma = (unsigned char)sqrt((double)(sum2/(float)i))
 => ;

                            /* multiply window pixels by their weighting factors
 =>  */
                            sum1 = 0;
                            for (y=-half_wsize; y<=half_wsize ; y++)
                                for (x=-half_wsize ; x<=half_wsize ; x++)
                                    {
          sum1=sum1+(*(arrayptr+y*linesize+x))*get_weight(x,y,sigma,R);
                                    }
                            /* check for image saturation */
                            if ((sum1/(float)i <= 255.0) && (sum1/(float)i >= 0)
 => )
                                *filt_ptr = (unsigned char)(sum1/(float)i);
                            if (sum1/(float)i < 0)
                                *filt_ptr = 0;
                            if (sum1/(float)i > 255)
                                *filt_ptr = 255;

                            linepos++;
                                filt_ptr++;
                                arrayptr++;
                            }
                        linenum++;
                }

        /* now copy the filtered array back into the original
            for next iteration*/
              arrayptr = arrstart;
              filt_ptr = filtstart;
        for (x=0 ; x<imglines*linesize ; x++)
                {
                    *arrayptr = *filt_ptr;
                    arrayptr++;
                    filt_ptr++;
                }
        }
}

/*-----------PROC. TO WRITE FILTERED IMAGE-----------*/

void write_image(void)
{
    /* NOW  WRITE TO THE FILE */

    filt_ptr = filtstart;
    arrayptr = arrstart;

    for (y=0; y<imglines; y++)
        {
      fwrite(filt_ptr,linesize*size,1,fp2);
      filt_ptr = filt_ptr + linesize;
        }
    puts("Written to file, program complete.");
}

/* ----------- MAIN PROGRAM ---------------------*/

int main()
{
    assignments();
    puts("Filtering SAR image, using Frost filter");
    get_image();
    processing();
    write_image();

    free(filt_ptr);
    free(arrayptr);
    free(filtstart);
    free(arrstart);
    fclose(fp2);
    fclose(fp1);
    return 0;
}

/*---------------------function: get_weight------------------*/

float get_weight(int x,int y,float sigma,float mean)
{
    float m;

    m = K1 * exp( -K2 * (sigma/mean) * sqrt(x*x + y*y) );
    return m;
}
```

IC/14PRIN

## H.2 Parameter Investigation

The parameters which can be specified for the Frost filter are window size, number of iterations, *K1* and *K2*. *K1* and *K2* are parameters which influence the weighting and shape of the Frost function. *K1* is an arbitrary scaling constant which should be adjusted to give an acceptable image mean value and *K2* is a factor which influences the width of the Frost function. Table H.1 gives edge measure and ENL values for various values of the input parameters. It can be observed that the ENL value varies from 3.08 to 4.41 as the edge

Table H.1: ENL and edge values from Frost filtered image for various parameter values.

| iterations | window size | K2 | edge measure | ENL |
|---|---|---|---|---|
| 1 | 5 | 5 | 0.44 | 4.41 |
| 1 | 7 | 5 | 0.51 | 4.28 |
| 1 | 7 | 7 | 0.52 | 3.48 |
| 1 | 7 | 10 | 0.52 | 3.08 |
| 1 | 9 | 5 | 0.47 | 4.34 |

measure changes from 0.52 to 0.44. The nature of the Frost filter makes it resistant to changes in window size, as its adaptivity allows it to change its width, regardless of window size. Over a small range, the filter parameter, $K2$ has little or no effect on the edge measure, but does effect the ENL value. An increase in $K2$ yields a decrease in ENL. It was found that more iterations caused the Frost filter to blur edges excessively. As a compromise between edge preservation and speckle reduction, the following parameters were chosen for the Frost filter: $iterations = 1, window\ size = 9, K2 = 5$.

# Appendix I

# MHR Filter

The algorithm for the Maximum Homogeneous Region filter is described in Section 4.8. This appendix contains an elaboration on the algorithm, a listing of the source code and an investigation of the effect of varying the filter parameters.

## I.1 Algorithm

The algorithm for the MHR filter has been modified slightly and the flowchart of the new algorithm is given in Figure I.1. An algorithm has been developed for the block labelled *semi-window configuration* and this is briefly described in Section 4.8. The local statistics filtering was performed by a Frost filter, as described elsewhere in this thesis.

## I.2 Source Code

The following 8 pages contain the source code for the MHR filter.

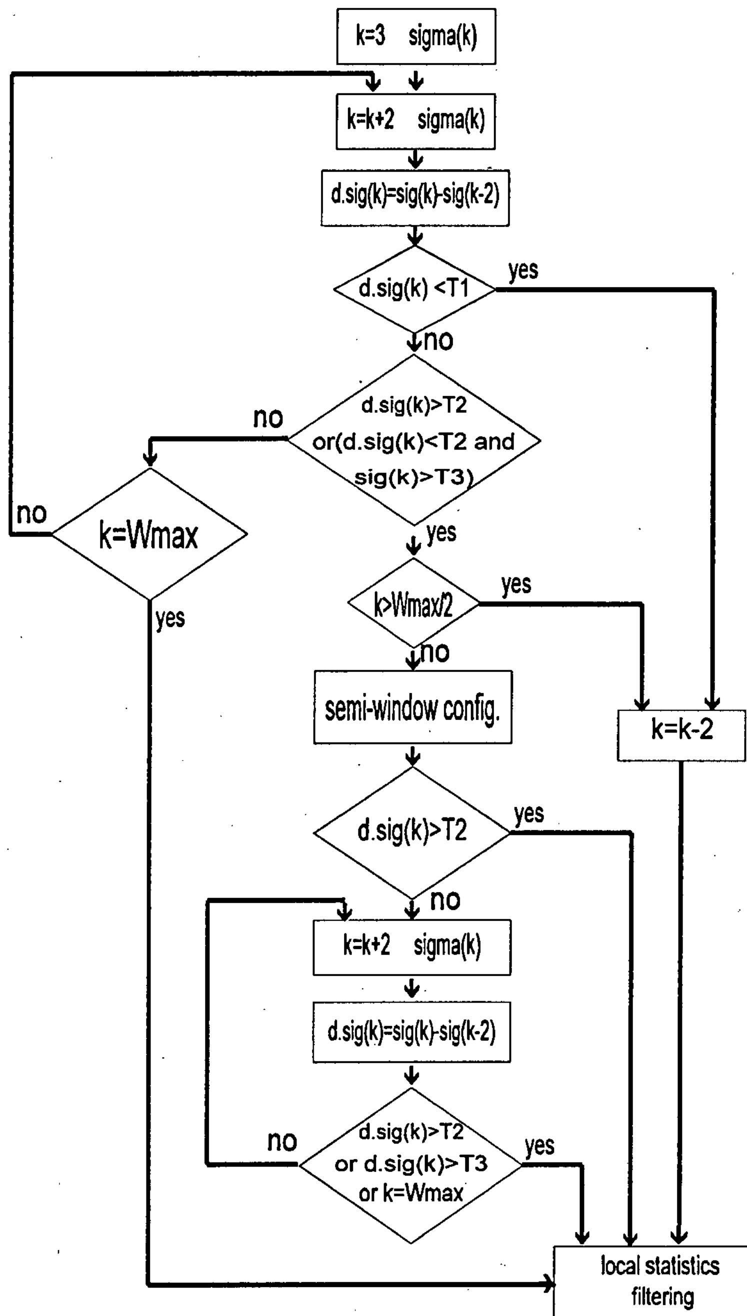


Figure I.1: Flow chart of the modified MHR algorithm

```
/*======================================================================

 implementation of Maximum Homogeneous Region (MHR) filter

by:  Mark Gebhardt, (c) University of Cape Town  (April 1995)

 ======================================================================*/

/*----------------------------------------------------------------------

CLASS:  Speckle Filters

MODULE: mhr

PARAMETERS:   inname - name of input file
    outname - name of output file
    imglines - y dimension of input block
    linesize - x dimension of input block
                T1,T2,T3 - filter thresholds
    wmax - maximum window size
                (default values assigned in code)

------------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>
#include "iofunct.h"

/*----------CONSTANT  DEFINITIONS--------------*/

#define TRUE 1
#define FALSE 0

/*------------- global variable declarations --------------------------*/

FILE *fp1,*fp2;
FILE *outfile,*specfile,*infile;
char  specname[80],inname[80],outname[80],ext[4];
int linesize,imglines,wmax,T1,T2,T3;
int x,y,i,k,got_region,cfg,dsigma,halfwin;
int win_off,k,linepos,linenum,imgpos,size;
unsigned char *arrayptr,*filt_ptr,*filt_line_ptr;
unsigned char *arrstart,*filtstart,*arr_line_ptr;
unsigned long imgsize,sum;
unsigned char sigma1, sigma2, mean_val;
unsigned char lsb,msb;
unsigned char mask11[5][5],mask12[5][5],mask13[5][5],mask14[5][5],mask15[5][5],m
  => ask16[5][5],mask17[5][5],mask18[5][5];
unsigned char mask21[7][7],mask22[7][7],mask23[7][7],mask24[7][7],mask25[7][7],m
  => ask26[7][7],mask27[7][7],mask28[7][7];
unsigned char mask31[9][9],mask32[9][9],mask33[9][9],mask34[9][9],mask35[9][9],m
  => ask36[9][9],mask37[9][9],mask38[9][9];

/*----------------- procedure and function declarations ----------------*/

void assignments(void);
void get_image(void);
void processing(void);
void put_image(void);
void replace_pixel(void);
void get_mhr(void);
void semi_win_cfg(void);
float std_dev(int win_dim, unsigned char *data_ptr, int cfg);
float mean(int win_dim, unsigned char *data_ptr, int cfg);
unsigned char dofrost(unsigned char *data_ptr,int win_size,int cfg);
float get_weight(int x,int y,unsigned char sigma,unsigned char mean);


/*-------------procedure to assign and initialise variables etc ---------*/

void assignments()
{
     /* Open spec file */
     printf ("\nExtension for spec file (mhr.???) - max 3 characters : ");
     scanf ("%s", ext);
     sprintf(specname, "mhr.%s", ext);
     if ( (specfile = fopen (specname, "rt") ) == NULL )
       {
         printf ("ERROR: Specfile %s not found/opened\n", specname);
         exit(0);
       }

