A Spaceborne Synthetic Aperture Radar Data Processor

Prepared By: Simon Welsh, Masters Student in the Department of Electrical Engineering at the University of Cape Town.

Prepared For: Professor M.R. Inggs, Department of Electrical Engineering at the University of Cape Town.

30 September 1991

Thesis prepared in Partial Fulfillment of the requirements for the degree of Masters in Electrical Engineering.

Table of Contents

Table Of Figures

Glossary

ANSI	American National Standards Institute.
ASCII	American Standard Code for Information Interchange.
Azimuth	Azimuth, in this case, is in the direction of travel of the radar.
Azimuth Compression	The Matched Filtering of the azimuth samples with the azimuth reference function.
Azimuth line	A line of samples in the azimuth direction, at a fixed range.
Bit	Boolean piece of information either 0 or 1.
Byte	8 bits.
CFI	Standard Image File Format used at UCT, using 512 by 512 pixels with 8 bit magnitude.
Chirp Waveform	A Waveform that constantly increases or decreases in frequency for a fixed period.
DFT	Discrete Fourier Transform.
FFT	Fast Fourier Transform, which is a fast way of computing the DFT.
JPL	Jet Propulsion Laboratories.
Matched Filter	A filter that is matched to the expected input waveform, thus providing maximum signal to noise on the filter output.
PC	Personal Computer.
Pixel	An area on the ground represented by a single number during processing.
PRF	Pulse Repetition Frequency, the frequency with which the radar transmits pulses.
Pulse	Refers to a single transmitted radar pulse which may consist of a frequency swept waveform.
Radar Footprint	The area on the ground intercepted by the radar beam, while in a fixed position.
Range	Range is measured in the direction in which the radar is pointing.
Range Bin	A range bin is a single cell at a fixed distance from the radar.

Range Compression	The Matched Filtering of the returned range samples with the range reference function.
Range line	A line of sample points in the range direction, at a fixed azimuth position.
SAR	Synthetic Aperture Radar.
SIR-B	Shuttle Imaging Radar - B.
Speckle	The random variation of intensity between pixels in close proximity, caused by surface roughness.
Swath	The complete area on the ground covered by the radar beam while it moves along.
Swath Velocity	The forward ground velocity of the radar.
UCT	University of Cape Town.
Word	Normally 4 Bytes, as in this case.

Acknowledgments

I would like to thank my thesis Supervisor, Professor Mike Inggs, firstly for proposing research into this exciting field of engineering, and secondly for his continued assistance throughout the period of development. I would also like to thank my colleagues Paul Kritzinger, for his theoretical input to the problem and Alastair Knight for his assistance with the software. Thanks also to Amy Pang of JPL for supplying the SIR-B data, and all others who assisted with the production of this thesis.

Synopsis

This thesis is concerned with the design and implementation of a Synthetic Aperture Radar (SAR) data processor. The implementation of the processing is based on a standard sequential approach to the problem and employs commonly used algorithms. The processing was done using the C language running on an IBM Compatible Personal Computer.

The raw data processed was that obtained from the Shuttle Imaging Radar B (SIR-B), and was supplied by the Jet Propulsion Laboratories (JPL) in California. The basic functions performed by the software include range and azimuth processing, which involve the match filtering of reference functions with the raw data. Compensation for the effects of being a spaceborne SAR were also implemented, which involved compensation for the effect of planet rotation and radar height.

Images processed by JPL of the same area were also available, which allowed for direct comparisons between the outputs of the two SAR processors. The images produced were passed through a number of filters, to improve the image quality, and resulted in favorable comparisons to the JPL generated images. The actual images are included in the later sections of the thesis.

1. Introduction

The processing of Synthetic Aperture Radar (SAR) data is being done in a number of locations around the world, outside of South Africa. Various algorithms are being used, and the raw radar data is being produced by a number of radar sources. Processing of raw SAR data has not before been performed in South Africa, and the purpose of this thesis is to present a fully functional and locally designed SAR processor.

At the University of Cape Town (UCT), we were lucky enough to be provided with Raw and Processed SAR data from the Shuttle Imaging Radar B (SIR-B). This was supplied by the Jet Propulsion Laboratory (JPL) in California, who currently process this on their SAR processor. The development of the SAR processor presented here was thus based around the processing of this data.

This thesis is mainly concerned with the actual processing of the raw radar data. The masters thesis submitted by Paul Kritzinger, also at the University of Cape Town, deals with the theoretical aspects of the processing. This thesis is referenced a number of times and should be referred to for the actual formulae implemented.

In the first section of this thesis, a brief explanation of the principles behind SAR will be given, as well as a brief history of SAR processing.

The main sections will be concerned with the design and implementation of the software. The basic design requirements of the software may be summarised as follows:

- The Software will provide for the processing of raw Synthetic Aperture Radar data, taken from a moving radar platform, and will provide for the compensation of spaceborne SAR effects.

- The output from the program will consist of an image file suitable for viewing with a simple viewing program. A Viewer will be provided for a standard IBM VGA monitor, and a printer driver for the Hewlett Packard Laserjet.

- The program will be written in a portable language, such that it may be run on a variety of systems.

- The program shall be reasonably efficient in terms of speed, memory and disk space usage. However, as an initial SAR processor implementation, correct operation was considered more important than efficiency.

2. Background to Spaceborne SAR Processing

A brief background to Spaceborne SAR Processing will be given first. This will include some of the applications, the basic principles of operation and a mention of some history to SAR and the processing thereof.

2.1 Applications of Spaceborne SAR

The main application of spaceborne SAR is in the production of high resolution radar images of the earth's surface. This provides a different image to that provided by optical means, and is formed by variations in the radar reflectivity of the surface being mapped.