   /* Read processing specs *
    *************************/

    skipchar(specfile,'>');
    fscanf (specfile, "%d", &linesize);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &imglines);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &wmax);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &T1);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &T2);
    skipchar(specfile,'>');
    fscanf (specfile, "%d", &T3);
    skipchar(specfile,'>');
    GetString(specfile,inname);           /* Space taken care of */
    skipchar(specfile,'>');
    GetString(specfile,outname);          /* Space taken care of */

    fclose(specfile);

   size = sizeof(unsigned char);
   imgsize = imglines*linesize;
   if ((filt_ptr = (unsigned char *) calloc(imgsize,size)) == NULL )
     puts("can't make array");
   if ((arrayptr = (unsigned char *) calloc(imgsize,size)) == NULL )
     puts("can't make array");
   arrstart = arrayptr;
   filtstart = filt_ptr;

   /* define 24 masks for semi-windows */
   k = 5;
   for (y = 0; y < k; y++)        /* 5X5 masks */
       for (x = 0; x < k; x++)
          {
               mask11[x][y] = 1;
               mask12[x][y] = 1;
               mask13[x][y] = 1;
               mask14[x][y] = 1;
               mask15[x][y] = 1;
               mask16[x][y] = 1;
               mask17[x][y] = 1;
               mask18[x][y] = 1;
               if (x == 0)
                   mask11[x][y] = 0;
               if ((x < 3) && ((y-x) >= 2))
```

IC/14PRINT

```
                mask12[x][y] = 0;
            if (y == k-1)
                mask13[x][y] = 0;
            if ((y>1) && ((x+y) >= 6))
                mask14[x][y] = 0;
            if (x == k-1)
                mask15[x][y] = 0;
            if ((y<3) && ((x-y) >= 2))
                mask16[x][y] = 0;
            if (y == 0)
                mask17[x][y] = 0;
            if ((x<3) && ((x+y) <= 2))
                mask18[x][y] = 0;
        }
  k = 7;
  for (y = 0; y < k; y++)            /* 7x7 masks */
      for (x = 0; x < k; x++)
        {
            mask21[x][y] = 1;
            mask22[x][y] = 1;
            mask23[x][y] = 1;
            mask24[x][y] = 1;
            mask25[x][y] = 1;
            mask26[x][y] = 1;
            mask27[x][y] = 1;
            mask28[x][y] = 1;
            if (x == 0)
                mask21[x][y] = 0;
            if ((x < 3) && ((y-x) >= 4))
                mask22[x][y] = 0;
            if (y == k-1)
                mask23[x][y] = 0;
            if ((y>3) && ((x+y) >= 10))
                mask24[x][y] = 0;
            if (x == k-1)
                mask25[x][y] = 0;
            if ((y<3) && ((x-y) >= 4))
                mask26[x][y] = 0;
            if (y == 0)
                mask27[x][y] = 0;
            if ((x<3) && ((x+y) <= 2))
                mask28[x][y] = 0;
        }
  k = 9;
  for (y = 0; y < k; y++)           /* 9x9 masks */
      for (x = 0; x < k; x++)
        {
            mask31[x][y] = 1;
            mask32[x][y] = 1;
            mask33[x][y] = 1;
            mask34[x][y] = 1;
            mask35[x][y] = 1;
            mask36[x][y] = 1;
            mask37[x][y] = 1;
            mask38[x][y] = 1;
            if (x == 0)
                mask31[x][y] = 0;
            if ((x < 3) && ((y-x) >= 6))
                mask32[x][y] = 0;
            if (y == k-1)
                mask33[x][y] = 0;
            if ((y>5) && ((x+y) >= 14))
                mask34[x][y] = 0;
            if (x == k-1)
                mask35[x][y] = 0;
            if ((y<3) && ((x-y) >= 6))
                mask36[x][y] = 0;
            if (y == 0)
                mask37[x][y] = 0;
            if ((x<3) && ((x+y) <= 2))
                mask38[x][y] = 0;
        }
}

/*----------------- procedure to read in the image -----------------------*/

void get_image()
{
   /* OPEN INPUT AND OUTPUT IMAGE FILES */
  if ((fp1 = fopen(inname,"rb"))==NULL)
    {
      printf("cannot open input file\n");
      exit(1);
    }
  if ((fp2 = fopen(outname,"wb"))==NULL)
    {
      printf("cannot open output file\n");
      exit(1);
    }
  puts("Filtering SAR image, using MHR filter");

  /* Read in the image */
  arrayptr = arrstart;
  for (y=0;y<imglines;y++)   /* read a line */
    {
      i=0;
      fread(arrayptr,linesize*size,1,fp1);
      arrayptr = arrayptr+linesize;
    }
  puts("Image read.");
}

/*----------- procedure to determine new value of the pixel -----------*/

void replace_pixel()
{
  *filt_ptr = dofrost(arrayptr,k,cfg);
}

/*----------- procedure to determine the edge orientation --------------*/
                 /* and assign a configuration number. */

void semi_win_cfg(void)
{
  float dx,dy,grad;

  cfg = 0;
  halfwin = ((int)(((float)k/2)-0.5));

  if (dsigma > T2) /* pixel close to step edge     -----     case 1       */
    {
      for (i = -2;i<=2;i++) /* use  row of 5 pixels to calculate the slope direc
  => tion */
        {
   dx =(float)(*(arrayptr+halfwin+i*linesize) - *(arrayptr-halfwin+i*linesize))
  => ;
   dy =(float)(*(arrayptr+linesize*halfwin+i)-*(arrayptr-linesize*halfwin+i));
        }
      grad = dy/dx;

      if (k==5)
        cfg=10;
```

```
    else if (k==7)
      cfg=20;
    else if (k==9)
      cfg=30;

    if (grad < -1.5)
      {
  if (*(arrayptr)<*(arrayptr-linesize*halfwin))
          cfg++;
  if (*(arrayptr)>*(arrayptr-linesize*halfwin))
          cfg = cfg + 5;
      }
    else if ((-0.5 > grad) && (grad >= -1.5))
      {
        if (dx >= 0)
          {
            if (*(arrayptr+halfwin)>*(arrayptr))
              cfg = cfg+2;
            else cfg = cfg+6;
          }
        if (dx < 0)
          {
            if (*(arrayptr)>*(arrayptr+halfwin))
              cfg = cfg+2;
            else cfg = cfg+6;
          }
      }
    else if ((-0.5 <= grad) && (grad < 0.5))
      {
        if (dx >= 0)
          {
            if (*(arrayptr+halfwin)>*(arrayptr))
              cfg = cfg+3;
            else cfg = cfg+7;
          }
        if (dx < 0)
          {
            if (*(arrayptr+halfwin)<*(arrayptr))
              cfg = cfg + 3;
            else cfg = cfg+7;
          }
      }
    else if ((0.5 <= grad) && (grad < 1.5))
      {
        if (dx >= 0)
          {
            if (*(arrayptr+halfwin)>*(arrayptr))
              cfg = cfg+4;
            else cfg = cfg + 8;
          }
        if (dx < 0)
          {
            if (*(arrayptr+halfwin)<*(arrayptr))
              cfg = cfg+4;
            else cfg = cfg+8;
          }
      }
    else if (grad >= 1.5)
      {
  if (*(arrayptr)>*(arrayptr-linesize*halfwin))
          cfg++;
  if (*(arrayptr)<*(arrayptr-linesize*halfwin))
          cfg = cfg + 5;
      }
  }
```

```
if (((dsigma <=  T2) && (sigma2>T3))&&(sigma2>T3))  /* pixel on step edge */
  {
    dx =(float)(*(arrayptr+1)+ *(arrayptr+2) -(*(arrayptr-1)+ *(arrayptr-2)));
    dy =(float)(*(arrayptr+linesize)+ *(arrayptr+linesize*2)-(*(arrayptr-lines
=> ize)+*(arrayptr-linesize*2)));
    grad = dy/dx;

    if (grad < -1.5)
      {
  if (*(arrayptr)>*(arrayptr+linesize))
          cfg=1;
        else cfg = 5;
      }
    else if ((-0.5 > grad) && (grad >= -1.5))
      {
        if (dx >= 0)
          {
            if (*(arrayptr)>*(arrayptr-1))
              cfg = 2;
            else cfg = 6;
          }
        if (dx < 0)
          {
            if (*(arrayptr)<*(arrayptr-1))
              cfg = 2;
            else cfg = 6;
          }
      }
    else if ((-0.5 <= grad) && (grad < 0.5))
      {
        if (dx >= 0)
          {
            if (*(arrayptr)>*(arrayptr-1))
              cfg = 3;
            else cfg = 7;
          }
        if (dx < 0)
          {
            if (*(arrayptr)<*(arrayptr-1))
              cfg = 3;
            else cfg = 7;
          }
      }
    else if ((0.5 <= grad) && (grad < 1.5))
      {
        if (dx >= 0)
          {
            if (*(arrayptr)>*(arrayptr-1))
              cfg = 4;
            else cfg = 8;
          }
        if (dx < 0)
          {
            if (*(arrayptr)<*(arrayptr-1))
              cfg = 4;
            else cfg = 8;
          }
      }
    else if (grad >= 1.5)
      {
  if (*(arrayptr)<*(arrayptr+linesize))
          cfg=1;
        else cfg = 5;
      }
  }
}
```

```
/*----------- procedure to implement the filter algorithm for 1 pixel ----------
  => ---*/

void get_mhr()
{
  got_region = FALSE;
  cfg = 0;

  k=3;
  sigma2 = (unsigned char)std_dev(k,arrayptr,cfg);

  do
    {
      if ((linepos >= k*size) && (linenum >= k))
        {
          do
            {
              sigma1 = sigma2;
              k=k+2;
              sigma2 = (unsigned char)std_dev(k,arrayptr,cfg);

              dsigma = sigma2 - sigma1;

              if (dsigma < -T1)
                {
                  k=k-2;
                  got_region = TRUE;
                }
            }
     while (!(k==wmax) && !((dsigma>T2) || ((dsigma<T2) && (sigma2>T3))) && !(got
  => _region));

     if (!((dsigma>T2) || ((dsigma<T2) && (sigma2>T3))) && (k==wmax) && !(got_reg
  => ion))
              got_region = TRUE;

          if (((dsigma>T2) || ((dsigma <= T2) && (sigma2>T3))) && !(got_region))
            {
        if (k>wmax/2)
                {
                  k=k-2;
                  got_region = TRUE;
                }
              else
                {
                  semi_win_cfg();

                  if (dsigma > T2)          /* this mod to avoid huge window size
  => s */
                    got_region = TRUE;

                  if ((linepos >= k*size) && (linenum >= k) && (!got_region))
                    {
                      do
                        {
                          k=k+2;
                          sigma1 = sigma2;
                          sigma2 = (unsigned char)std_dev(k,arrayptr,cfg);