Due to this fact, the number of applications increases considerably. For a start the radar pulses can penetrate through cloud cover, and are not dependant on any illumination such as the sun. Different features show up more strongly, particularly those with high radar reflectivity such as man made metallic objects.

The radar pulses can also penetrate through low loss materials such as low moisture content soil, as found in the arid regions of Africa. This allows the underlying rock structures to be observed.

Oceanographic applications include the detection of different sea states, as rough sea conditions would lead to an increase in radar reflectivity. This could indicate the presence of high wind conditions on the ocean surface, and allow for the location of storms.

These and many other radar imaging applications are discussed by Elachi[1].

2.2 History of Spaceborne SAR Processing

The SEASAT global ocean-monitoring satellite launched by NASA in June 1978 was the first spaceborne SAR that proved imaging of the earth's surface to be practically possible. This was followed by the Shuttle Imaging Radar A (SIR-A) which was the first of the Shuttle borne radars.

In the early days of SAR processing, optical processors were used. These could basically perform a two-dimensional Fourier Transform instantaneously, which is what SAR processing essentially involves. The main processors for the Seasat SAR and the SIR-A were both optical. The actual optical implementation is discussed by Elachi[2].

Optical processors have a number of shortcomings, including calibration and the delay in film developing and handling. With the advent of fast digital processors, a number of SAR processing algorithms have been devised. The digital approach provides for far more flexibility and access to intermediate processed data, as well as the possibility of real time SAR processing.

The processing of the SIR-B radar data is currently done by the Jet Propulsion Laboratories (JPL) in California, using a digital processor.

2.3 Basic Principles of Operation

The basic principles of operation of SAR will be discussed here, but a more in depth analysis is given by Kritzinger. The main purpose of SAR is to obtain a high resolution in both the range and azimuth directions.

The Complete SAR scenario is shown in the diagram below, which also allows for the definition of some terms:

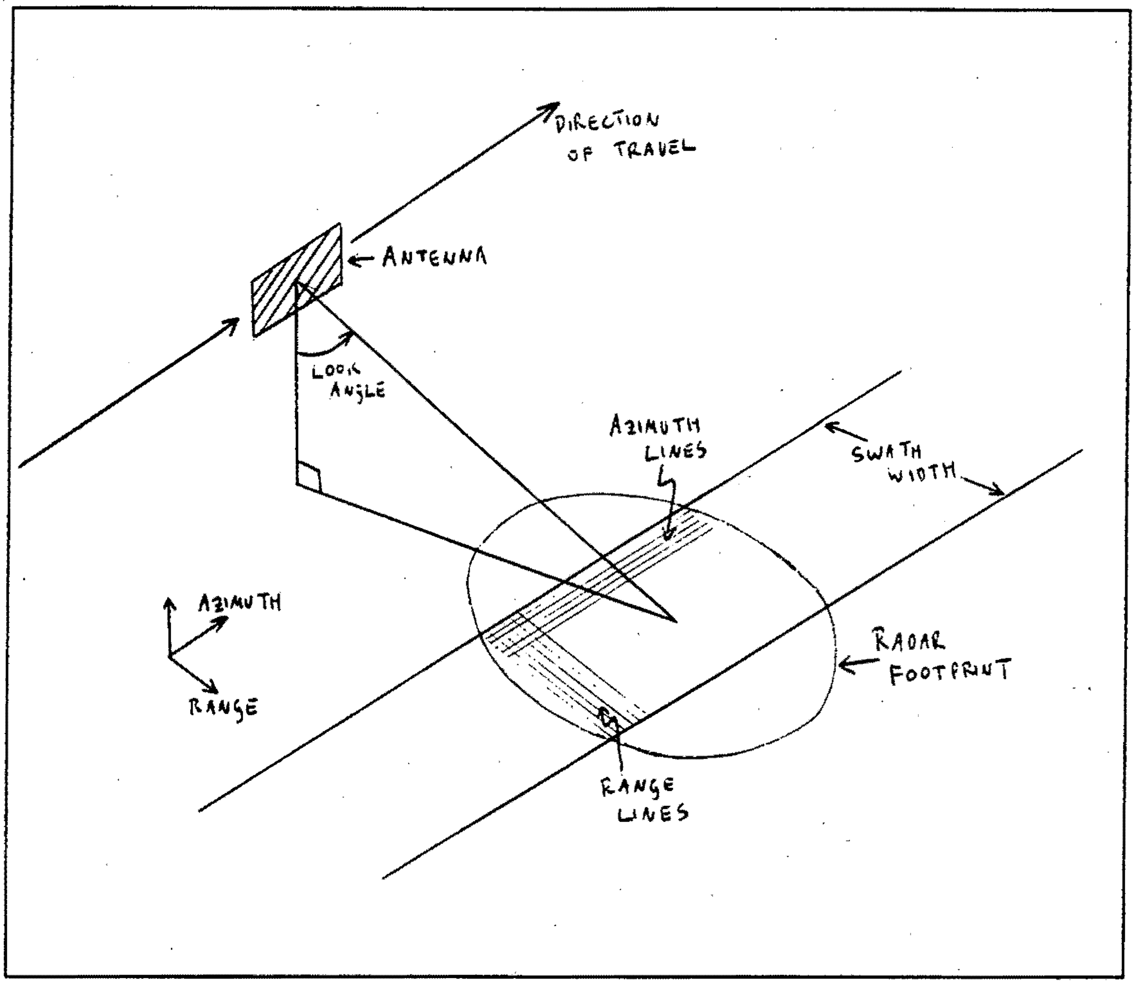


Figure 1 SAR geometry

From the given diagram, it can be seen that the radar platform moves over the earth's surface while projecting a wide radar beam. This beam illuminates an area on the earth's surface called the radar footprint, and is projected at an angle measured from the vertical, called the look angle.