                          dsigma = sigma2-sigma1;
                        }
          while (!((dsigma>T2) || (sigma2>T3) || (k==wmax)));
          if ((dsigma>T2) || (sigma2>T3) || (k==wmax))
                        got_region = TRUE;
                    }
                  else (got_region = TRUE);
                }
            }
        }
      else (got_region = TRUE);
    }
  while (!got_region);

  if (got_region == TRUE)
    replace_pixel();
}

/*-------------- procedure to perform the processing ------------------*/

void processing()
{
  /*  THE PROCESSING GETS DONE HERE !!!!!!! */
  k = 3;

  win_off = ((int)(((float)k/2)+0.5));
  linenum = win_off;
  puts("Busy filtering ...");

  while (linenum < (imglines - (win_off - 1)))
    {
      printf("line = %d\n",linenum);
      filt_line_ptr = filtstart+(linenum*linesize);
      arr_line_ptr = arrstart+(linenum*linesize);
      linepos = win_off;
      filt_ptr = filt_line_ptr + linepos;
      arrayptr = arr_line_ptr + linepos;
      while (linepos < (linesize - (win_off - 1)))
        {
          get_mhr();
          linepos++;
          filt_ptr++;
          arrayptr++;
        }
      linenum++;
    }
}

/*------------- procedure to write the file to disk ---------------------*/

void put_image()
{
  /* NOW  WRITE TO THE FILE */

  filt_ptr = filtstart;
  arrayptr = arrstart;

  for (y=0; y<imglines; y++)
    {
      fwrite(filt_ptr,linesize*size,1,fp2);
      filt_ptr = filt_ptr + linesize;
    }
  puts("Written to file, program complete.");
}

/*---------------------- The MAIN PROGRAM -----------------------------*/

int main()
{
  assignments();
  get_image();
```

```
  processing();
  put_image();

  free(filt_ptr);
  free(arrayptr);
  fclose(fp2);
  fclose(fp1);
  return 0;
}


/*----------- function to determine the local mean value ----------------*/

float mean(int win_dim, unsigned char *data_ptr, int cfg)
{
  int limit,a,b,c,d,fact;
  double integer;
  unsigned char tmparr[9][9],temp;
  long int sum;
  float temp2;

  limit = (int)(((float)win_dim)/2 - 0.5);
  a = -limit;
  b = limit;
  c = -limit;
  d = limit;

/*  for case 1:
          - read vals into small array
          - multiply by mask (0000111111) etc
          - add rest and divide by number.
                NB need 8 masks for each of 5,7,9 - these defined in 'assignmen
  => ts'

     for case 2:
     calculate w/o masks for k=5,7,9,11,13,15,17,19,21
*/
  sum = 0;
  fact = 0;

  if ((cfg > 10) && (cfg < 40))    /* this is case 1 */
      {
          /* copy values into tmparr */
        for (y = -limit; y <= limit; y++)
          for (x = -limit; x <= limit; x++)
      tmparr[x+limit][y+limit] = *(data_ptr + y + x*linesize);

        /* multiply tmparr by relevant mask as shown by cfg */

        for (y=0; y<win_dim; y++)
          for (x=0; x<win_dim; x++)
            {
              switch (cfg)
                {
                case 11: tmparr[x][y] = tmparr[x][y] * mask11[x][y];
                  break;
                case 12: tmparr[x][y] = tmparr[x][y] * mask12[x][y];
                  break;
                case 13: tmparr[x][y] = tmparr[x][y] * mask13[x][y];
                  break;
                case 14: tmparr[x][y] = tmparr[x][y] * mask14[x][y];
                  break;
                case 15: tmparr[x][y] = tmparr[x][y] * mask15[x][y];
                  break;
                case 16: tmparr[x][y] = tmparr[x][y] * mask16[x][y];
                  break;
                case 17: tmparr[x][y] = tmparr[x][y] * mask17[x][y];
                  break;
                case 18: tmparr[x][y] = tmparr[x][y] * mask18[x][y];
                  break;
                case 21: tmparr[x][y] = tmparr[x][y] * mask21[x][y];
                  break;
                case 22: tmparr[x][y] = tmparr[x][y] * mask22[x][y];
                  break;
                case 23: tmparr[x][y] = tmparr[x][y] * mask23[x][y];
                  break;
                case 24: tmparr[x][y] = tmparr[x][y] * mask24[x][y];
                  break;
                case 25: tmparr[x][y] = tmparr[x][y] * mask25[x][y];
                  break;
                case 26: tmparr[x][y] = tmparr[x][y] * mask26[x][y];
                  break;
                case 27: tmparr[x][y] = tmparr[x][y] * mask27[x][y];
                  break;
                case 28: tmparr[x][y] = tmparr[x][y] * mask28[x][y];
                  break;
                case 31: tmparr[x][y] = tmparr[x][y] * mask31[x][y];
                  break;
                case 32: tmparr[x][y] = tmparr[x][y] * mask32[x][y];
                  break;
                case 33: tmparr[x][y] = tmparr[x][y] * mask33[x][y];
                  break;
                case 34: tmparr[x][y] = tmparr[x][y] * mask34[x][y];
                  break;
                case 35: tmparr[x][y] = tmparr[x][y] * mask35[x][y];
                  break;
                case 36: tmparr[x][y] = tmparr[x][y] * mask36[x][y];
                  break;
                case 37: tmparr[x][y] = tmparr[x][y] * mask37[x][y];
                  break;
                case 38: tmparr[x][y] = tmparr[x][y] * mask38[x][y];
                  break;
                }
              /* add values in tmparr */
              sum = sum + tmparr[x][y];
            }
      }

  if (cfg < 10) /* this is case 2 */
    {
      switch (cfg)
        {
        case 0 :  for (y=a; y<=b; y++)
            for (x=c; x<=d; x++)
      sum = sum + *(data_ptr + y*linesize + x);
        break;
        case 1 :    for (y=a; y<=0; y++)
            for (x=c; x<=d; x++)
      sum = sum + *(data_ptr + y*linesize + x);
        break;
        case 2 :    for (y=a; y<=b; y++)
            for (x=y; x<=d; x++)
      sum = sum + *(data_ptr + y*linesize + x);
        break;
        case 3 :    for (y=a; y<=b; y++)
            for (x=0; x<=d; x++)
      sum = sum + *(data_ptr + y*linesize + x);
        break;
        case 4 :    for (y=a; y<=b; y++)
            for (x=-y; x<=d; x++)
      sum = sum + *(data_ptr + y*linesize + x);
        break;
```

IC/14PRINT

```
        case 5 :    for (y=0; y<=b; y++)
          for (x=c; x<=d; x++)
      sum = sum + *(data_ptr + y*linesize + x);
        break;
        case 6 :    for (y=a; y<=b; y++)
          for (x=c; x<=y; x++)
      sum = sum + *(data_ptr + y*linesize + x);
        break;
        case 7 :    for (y=a; y<=b; y++)
          for (x=c; x<=0; x++)
      sum = sum + *(data_ptr + y*linesize + x);
        break;
        case 8 :    for (y=a; y<=b; y++)
          for (x=c; x<=-y; x++)
      sum = sum + *(data_ptr + y*linesize + x);
        break;
        }
    }

  if (cfg == 0)
    {
      fact = win_dim*win_dim;
    }
  if ((cfg < 10) && (cfg != 0))
    {
      for (i=win_dim; i>=0; i--)
        fact = fact + i;
    }
  if (cfg > 10)
    {
      temp2 = modf((float)cfg/2,&integer);
      if (temp2 == 0.5)
        fact = win_dim;
      else fact = 6;
    }

  temp = ((float)sum/fact);
  if (temp < 1)
    temp = 1;

  return (temp);
}

/*----------- function to determine the local standard deviation ---------*/

float std_dev(int win_dim, unsigned char *data_ptr, int cfg)
{
  int lim,a,b,c,d,fact;
  unsigned char mean_val, tmparr[9][9];
  double integer;
  long int sum;
  float temp2;

  lim = (int)((float)win_dim/2 - .5);

  a = -lim;
  b = lim;
  c = -lim;
  d = lim;

  sum = 0;      /*now find std. deviation*/
  fact = 0;

  mean_val = (unsigned char)mean(win_dim,data_ptr,cfg);

  if ((cfg > 10) && (cfg < 40))    /* this is case 1 */
    {
     /* copy values into tmparr */

     for (y = -lim; y <= lim; y++)
       for (x = -lim; x <= lim; x++)
    tmparr[x+lim][y+lim] = *(data_ptr + y + x*linesize);

     /* multiply tmparr by relevant mask as shown by cfg */

     for (y=0; y<win_dim; y++)
       for (x=0; x<win_dim; x++)
         {
           switch (cfg)
             {
             case 11: tmparr[x][y] = tmparr[x][y] * mask11[x][y];
               break;
             case 12: tmparr[x][y] = tmparr[x][y] * mask12[x][y];
               break;
             case 13: tmparr[x][y] = tmparr[x][y] * mask13[x][y];
               break;
             case 14: tmparr[x][y] = tmparr[x][y] * mask14[x][y];
               break;
             case 15: tmparr[x][y] = tmparr[x][y] * mask15[x][y];
               break;
             case 16: tmparr[x][y] = tmparr[x][y] * mask16[x][y];
               break;
             case 17: tmparr[x][y] = tmparr[x][y] * mask17[x][y];
               break;
             case 18: tmparr[x][y] = tmparr[x][y] * mask18[x][y];
               break;
             case 21: tmparr[x][y] = tmparr[x][y] * mask21[x][y];
               break;
             case 22: tmparr[x][y] = tmparr[x][y] * mask22[x][y];
               break;
             case 23: tmparr[x][y] = tmparr[x][y] * mask23[x][y];
               break;
             case 24: tmparr[x][y] = tmparr[x][y] * mask24[x][y];
               break;
             case 25: tmparr[x][y] = tmparr[x][y] * mask25[x][y];
               break;
             case 26: tmparr[x][y] = tmparr[x][y] * mask26[x][y];
               break;
             case 27: tmparr[x][y] = tmparr[x][y] * mask27[x][y];
               break;
             case 28: tmparr[x][y] = tmparr[x][y] * mask28[x][y];
               break;
             case 31: tmparr[x][y] = tmparr[x][y] * mask31[x][y];
               break;
             case 32: tmparr[x][y] = tmparr[x][y] * mask32[x][y];
               break;
             case 33: tmparr[x][y] = tmparr[x][y] * mask33[x][y];
               break;
             case 34: tmparr[x][y] = tmparr[x][y] * mask34[x][y];
               break;
             case 35: tmparr[x][y] = tmparr[x][y] * mask35[x][y];
               break;
             case 36: tmparr[x][y] = tmparr[x][y] * mask36[x][y];
               break;
             case 37: tmparr[x][y] = tmparr[x][y] * mask37[x][y];
               break;
             case 38: tmparr[x][y] = tmparr[x][y] * mask38[x][y];
               break;
             }
           /* add values in tmparr */
           sum = sum + (tmparr[x][y]-mean_val)*(tmparr[x][y]-mean_val);
         }
```

1C/14PRINT