The radar footprint moves over the earth's surface as the radar moves in the azimuth direction, illuminating a continuous area called the radar swath.

During the processing, range and azimuth lines are formed from the radar echoes. These are also defined on the diagram for clarity later on.

2.3.1 Range Resolution

To achieve a high range resolution while still maintaining a good Signal To Noise ratio, conventional radar systems as well as SAR's may spread out the transmit pulse while using some form of coding, such as a frequency swept pulse. The return signal is then match filtered with a replica of the transmitted pulse, thus effectively allowing a very narrow pulse to be formed.

This match filtering may be performed either in the hardware of the radar system itself, or it may be performed later in software after receiving and sampling the returned signal.

In the case of the SIR-B system, a down swept frequency pulse, or chirp, is used, which is not match filtered by the radar. The software processor must thus match filter with the transmitted chirp waveform. This matched filtering of the range returns will be termed **Range Compression.**

The operation of Matched filtering is a standard signal processing technique and is discussed by Stremler[3].

2.3.2 Azimuth Resolution

Azimuth resolution is achieved using the principle of the Synthetic Aperture Radar. This allows a high azimuth resolution to be achieved by synthesising a very long antenna. Each of the radar returns, at given ranges, are phase shifted and summed along the length of the synthetic antenna allowing a narrow beamwidth to be synthesised.

The real beamwidth of the radar is designed to be wide enough to illuminate a single point target for a number of pulses while the radar moves past. This allows the Synthetic array to be the distance covered by the radar while illuminating the single target.

Complete descriptions of how Azimuth Resolution may be achieved using the SAR process have been documented by a number of authors, including Kritzinger[4] and Elachi[5]. The results of this theory are relevant to the processing, but the theory itself is not, and so will not be described here in detail.

2.3.3 Spaceborne Considerations

A number of SAR imaging radars in use are airborne, as opposed to spaceborne. An aircraft borne SAR would not require the consideration of the following two factors, which are specific to spaceborne SAR imaging.

i) Range Walk

Range walk may be defined as the movement of the target in the range direction, while being illuminated by the radar beam. The fact that the radar platform is not within the earth's atmosphere, and is not rotating with it, means that the target will always have a lateral movement component unless the platform is moving with the rotation. This could only occur if the platform remained over the equator. The range walk is thus dependant on the direction of travel and latitude of the radar platform.

The length of time for which the target is illuminated, and the geographical position of the radar will determine the magnitude of the range walk. This must be compensated for during the processing.

ii) Range Curvature

Range Curvature is the variation in distance to a target on the earth's surface, while being illuminated by the radar beam. This variation is in addition to that caused by the range walk. The range curvature effect is caused by the geometry of the situation, and the fact the radar platform is so high off the ground. It is due to the spherical nature of the earth's surface.

3. Processing Algorithms

A discussion on the various algorithms required to process the raw data will be presented in this section.

3.1 Range Processing

The range compression operation requires the match filtering of the returned radar echoes with the radar transmit waveform. The frequency swept transmit waveform of the SIR-B may be represented by a number of parameters which allow its re-creation.

The actual match filtering operation may then be performed using the Discrete Fourier Transform (DFT). This is the most efficient way of convolving the reference transmit waveform with the returned samples, which is what matched filtering amounts to.

The most efficient way of implementing the DFT is by using the Fast Fourier Transform (FFT). The matched filtering operation using FFT's may be summarised as follows:

1) FFT's of the range reference function and input samples are taken.
2) The FFT of the range reference function is conjugated, and then multiplied by the FFT of the input samples.
3) The inverse FFT of the result is taken, which yields the matched filter output.

In the case of the SIR-B, the received radar waveform is sampled at an intermediate frequency. This requires demodulation of the signal down from the intermediate frequency to baseband. A digital technique for achieving this was developed by Kritzinger[6], and may be summarised as follows:

1) Perform a forward FFT on the sampled range waveform.
2) Set the Negative frequency spectrum to zero.
3) Demodulate by Shifting the positive spectrum to baseband.
4) Perform an inverse FFT.

This process of demodulating may be combined in the range compression matched filtering operation, which includes the forward and inverse FFT's.

3.2 Azimuth Processing

Azimuth processing requires the summing of the returns in the azimuth direction, with various phase shifts applied prior to each summation. Various approaches to this may be taken as discussed below.

3.2.1 Unfocussed SAR Approximation

The simplest approach to azimuth processing is not to apply any phase shifts to the data prior to summation. This results in the basic unfocussed solution.

The number of azimuth pulses that can be summed in this way before degradation in the image occurs, is limited, and is discussed by Kritzinger[7]. This is due to the variation in distance to the target, due to the trajectory of the radar platform.

3.2.2 Focussed SAR

The Focussed Azimuth Compression is essentially the application of the Synthetic Aperture Principle to the data. Returned pulses are summed in the azimuth direction after suitable phase shifting. The phase shift required by each returned sample may be represented by the azimuth compression reference function. Matched filtering with this reference function then effectively allows the pulses to be summed with the required phase shifts.

As in the range compression operation this may be most efficiently achieved with the Fast Fourier Transform.

As the distances and geometry vary over the width of the swath, so the azimuth reference function varies. This means that for each azimuth line, a different azimuth reference function should be generated. It was found, however, with a suitably shortened azimuth reference function, that the same function could be used over the entire swath. This simply required that the function used was generated based on the distances to the centre of the swath. This variation in the azimuth reference function is called depth of focus and is discussed further by Kritzinger[8].