```
  }

if (cfg < 10) /* this is case 2 */
  {
    switch (cfg)
      {
      case 0 :    for (y=a ; y<=b; y++)
        for (x=c; x<=d; x++)
    sum = sum + (((int)*(data_ptr+y*linesize+x)-mean_val)*((int)*(data_ptr+y*l
=> inesize+x)-mean_val));
      break;
      case 1 :    for (y=a; y<=0; y++)
        for (x=c; x<=d; x++)
    sum = sum + (((int)*(data_ptr+y*linesize+x)-mean_val)*((int)*(data_ptr+y*l
=> inesize+x)-mean_val));
      break;
      case 2 :    for (y=a; y<=b; y++)
        for (x=y; x<=d; x++)
    sum = sum + (((int)*(data_ptr+y*linesize+x)-mean_val)*((int)*(data_ptr+y*l
=> inesize+x)-mean_val));
      break;
      case 3 :    for (y=a; y<=b; y++)
        for (x=0; x<=d; x++)
    sum = sum + (((int)*(data_ptr+y*linesize+x)-mean_val)*((int)*(data_ptr+y*l
=> inesize+x)-mean_val));
      break;
      case 4 :    for (y=a; y<=b; y++)
        for (x=-y; x<=d; x++)
    sum = sum + (((int)*(data_ptr+y*linesize+x)-mean_val)*((int)*(data_ptr+y*l
=> inesize+x)-mean_val));
      break;
      case 5 :    for (y=0; y<=b; y++)
        for (x=c; x<=d; x++)
    sum = sum + (((int)*(data_ptr+y*linesize+x)-mean_val)*((int)*(data_ptr+y*l
=> inesize+x)-mean_val));
      break;
      case 6 :    for (y=a; y<=b; y++)
        for (x=c; x<=y; x++)
    sum = sum + (((int)*(data_ptr+y*linesize+x)-mean_val)*((int)*(data_ptr+y*l
=> inesize+x)-mean_val));
      break;
      case 7 :    for (y=a; y<=b; y++)
        for (x=c; x<=0; x++)
    sum = sum + (((int)*(data_ptr+y*linesize+x)-mean_val)*((int)*(data_ptr+y*l
=> inesize+x)-mean_val));
      break;
      case 8 :    for (y=a; y<=b; y++)
        for (x=c; x<=-y; x++)
    sum = sum + (((int)*(data_ptr+y*linesize+x)-mean_val)*((int)*(data_ptr+y*l
=> inesize+x)-mean_val));
      break;
      }
  }

if (cfg == 0)
  {
    fact = win_dim*win_dim;
  }
if ((cfg < 10) && (cfg != 0))
  {
    for (i=win_dim; i>=0; i--)
      fact = fact + i;
  }
if (cfg > 10)
  {
    temp2 = modf((float)cfg/2,&integer);
```

```
      if (temp2 == 0.5)
        fact = win_dim;
      else fact = 6;
    }

  return (sqrt(((float)sum)/fact));

}

/*--------------function  dofrost------------------*/

unsigned char dofrost(unsigned char* data_ptr,int win_dim,int cfg)
{
  int a,b,c,d,lim,fact;
  unsigned char sigma,mean_val,tmparr[9][9];
  float temp2;
  double integer,sum,temp;

  sigma = (unsigned char)std_dev(win_dim,data_ptr,cfg);
  mean_val = (unsigned char)mean(win_dim,data_ptr,cfg);
  lim = (int)((float)win_dim/2 - .5);
  a = -lim;
  b = lim;
  c = -lim;
  d = lim;

  sum = 0;
  fact = 0;

  if ((cfg > 10) && (cfg < 40))   /* this is case 1 */
    {
      /* copy values into tmparr */

      for (y = -lim; y <= lim; y++)
        for (x = -lim; x <= lim; x++)
    tmparr[x+lim][y+lim] = *(data_ptr + y + x*linesize);

      /* multiply tmparr by relevant mask as shown by cfg */

      for (y=0; y<win_dim; y++)
        for (x=0; x<win_dim; x++)
          {
            switch (cfg)
              {
              case 11: tmparr[x][y] = tmparr[x][y] * mask11[x][y];
                break;
              case 12: tmparr[x][y] = tmparr[x][y] * mask12[x][y];
                break;
              case 13: tmparr[x][y] = tmparr[x][y] * mask13[x][y];
                break;
              case 14: tmparr[x][y] = tmparr[x][y] * mask14[x][y];
                break;
              case 15: tmparr[x][y] = tmparr[x][y] * mask15[x][y];
                break;
              case 16: tmparr[x][y] = tmparr[x][y] * mask16[x][y];
                break;
              case 17: tmparr[x][y] = tmparr[x][y] * mask17[x][y];
                break;
              case 18: tmparr[x][y] = tmparr[x][y] * mask18[x][y];
                break;
              case 21: tmparr[x][y] = tmparr[x][y] * mask21[x][y];
                break;
              case 22: tmparr[x][y] = tmparr[x][y] * mask22[x][y];
                break;
              case 23: tmparr[x][y] = tmparr[x][y] * mask23[x][y];
                break;
```

IC/14PRIN

```
        case 24: tmparr[x][y] = tmparr[x][y] * mask24[x][y];
          break;
        case 25: tmparr[x][y] = tmparr[x][y] * mask25[x][y];
          break;
        case 26: tmparr[x][y] = tmparr[x][y] * mask26[x][y];
          break;
        case 27: tmparr[x][y] = tmparr[x][y] * mask27[x][y];
          break;
        case 28: tmparr[x][y] = tmparr[x][y] * mask28[x][y];
          break;
        case 31: tmparr[x][y] = tmparr[x][y] * mask31[x][y];
          break;
        case 32: tmparr[x][y] = tmparr[x][y] * mask32[x][y];
          break;
        case 33: tmparr[x][y] = tmparr[x][y] * mask33[x][y];
          break;
        case 34: tmparr[x][y] = tmparr[x][y] * mask34[x][y];
          break;
        case 35: tmparr[x][y] = tmparr[x][y] * mask35[x][y];
          break;
        case 36: tmparr[x][y] = tmparr[x][y] * mask36[x][y];
          break;
        case 37: tmparr[x][y] = tmparr[x][y] * mask37[x][y];
          break;
        case 38: tmparr[x][y] = tmparr[x][y] * mask38[x][y];
          break;
        }
      /* add values in tmparr */
      temp2 = (float)tmparr[x][y]*get_weight(x-lim,y-lim,sigma,mean_val);
      if (temp2 > 255)
        temp2 = 255;
      else if (temp2 < 0)
        temp2 = 0;
      sum = sum + temp2;
    }
  }

if (cfg < 10) /* this is case 2 */
  {
    switch (cfg)
      {
      case 0 :     for (y=a ; y<=b; y++)
        for (x =c; x<=d; x++)
    sum=sum+(*(data_ptr+y*linesize+x))*get_weight(x,y,sigma,mean_val);
      break;
      case 1 :     for (y=a ; y<=0; y++)
        for (x =c; x<=d; x++)
    sum=sum+(*(data_ptr+y*linesize+x))*get_weight(x,y,sigma,mean_val);
      break;
      case 2 :     for (y=a ; y<=b; y++)
        for (x =y; x<=d; x++)
    sum=sum+(*(data_ptr+y*linesize+x))*get_weight(x,y,sigma,mean_val);
      break;
      case 3 :     for (y=a ; y<=b; y++)
        for (x =0; x<=d; x++)
    sum=sum+(*(data_ptr+y*linesize+x))*get_weight(x,y,sigma,mean_val);
      break;
      case 4 :     for (y=a ; y<=b; y++)
        for (x =-y; x<=d; x++)
    sum=sum+(*(data_ptr+y*linesize+x))*get_weight(x,y,sigma,mean_val);
      break;
      case 5 :     for (y=0 ; y<=b; y++)
        for (x =c; x<=d; x++)
    sum=sum+(*(data_ptr+y*linesize+x))*get_weight(x,y,sigma,mean_val);
      break;
      case 6 :     for (y=a ; y<=b; y++)
        for (x =c; x<=y; x++)
    sum=sum+(*(data_ptr+y*linesize+x))*get_weight(x,y,sigma,mean_val);
      break;
      case 7 :     for (y=a ; y<=b; y++)
        for (x =c; x<=0; x++)
    sum=sum+(*(data_ptr+y*linesize+x))*get_weight(x,y,sigma,mean_val);
      break;
      case 8 :     for (y=a ; y<=b; y++)
  sum=sum+(*(data_ptr+y*linesize+x))*get_weight(x,y,sigma,mean_val);
      break;
      }
  }

  if (cfg == 0)
    {
      fact = win_dim*win_dim;
    }
  if ((cfg < 10) && (cfg != 0))
    {
      for (i=win_dim; i>=0; i--)
        fact = fact + i;
    }
  if (cfg > 10)
    {
      temp2 = modf((float)cfg/2,&integer);
      if (temp2 == 0.5)
        fact = win_dim;
      else fact = 6;
    }

  temp = 0.7*(sum/fact);

  if (temp>255.0)
    temp = 255.0;
  else if (temp<0.0)
    temp =0.0;

  return (unsigned char)temp;
}

/*--------------function  get_weight--------------------------*/

float get_weight(int x,int y,unsigned char sigma,unsigned char mean_val)
{
  int m;
  float K1,K2;

  K1 = 2;
  K2 = 5;
  m = K1 * exp( -K2 * (sigma/mean_val) * sqrt(x*x + y*y) );

  return m;
}
```

IC/14PRINT

## I.3 Parameter Investigation and Results

A qualitative comparison of the various filter parameters was conducted. The parameters for the MHR filter are $T1, T2$ and $T3$. These are threshold values for determining whether an area contains a pike, an edge or is homogeneous. In order to reduce the effect of false edge detection, the following parameters were chosen: $T1 = 5, T2 = 15, T3 = 3$.

For the chosen parameters, the results presented in Table I.1 were obtained. It should be noted, however, that the filter does introduce false edge effects into the image. The results for the one look image indicate that the ENL

Table I.1: ENL and edge measure results for the MHR filter

| **image** | **edge measure** | **ENL** | |
|---|---|---|---|
| | | *area A* | *area B* |
| ESAR 1 look | 1.13 | 1.9 | 1.6 |
| ESAR 4 look | 0.23 | 12.0 | 16.5 |
| simulated 4 look | 0.27 | 9.9 | 20.2 |

was increased slightly, while the edge measure shows improvement over the unfiltered case (see Appendix A). A qualitative analysis of the test images shows that false edges have been created around certain existing structures and this has contributed to the increased edge measure value. Both of the 4 look images yield poor edge measure values and high ENL results. The ENL values differ between area A and area B. This indicates that the filter performance is dependent on local mean value. The results indicate that the filter performance is not satisfactory and support the need for a more robust edge detection algorithm.

# Appendix J

# Miscellaneous Code

This appendix contains the listing of *iofunct.h* as well as a sample input parameter file, *mean.dat*.

```c
/************************************
 * HEADER FILE DEFINING IO FUNCTIONS *

 file: iofunct.h

 ************************************/

#ifndef __IOFUNCT_H__
#define __IOFUNCT_H__

/* Define prototypes */
int skipchar(FILE *fp,char ch);
int GetString(FILE *fp,char *string);

/* NAME: skipchar()
   DESCRIPTION: Reads characters from a file until a certain chosen
                character ch is reached
   PARAMETERS:
        fp              local     R    Pointer to file to read
        ch              local     R    Character to read until    */

int skipchar(FILE *fp,char ch)

{

 char tch;       /* Store of character read in from the file */

 do
  {
   tch = getc(fp);     /* Reads character from file fp */
  }
 while ( (tch!= ch) && (tch != EOF) );  /* Reads while character is not */
                                         /* the end character or the EOF */
 return(0);
}


/* NAME: GetString()
   DESCRIPTION: Reads string from a file
   PARAMETERS:
          fp            local     R    Pointer to file to read
          string        local     R    Char pointer to start of string */

int GetString(FILE *fp,char *string)

{

 int n=-1;                       /* Counter of string postion */

 getc(fp);                       /* Get first space */
 do
  {
   string[++n] = getc(fp);       /* Reads character from file fp */
  }
 /* Reads while character is not a space, end of line, or end-of-file*/
 while ( (string[n] != ' ') && (string[n] != '\n') && (string[n] != EOF));
 string[n] = '\0';               /* Indicates end of string */
                                 /* the end character or the EOF */
 return(0);
}

#endif
```



```
/*************************************************
 *   file: mean.dat
 *   sample parameter entry file
 *************************************************/