3.3 Processing Approach

The maximum amount of memory required by the program will be dependant on the way in which the processing is performed. Two possible approaches may be taken, each of which will be discussed below.

3.3.1 Non-Separable Processing

The range walk and curvature effects, as mentioned earlier under spaceborne considerations, should be compensated for during the processing. These effects would normally require that range and azimuth processing be done together. The movement of the radar returns between azimuth lines (Range bins) while illuminated by the radar beam necessitates the consideration of more than one azimuth line at a time when performing the azimuth compression.

This would imply that a simple one-dimensional approach would fail. With suitable compensation however, as will be discussed under separable processing, the one-dimensional separable approach may still be used.

The non-separable two dimensional approach would be the ideal way of processing the data, which would combine the range and azimuth reference functions into a single two dimensional reference function. This would allow the range curvature and walk compensation to be built into this two dimensional reference function. A two dimensional matched filtering operation would then be performed.

The problem with the non-separable approach, however, is the large amount of memory that is required. A complete two dimensional array of data would need to be processed at a time. This could be done by spooling from the mass storage device, but would probably slow down the processing considerably.

Non-separable approaches to SAR processing are discussed by Franceschetti[9] and Di Cenzo[10].

3.3.2 Separable Processing

The processing may be separated if the two operations can be performed one dimensionally. This may be done if the range walk is compensated for and if the range curvature is ignored. The range walk is compensated for by slewing the data returned from each pulse transmit. This effectively lines up the azimuth lines allowing one-dimensional processing on each individual azimuth output line.

The range curvature, for the case of the SIR-B geometry, was found by Kritzinger[11], to be negligible for a suitably shortened synthetic array. It could thus be ignored in the processing without a severe degrading of the image quality.

Separable processing was thus decided upon, as this requires far less memory than the non-separable approach. Using separable processing, the maximum amount of memory required by the program will be that required to execute a complete match filter operation on a single range or azimuth line.

3.4 Compensation for Separable Processing

Range walk compensation is all that is required to perform separable processing. Range walk is caused by the movement of the earth beneath the radar platform in the range direction. As the radar platform moves forward, so the planet rotates beneath it, causing the data returns to be slewed.

The actual slewing effect must be compensated for in the time domain. The frequency domain effects are compensated for in the azimuth compression reference function.

Time domain compensation can take place at any stage in the processing, and it was found to be optimal to place it after the range compression operation and before the outputting of the data to the mass storage device. Thus when writing out each range line, it is slewed back to compensate for the range walk.

The azimuth reference function is also affected by the range walk in a significant way, but this is handled separately to the slewing.

3.5 Pixel Spacing and Ground Truth

The output image files will be made up with a number of pixels, or points, each representing a given area on the ground.

The pixel spacing is independent of the finest achievable resolution. The resolution is the minimum distance between which separate targets may be discerned, which may be over a number of pixels. The limits on resolution will not be discussed here, as they have no bearing on the actual processing. The pixel spacing, however, will be discussed briefly, as this is directly linked to the size and dimension of the data files. It also allows actual ground distances to be matched to the output image files.

The separate pixel dimensions in range and azimuth will be discussed first, followed by the effect of the radar look angle and how this may be compensated for.

3.5.1 Range Pixel Spacing

The pixel spacing in the range direction is determined by the sampling frequency of the radar. When a single pulse is transmitted by the radar, a period of time is allowed to elapse while the radar returns are sampled. As the samples are taken, returns from further and further away are recorded, since these take longer to return to the radar.

The time between taking samples is equivalent to the difference in ranges reached by the transmitted pulse. Thus the sample taken represents a single pixel, which is spaced from the adjacent pixel by an amount equal to the additional range covered by the pulse since the sampling of the previous return.

3.5.2 Azimuth Pixel Spacing

The returns from each pulse transmitted by the radar are sampled to determine the range pixel spacing. The transmission of pulses at different azimuth positions determines the azimuth pixel spacing. Each transmitted pulse is used to form a single line of data in the range direction, and the time between these pulse transmits determines the spacing of these lines.

Thus, the spacing of the pixels in the azimuth direction is determined by the ground distance covered between radar pulse transmits. This is dependant on the forward ground velocity of the radar, and the Pulse Repetition Frequency (PRF) of the radar. The ground velocity of the radar is dependant on the earth's rotation and the trajectory of the radar platform, and the PRF is a known radar design parameter.

3.5.3 Look Angle Distortion

The calculation of the range pixel spacing assumes that the returned radar echoes are from objects on a two dimensional plane that passes through the trajectory of the radar. This is, however, not the case as the radar returns are reflected from the surface that is both below and to the side of the radar. This means that there is a distortion in the spacing of the range pixels, which is determined by the angle at which the radar is pointing relative to the reasonably flat surface below.

Thus to obtain the true range pixel spacing, one must take into account the size of the radar look angle, which is the angle that the centre of the radar beam makes with the vertical.

4. Software Design Considerations

The software design was made up of a number of steps, each of which will be discussed in the following sections.

4.1 Choice of Hardware and Operating System

The main operating systems and hardware considered in the design process are discussed below.

4.1.1 VAX mainframe

i) Advantages

This is a multi-user mainframe based system, which provides a number of advantages and disadvantages. The main feature of this system is the speed with which the VAX mainframe can operate, particularly in conjunction with an array processor. The amount of continuous memory that may be addressed is also large compared to that in a personal computer, which allows for the convenient allocation of large blocks of memory.

ii) Disadvantages

The main disadvantage is the fact that it must be used in a multi-user terminal environment where processing time is shared with other users. This environment slows development time, but in a dedicated system, with no other users, the speed could be greatly increased.