LINESIZE  [no. of range pixels]          ==> 512
IMGLINES  [no. of azimuth pixels]        ==> 512
WINSIZE   [size of side of window (odd)]==> 3
ITER      [no. of iterations]            ==> 1
INNAME    [Input file name]              ==> g:\masters\images\test.raw
OUTNAME   [Output file name]             ==> g:\masters\images\meantest.raw
```

# Appendix K

# Images

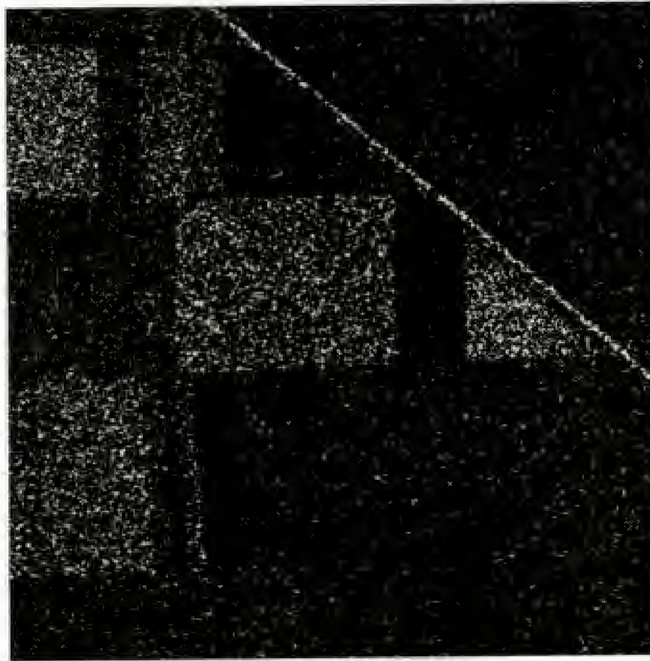

Figure K.1: Unfiltered 1 look ESAR image

Figure K.2: Mean filtered 1 look ESAR image

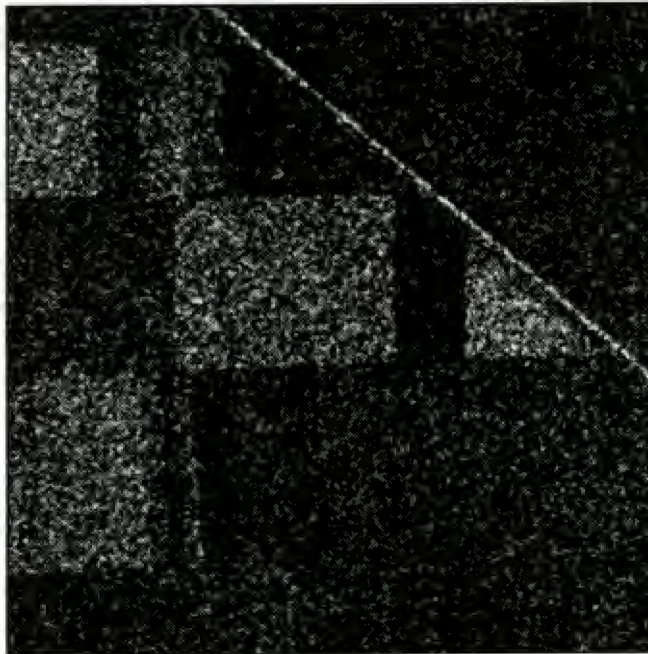

Figure K.3: Median filtered 1 look ESAR image

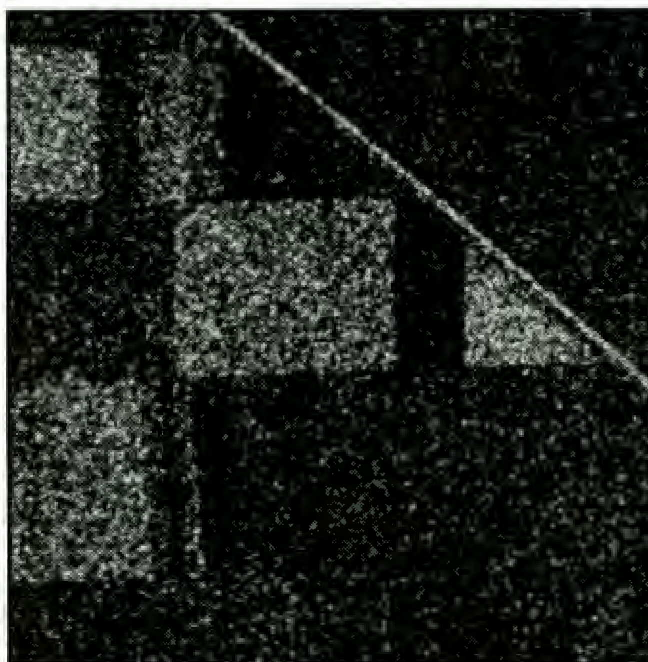

Figure K.4: Lorentzian filtered 1 look ESAR image

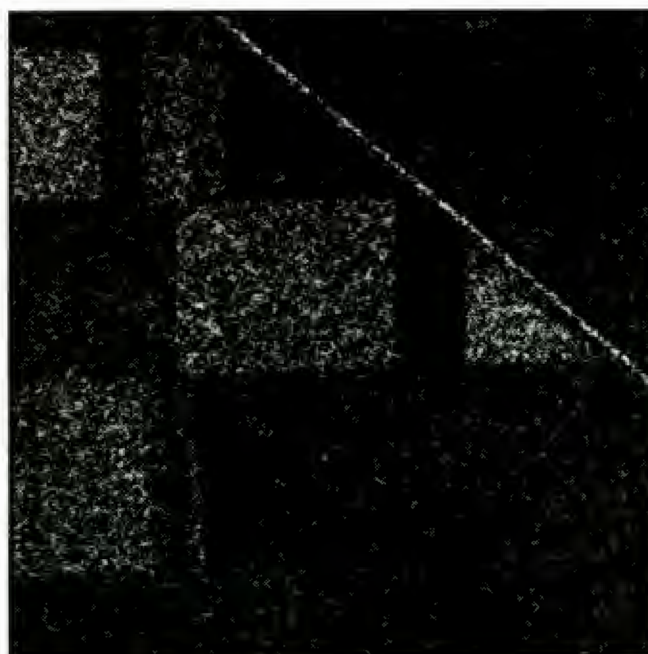

Figure K.5: KNN filtered 1 look ESAR image

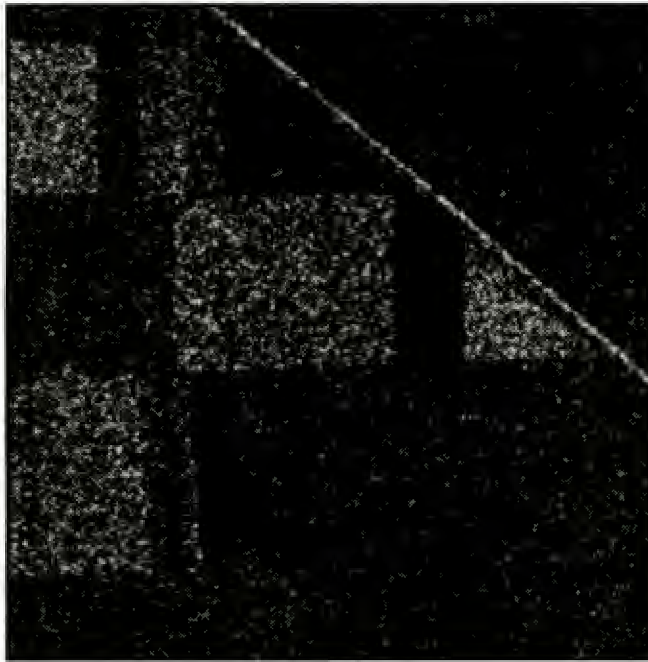

Figure K.6: Hirosawa filtered 1 look ESAR image

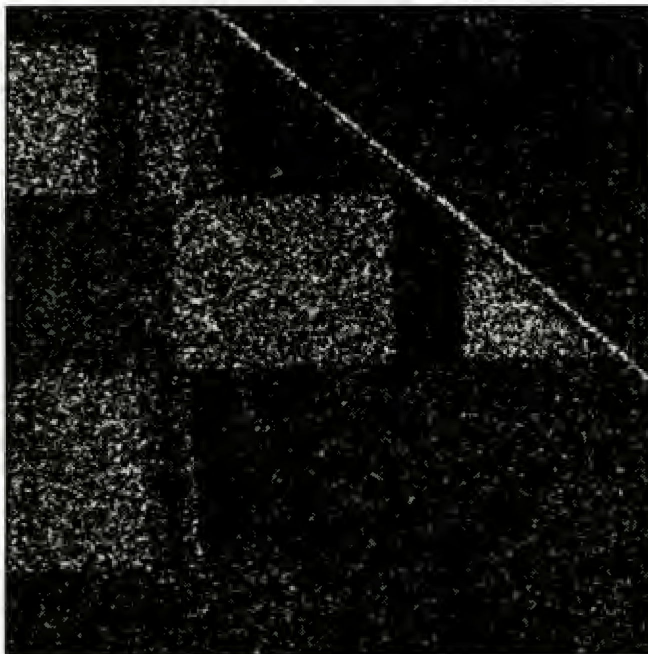

Figure K.7: MAP filtered 1 look ESAR image