4.1.2 IBM Compatible Personal Computer

i) Advantages

The main advantage of the personal computer (PC) approach is the wide availability of software for these systems, which reduces the need to re-write already available routines.

A number of PC compatible motherboards are also available which range from the 8088 to the 80486 processor based motherboards. These provide a large range of operating speeds without sacrificing any hardware compatibility.

Advanced co-processor boards are also available which can interface directly with the main PC motherboard. These can allow vastly improved performance, while still using the basic input and output provided by the PC. Examples of such co-processors are the Intel i860 and the Inmos T800.

ii) Disadvantages

The main problem with the personal computer approach is the fact that directly addressable memory is limited, and complex memory management is required. This was not considered a major limitation, as the algorithms used were designed to be memory efficient and would not suffer from a lack thereof. The compiler used would also automatically provide the necessary memory management where applicable.

The decision was thus to develop the software in the PC compatible environment using an 80386/80387 processor combination. This was also chosen due to its availability to the author.

4.2 Choice of Language

The C language was chosen due to the ease with which the code could be transported to run under different operating systems or on different hardware platforms. All C language implementations conform to a standard set of ANSI-C language commands, allowing any ANSI-C implementation to run any given piece of ANSI-C code without any modifications of the code required.

The actual version of C chosen was Borland's Turbo C V2.00. This was used due to its availability and the fact that it runs under the operating system chosen.

A number of additional commands are provided by this version of C, but as far as possible only the standard ANSI-C commands were used so as to maintain full compatibility. Non standard commands are clearly marked in the code and are only used to allow the code to be run on the specific hardware platform chosen.

4.3 Mass Storage and the Flow of Data

Various data storage structures are used throughout the program, depending on factors such as accuracy and purpose. The first data structure is that used by the raw data file as received and sampled by the radar system.

4.3.1 Raw Data Structure

The raw unprocessed SIR-B data was held in a single file consisting of a number of records, each with a header. The file layout is given in Appendix E. Each record contains 3415 samples of radar return data taken at successive azimuth positions. Each sample is represented with 6 bit precision, and the data file is stored in a compressed format of 5 samples per 4 byte word. The raw data extraction module must therefor include a data decompaction routine.

The total size of the raw data file is 66.06 Mbytes, which includes 21504 records each of 3415 samples. The size of this file is an indication of how essential the efficiency of the data handling must be.

4.3.2 File Access

File access must clearly be done sequentially to allow for maximum throughput. The initial processing requires that range compression be performed. This involves a single range line of data at a time, which is held in a single record in the raw data file. The initial processing can thus be performed directly and sequentially on the raw data file, since records are stored sequentially. The output from this first stage will of course also be in order of range line, as read in.

The azimuth compression stage requires the data one azimuth line at a time. The azimuth lines run across the range lines, and to read in a complete azimuth line would require reading in the entire file of range lines. This would be extremely inefficient as the complete file would need to be read for each azimuth line.

If memory permitted, then the complete file could be stored in memory, and then accessed in the desired way, instead of having to re-read it for each azimuth line. This would require 66 Mbytes of memory, which were assumed not to be easily available.

To allow for the efficient processing of azimuth lines, the complete file may be corner turned on the mass storage device. This would allow azimuth lines to be read in sequentially from the file and processed. This technique is commonly used and is mentioned by Barber[12] among others.

Depending on the algorithm used, the corner turn operation itself may require a certain amount of free memory, but this will be a fraction of the size of the complete data file.

Due to the size of files involved, segments from the raw data file may be separately extracted and processed. Thus a smaller segment may be selected for processing if disk storage space is limited.

4.3.3 Specific Intermediate Data Files

As can be seen from the preceding discussion, a number of intermediate files will need to be produced during the processing. The specific intermediate files produced by each of the main modules will be discussed here.

i) Range Compression file output

The first file produced will be by the range compression module. This file will hold a two dimensional array of complex numbers of size determined by the number of azimuth and range lines processed.

Each complex number will be represented with a dynamic range of 16 bits fixed point. 8 bits will be used for the real component and 8 bits for the imaginary component. This compact format is required to reduce the amount of disk storage space required. The original raw data was represented with 6 bits, so no loss of information is involved.

ii) Corner Turn file Output

The corner turn module will take as input the complex data file from the range compression module, and will output this file after corner turning. The range compression output disk file data structure is thus maintained, but with the azimuth and range dimensions swapped.

iii) Azimuth Compression file Output

The input to this module will come directly from the corner turn module. The output file produced by the azimuth compression module will be an image file representing the final output from the processing stages.

This output will be of real type only, with each pixel represented with 8 bits. This format is all that is required to produce complete 256 gray level output images.

4.3.4 Standard Data Structure for Image Representation

A standard file format for storing gray level images has been defined at the University of Cape Town. The file format is defined for images of 512 by 512 pixels, with each pixel represented with an 8 bit gray level magnitude. The file is structured as a byte dump of all 512 by 512 pixels, taken from left to right and from top to bottom of the image. All image files conforming to this standard will have the extension 'CFI'.

A module to produce standard CFI images from the main azimuth compression output image file will be provided. A number of existing programs compatible with this file format were used, and will be discussed later.

4.3.5 Scaling of Output File Data

The range compression and azimuth compression modules both output data in a fixed point format. The scaling of this output data is thus vital to ensure that it is centred in the middle of the fixed point range with a minimum of overflow and underflow errors. Output Scaling for each of these modules is performed as described below.

i) Range Compression Output Scaling

Before writing out any data to the disk, a suitable output scaling constant is determined. This is done by performing a number of sample range compressions and then averaging the peak output values from each of these sample range lines. In this way an average of the peak output values will be found.