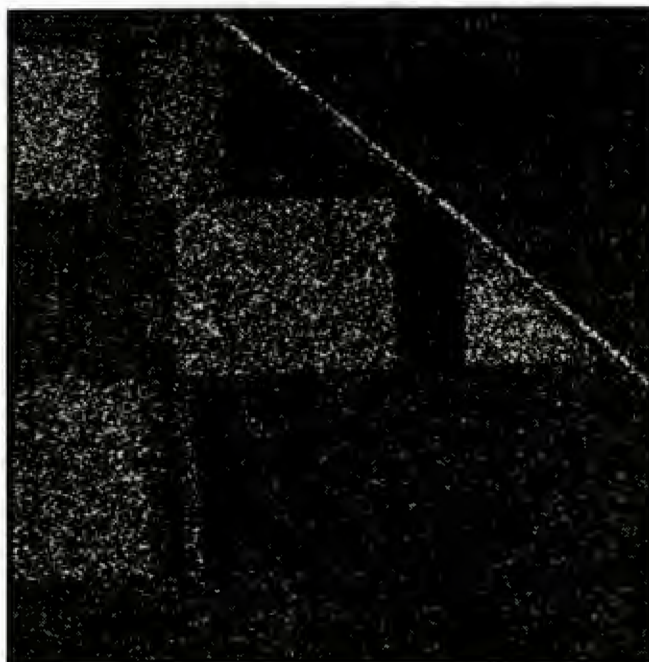

Figure K.8: Frost filtered 1 look ESAR image

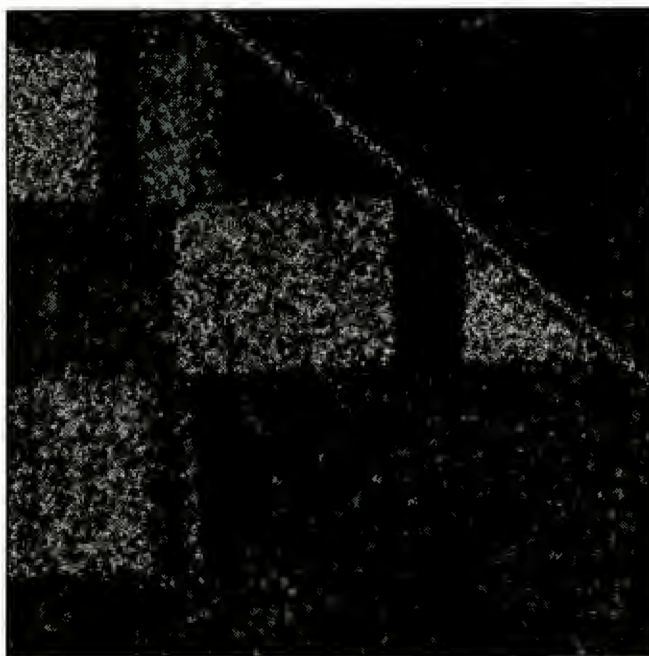

Figure K.9: MHR filtered 1 look ESAR image

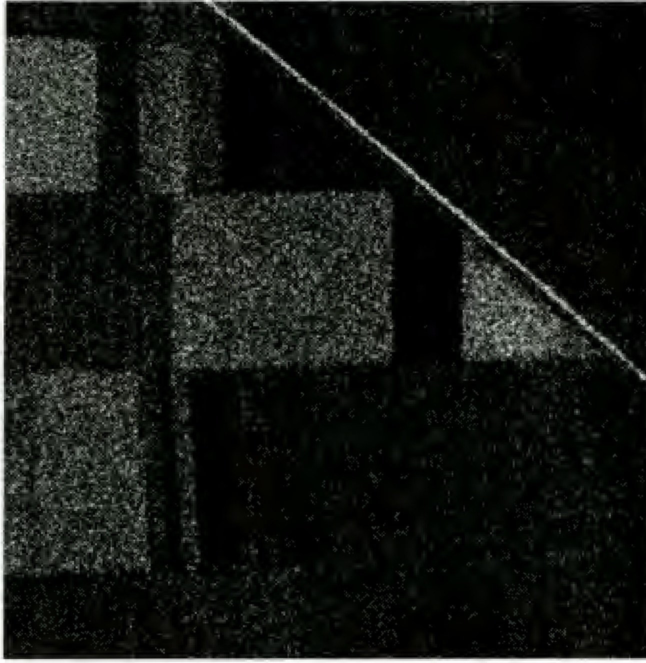

Figure K.10: Unfiltered 4 look ESAR image

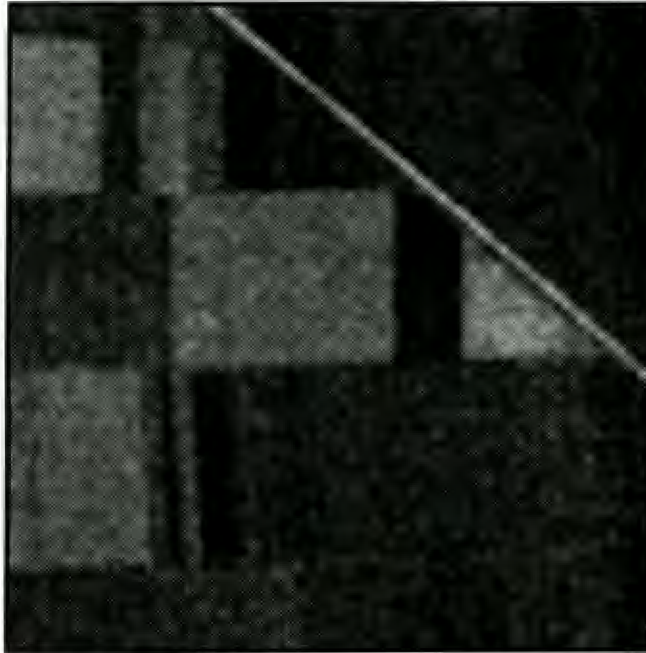

Figure K.11: Mean filtered 4 look ESAR image

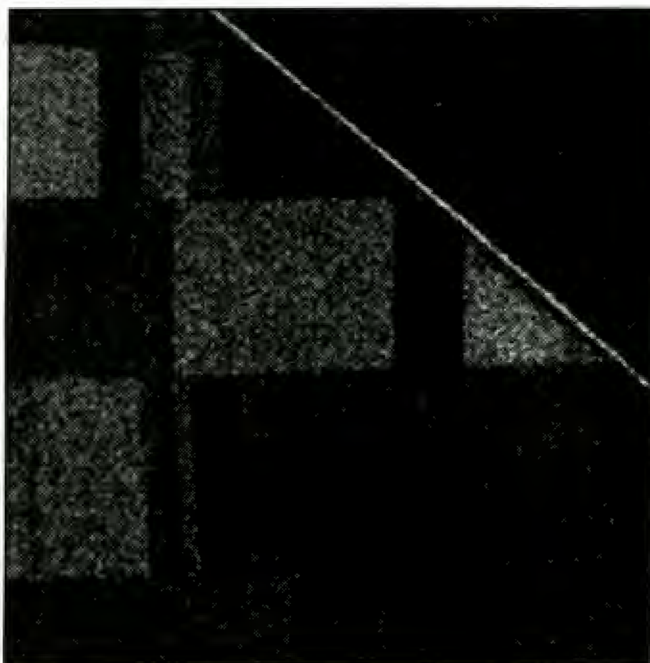

Figure K.12: Median filtered 4 look ESAR image

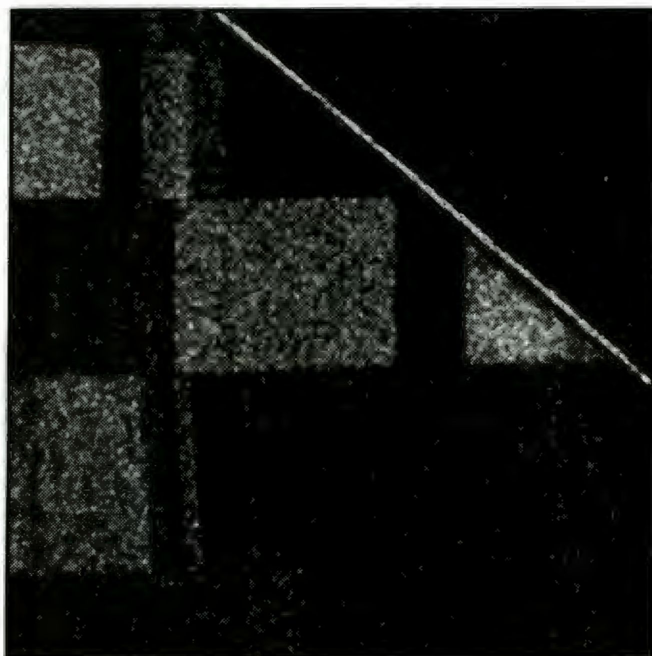

Figure K.13: Lorentz filtered 4 look ESAR image

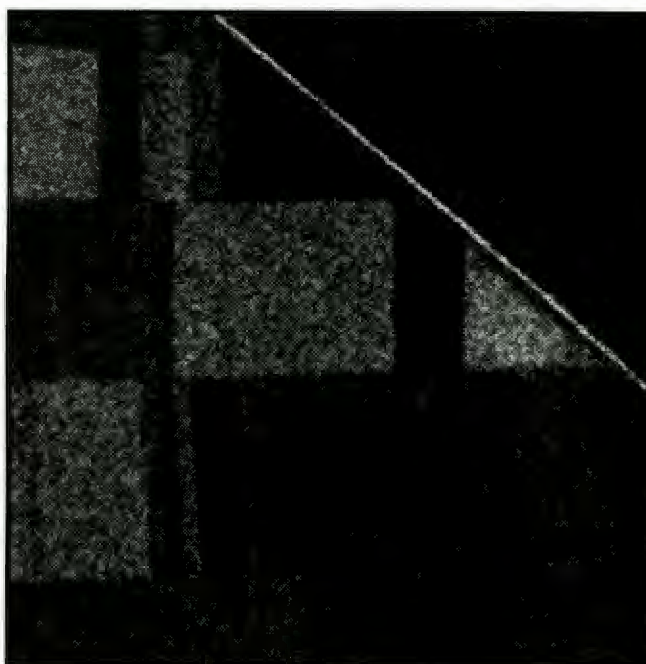

Figure K.14: KNN filtered 4 look ESAR image

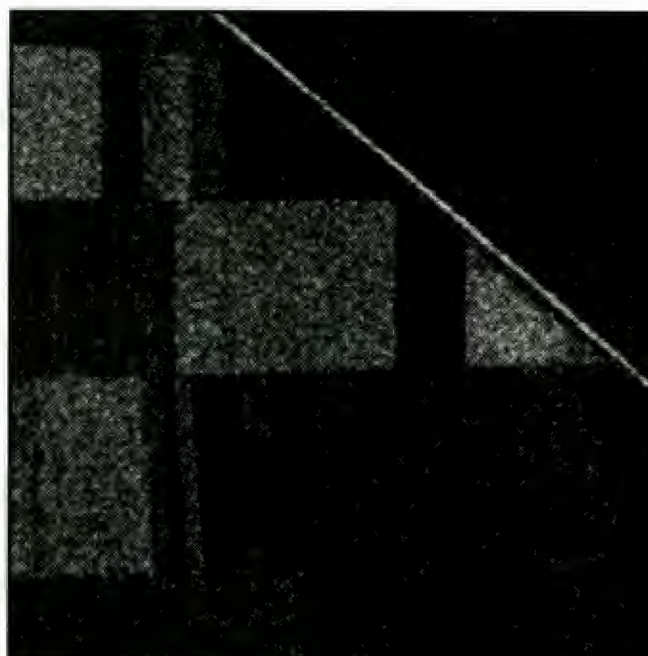

Figure K.15: Hirosawa filtered 4 look ESAR image

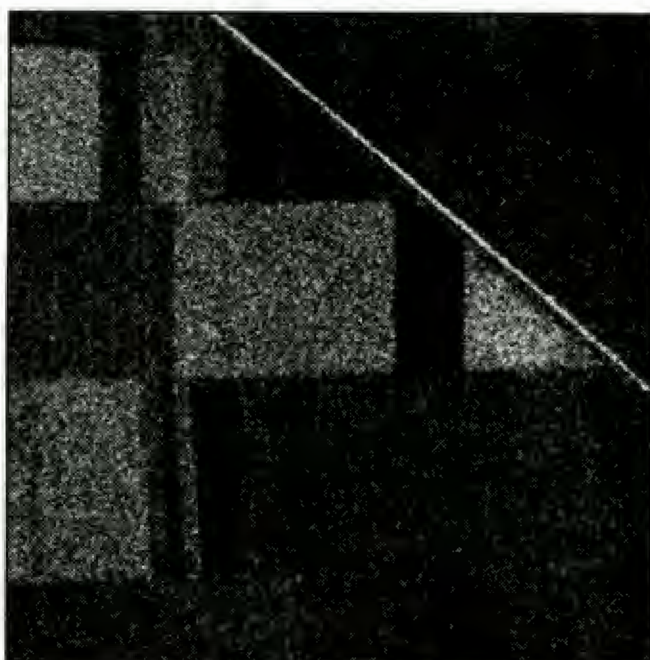

Figure K.16: MAP filtered 4 look ESAR image

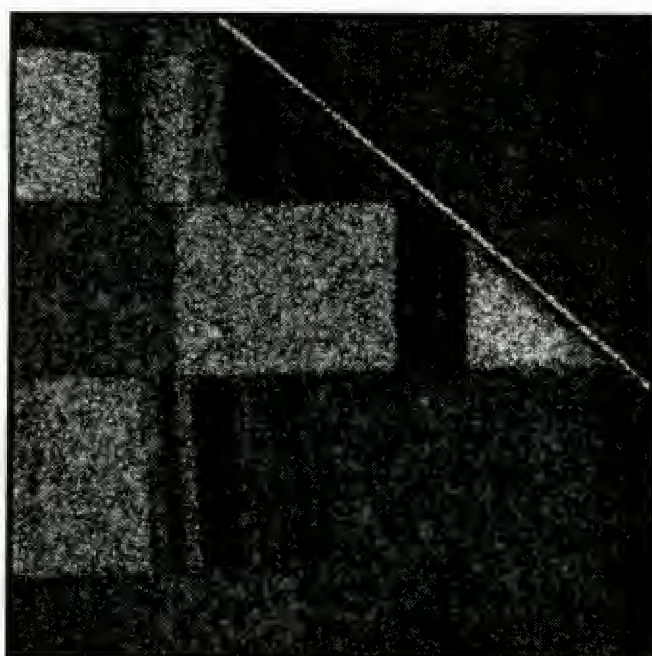

Figure K.17: Frost filtered 4 look ESAR image

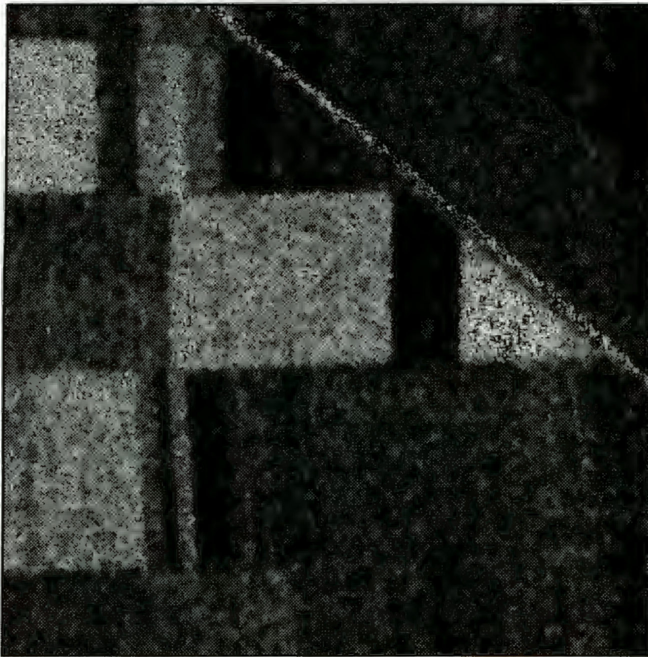