Using peak output values to scale will ensure that the peaks are not clipped off during the output process. The averaging is done to avoid scaling off an abnormally large peak.

ii) Azimuth Compression Output Scaling

This is done in a similar way to the range compression output scaling, except that a number of sample azimuth compressions are performed instead. Again the average of peaks is used to determine a scaling constant.

4.4 FFT Considerations

The Fast Fourier Transform as mentioned earlier was chosen to perform the matched filtering operations. Certain considerations need to be made with respect to this choice, and these will be discussed here.

4.4.1 Windowing of the Data

Whenever an FFT is applied to a set of data points, an assumption is made that the points may be made periodic, without affecting the outcome of the FFT. This is not normally the case, and discontinuities are usually found at the endpoints, where the data points have been made to repeat. This gives rise to additional unwanted harmonics after performing the FFT, as shown by Harris[13]. To overcome this processing side-effect, a window if often applied to the data in the other domain to which harmonic reduction is desired.

The provision to apply various windows should thus be made available prior to performing the inverse FFT in the range and azimuth compression modules.

4.4.2 Zero Padding the FFT

The FFT algorithm used requires that the number of input and output samples be an exact power of 2. Since this is never the case, the remaining samples required to make up an exact power of 2 must be zero padded. This technique is discussed by Barber[14].

5. Software Design Implementation

The actual software implementation will be discussed in this section. The first part will deal with the control and parameter files used by the program as a whole, and this will be followed by a breakdown of the various modules.

5.1 Common Control Files

A number of control and parameter files were required in the implementation of the program. These files maintained control of the complete program, and allowed for the setting and passing of parameters.

5.1.1 Batch Processing File

This file is a result of the overall program design. Each of the main stages of the program are designed as individual stand alone modules that must be independently called from some controlling file. This controlling file will be called the Batch Processing File, and will consist of calls to the relevant modules.

This approach provides for flexibility in that the calling program is a standard text file that is not compiled, and can be easily modified for different processing scenarios. For example, if the range compression was performed by the radar hardware, then the call to that module could easily be left out.

After the calling of each module, a test is made on an error flag, which may be set by the module if an error occurred. This allows for the abnormal termination of the batch processing file.

An example of a batch processing file as used under the Novell Network Operating system is shown in figure 2. Note that the first command line parameter specified when running the batch file is used to specify the name of a user input parameter file. This parameter is then passed to each of the main modules when called from the batch file.

The error flag as used by the Disk Operating System (DOS) and set by each module is called the DOS errorlevel flag. This flag is tested after each module is run, to allow for abnormal termination.

```
rem Turn off broadcast messages from other stations:
castoff

rem Generate Parameters for the other modules:
satparam %1
if errorlevel==1 goto error

rem Perform Range Compression
rangecom %1
if errorlevel==1 goto error

rem Perform Corner Turn:
corner %1
if errorlevel==1 goto error

rem Perform Azimuth Compression:
azimcom %1
if errorlevel==1 goto error

rem Successful completion will end up here:
:complete
echo Successful Completion
goto fin

rem Unsuccessful completion will force a jump to here:
:error
echo Processing Error has occurred - Inspect the status file

rem Exit back to DOS:
:fin
logout
```

Figure 2 Batch Processing File Example

5.1.2 User Input Parameter File

The user input parameter file takes the form of an ASCII file of input parameters and values. This allows the user easily to specify most of the input parameters to the various program modules. Each of the parameters is specified with a parameter name and value separated with a space. An example of an input parameter file is shown in figure 3.

```
Miscellaneous Constants:

earthradius 6372070
lambda 0.234

Processing parameters:

rawname g:/sir-b-79.dat
rngname g:/sar/6000/6000rng.dat
corname g:/sar/6000/6000cor.dat
azmname g:/sar/6000/6000azm.dat
statusname status.doc
rawtype sirb
rangebins 2494
rangefftsize 4096
pulses 4096
azimfftsize 4096
startpulse 6000
samples 512
window 0

Shuttle position and other flight parameters:

xpos 282499
ypos -5637355
zpos -3419207
xvel 4763.469
yvel 3359.391
zvel -5143.152
look 30.4
slant 272670
prf 1463.2

Range reference parameters:

pulselength 30.4e-6
sampfreq 30.353e6
sweep 12e6
ifreq 7.2e6

Parameters used by the CFI image extractor:

cfiazimsize 1.64
cfirangesize 1.00

Parameters used by the Quickcfi image generator:

quickazimave 16
quickazimsize 5
quickrangesize 3
```

Figure 3 Example of User Input Parameter File

The input parameters may be broken down into a number of sections as will be described below. The parameter names that are used are shown in brackets next to the parameter descriptions.

i) Filenames and Processing Parameters

The following parameters include the filenames that will be used for the various intermediate files, as well as the raw data filename:

- Raw Data File Name	[rawname]
- Range Compressed File Name	[rngname]
- Corner Turned File Name	[corname]
- Azimuth Compressed File Name	[azmname]
- Status File Name	[statusname]

The next parameters define the position from which to extract data from the raw data file, as well as the desired size of the output files:

- Raw Data file type	[rawtype]
- Starting Azimuth pulse to extract	[startpulse]
- Number of Azimuth pulses to extract	[pulses]
- Number of Range bins to output	[rangebins]

The next two parameters should be specified to set the size of Fast Fourier Transform to be used in the Range compression and Azimuth compression modules. The sizes may be different, but should be a power of 2. The third parameter below may be used to specify a window type to apply to the data prior to the Inverse FFT. The window types supported are shown.

- Range Compression FFT size	[rangefftsize]
- Azimuth Compression FFT size	[azimfftsize]
- Window type to apply	[window]

0 - No window
1 - Raised Cosine
2 - Kaiser Bessel

ii) Range Compression Parameters

Also required are those parameters that will allow the range compression reference function to be re-created:

- Intermediate Frequency [Hz]	[ifreq]
- Sampling Frequency [Hz]	[sampfreq]
- Pulse length [S]	[pulselength]
- Sweep Rate [Hz/S]	[sweep]

iii) Satellite Parameters

The satellite parameters are those that relate to the satellite position, trajectory and radar look geometry. These parameters should normally be extracted from the header section of the raw data file, but may be specified here if the header data is not available.

- Average Satellite Position	[xpos,ypos,zpos]
- Average Satellite Velocity	[xvel,yvel,zvel]
- Radar Look Angle [deg]	[look]
- Slant Range to centre of swath [m]	[slant]

iv) Viewing Parameters

The parameters defining the range and azimuth scaling constants for the CFI image extraction module and for the quick look image extractor must be specified. These may be set to scale the output images to provide correctly proportioned pixels. Also to be specified is the number of pixels to average for the unfocussed quick look image extractor, which will be discussed later.

- CFI Azimuth Pixel sizing	[cfiazimsize]
- CFI Range Pixel sizing	[cfirangesize]
- Quick Look Azimuth Pixel Sizing	[quickazimsize]
- Quick Look Range Pixel Sizing	[quickrangesize]
- Quick Look Averaging	[quickazimave]

v) Miscellaneous Constants

Other miscellaneous constants required by the program must also be specified. These include:

- Radar Pulse Repetition Frequency [Hz]	[prf]
- Radar Transmit Wavelength [m]	[lambda]
- Earth Radius [m]	[earthradius]

5.1.3 Status File

The status file will be created by the first processing module that is run, and then updated by the others during their runs. It will hold useful output data, providing a complete summary relevant to the particular program run.

At the start of each module, the module description will be written to the parameter file. As each step is performed in the module, relevant details will be written to the status file, thus giving an up to date summary of what is happening. Any errors will be written to the status file immediately before aborting a program run, thus allowing complete postmortem analysis.

The status file will also include the times taken to complete each module. An example of the status file is shown in figure 4.

```
Start of the Range Compression Module.
======================================

Output File g:/sar/7650/7650rng.dat
Range Walk Compensation: 0.236328 metres per pulse
Starting azimuth pulse: 7650
Total azimuth pulses to process: 4096
Range bins output per azimuth line: 2494
Window type: 0

Output Scaling Factor: 0.394372
Maximum Walk Shift in Range Bins: 99
Maximum out of range errors: 59 at 9330
Maximum zero errors: 10 at 11560
Mean Output Magnitude: 41.3628
Average Max Position before IFFT: 0
Processing Time: 787 min
Range Compression Module Complete

Start of the Corner Turn Module.
================================

Input File: g:/sar/7650/7650rng.dat
Output File: g:/sar/7650/7650cor.dat
Corner Turn Dimensions: 4096 by 2494 (2 byte complex)
Available Memory Allocated: 393312 bytes
Corner Turn Time Taken: 66 minutes
Corner Turn Module completed.

Start of the Azimuth Compression Module.
========================================

Input Data file g:/sar/7650/7650cor.dat
Output Data file g:/sar/7650/7650azm.dat
Window type: 0
Output Scaling Factor: 0.108394
Maximum out of range errors: 349 at 1037
Maximum zero errors: 3582 at 2415
Number of Azimuth Reference points: 512
Processing Time: 470 min
Azimuth Compression Module Completed.
```

Figure 4 Example of Status File

5.2 Included Sub-Modules

A number of separate sub-modules are included by the main program modules. These modules have been separated from the main modules for a number of reasons. They may be included in any of the main modules where necessary, and may be accessed or replaced easily if modifications need to be made. Each of these sub-modules will be discussed here, and are listed in appendix B. The extension .H is used, as this is a standard extension used for files that are to be included in C code.

5.2.1 EXTRACT.H

This module provides the code for extracting and decompacting range lines from the raw data file. It is included as a separate module, to allow it to be replaced easily if a different type of raw data were to be used. It is included by the range compression module, which requires access to the raw data file.

The file consists of a single function, '**extract()**', which takes as input the range line to extract, and pointers to the file and to storage space where the range line may be stored.

5.2.2 WINDOW.H

This module provides the functions to multiply a given complex input array by one of a number of selectable windows. This module is separated to allow different windows easily to be set up or modified. It is included by the range and azimuth compression modules, to provide windowing facilities to the FFT routines.

The externally callable function, '**window()**', takes as input pointers to real and imaginary components of the data to be windowed, as well as the array size and type of window to apply.

5.2.3 SAR.H

This module contains functions specific to the processing, which include the Fast Fourier Transform implementation and other associated routines. This module is included by the range and azimuth compression modules.

The FFT algorithm implemented is based on the standard Power-of-Two Butterfly FFT algorithm, which is widely documented. A good description of the computations involved is given by Roberts and Mullis[15].

This FFT algorithm requires certain multipliers which are dependant on the number of points in the FFT, and the sine and cosine functions. These multipliers take a considerable amount of time to calculate, due to the calls to the sine and cosine functions. The multipliers remain the same for a given number of FFT points, so they may be computed once at the start of the program, and then stored and referenced during later calls to the FFT procedure.

To facilitate this, these tables are set up prior to calling the FFT procedure, '**fft()**', with a call to the '**loadsincos()**' procedure. This procedure takes as input the size of the desired FFT, and pointers to the lookup tables. The lookup tables are then set up with the sine and cosine multipliers.

Other functions found in this module include a bit reversal function, '**bitrev()**' which is used by the FFT procedure to reverse the bits in a given number. The procedure, '**multiply()**' is used to multiply two complex arrays of numbers, and takes as input pointers to the real and imaginary array components, as well as the array size.