Figure K.18: MHR filtered 4 look ESAR image

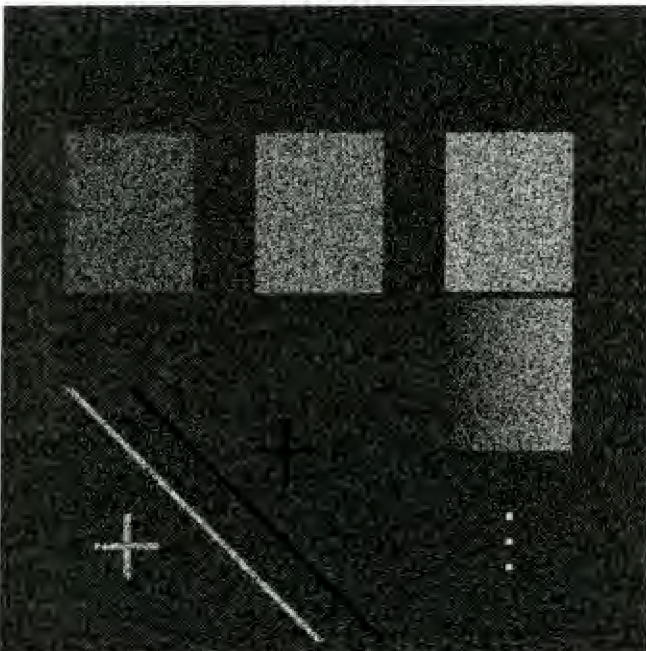

Figure K.19: Unfiltered simulated 4 look SAR image

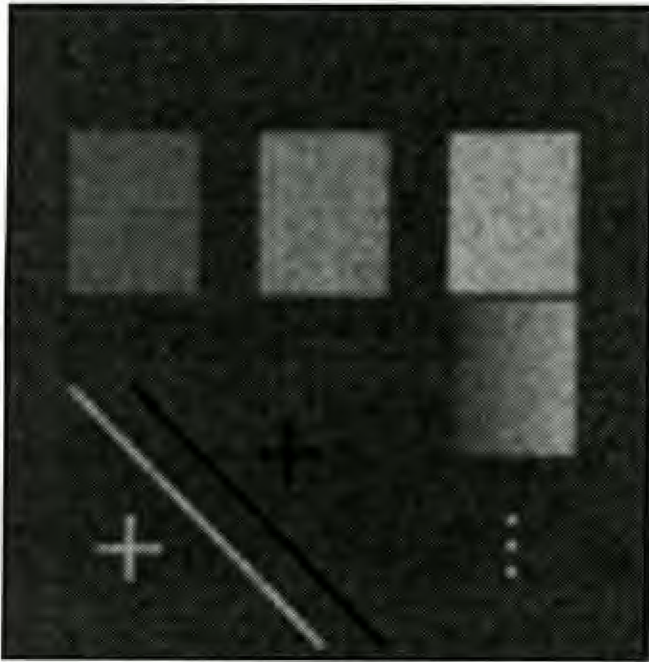

Figure K.20: Mean filtered simulated 4 look SAR image

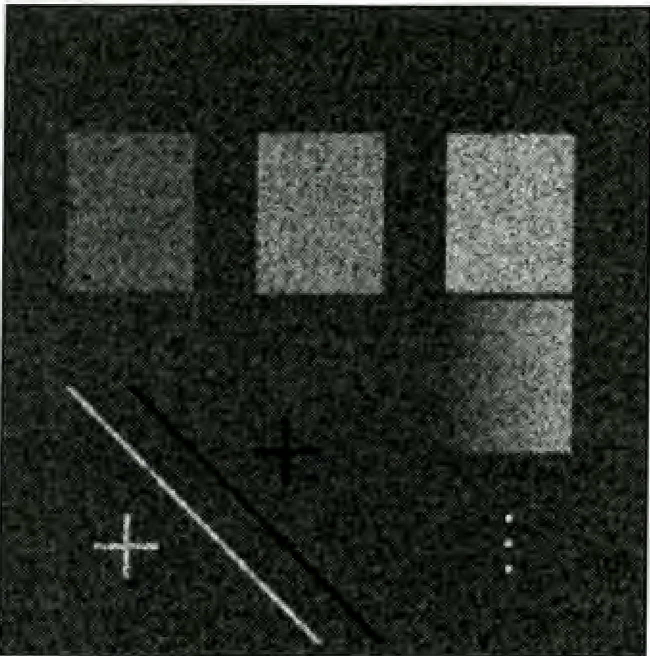

Figure K.21: Median filtered simulated 4 look SAR image

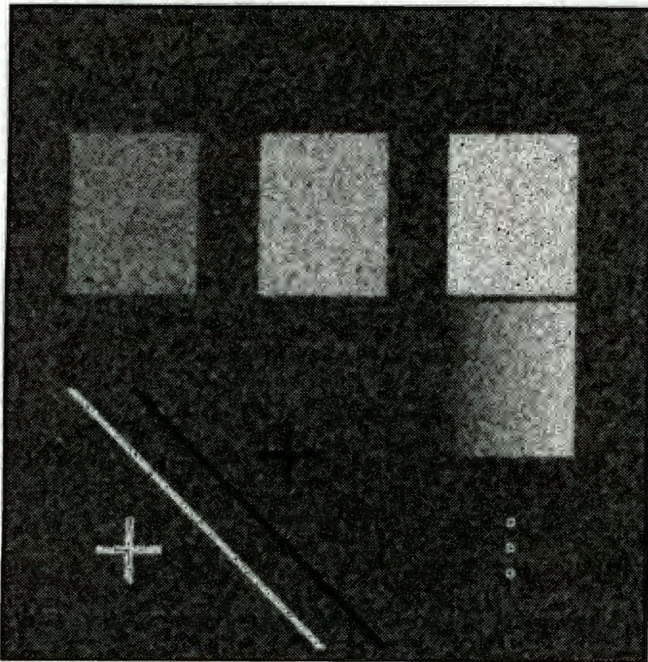

Figure K.22: Lorentzian filtered simulated 4 look SAR image

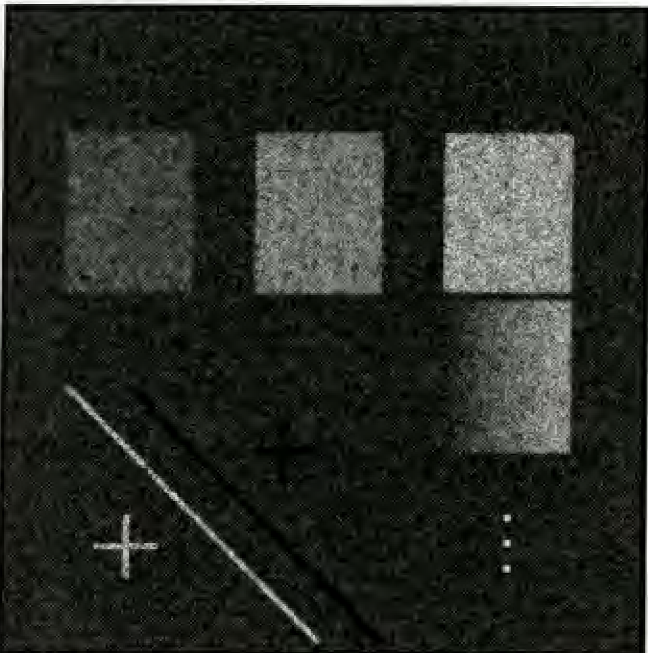

Figure K.23: KNN filtered simulated 4 look SAR image

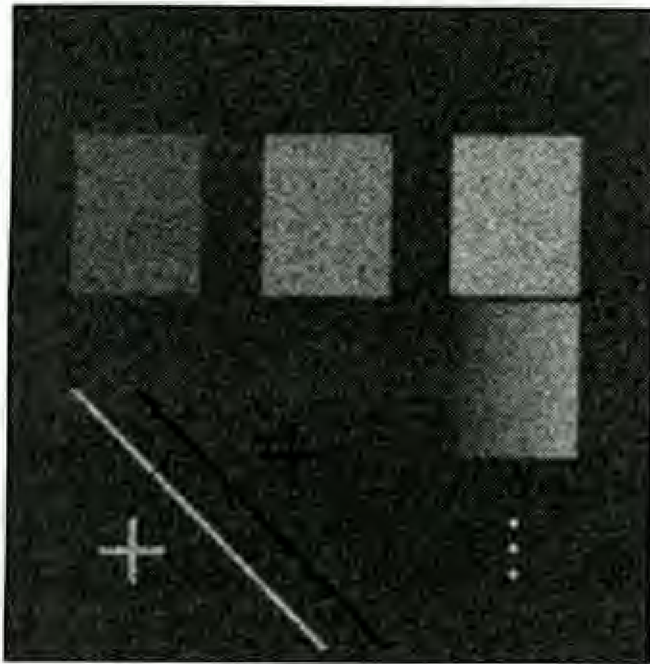

Figure K.24: Hirosawa filtered simulated 4 look SAR image

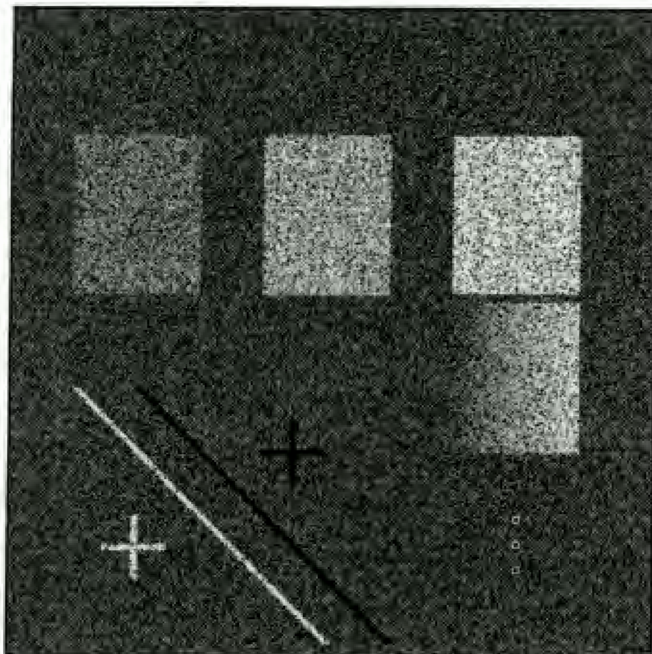

Figure K.25: MAP filtered simulated 4 look SAR image

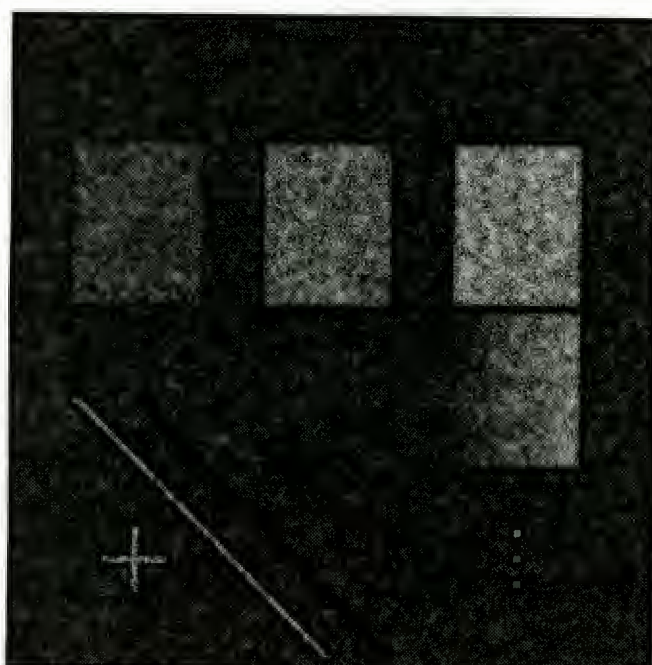

Figure K.26: Frost filtered simulated 4 look SAR image

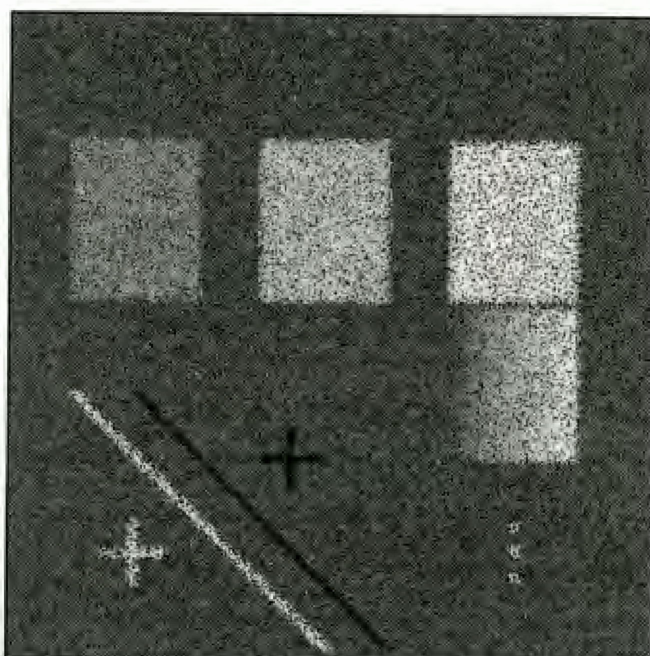

Figure K.27: MHR filtered simulated 4 look SAR image