5.2.4 PARAM.H

This module contains the functions used to access the parameter files. It allows specific parameters to be extracted from the files. It is included in all main modules that access any of the parameter files.

The functions '**getfloatparam()**' and '**getstringparam()**' may be used to extract floating point or string parameters from a parameter file. Function input parameters include the parameter name to extract, and the parameter file to use. An output parameter indicates the success of the extraction.

5.2.5 MATRIX.H

This module contains functions that relate to the manipulation of matrices. This module is included by the Parameter Setup Module, to compute the various parameters. It is separated only because the functions provided are all directly related to each other.

All matrix manipulations are based around three dimensional matrices, which are assumed throughout the module. This provides for faster and simpler matrix computations.

5.3 Main Program Modules

Each of the main program modules described in this section are self-standing programs, able to be run independently. They may thus be combined in different sequences depending on the requirements of the processing, as mentioned earlier under the section on the Batch Processing File.

A flowchart showing how the main modules and files would normally relate to each other is shown in figure 5. The modules described in this section are all shown on the flowchart, which may be used for reference purposes. Note that the User input parameter file is input to all of the modules, and thus appears more than once on the flowchart. The 'SATPARAM.EXE' module has been separated on the flowchart only for clarity.

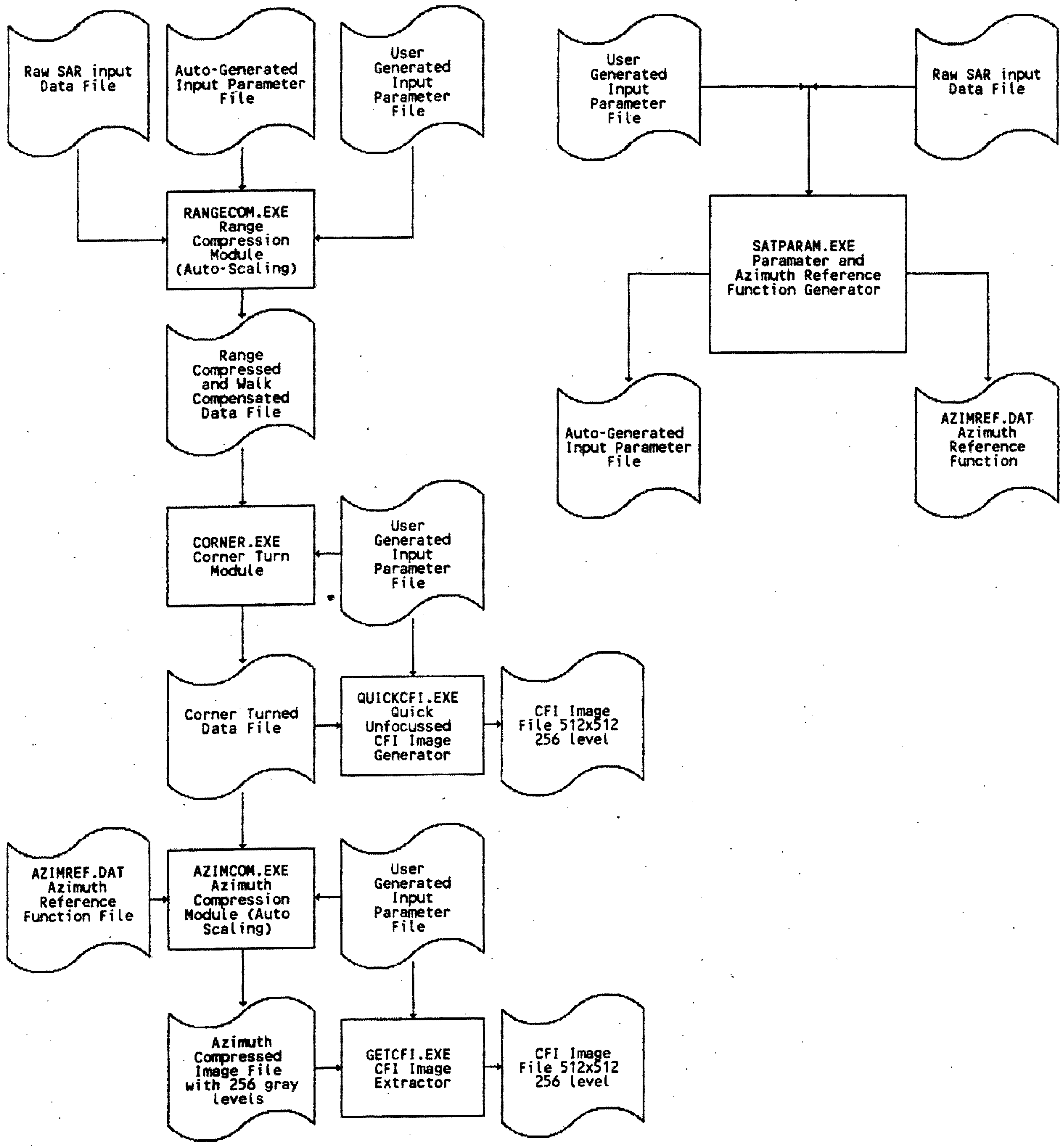

Raw SAR input Data File
Auto-Generated Input Parameter File
User Generated Input Parameter File
RANGECOM.EXE Range Compression Module (Auto-Scaling)
Range Compressed and Walk Compensated Data File
CORNER.EXE Corner Turn Module
User Generated Input Parameter File
Corner Turned Data File
QUICKCFI.EXE Quick Unfocussed CFI Image Generator
CFI Image File 512x512 256 level
AZIMREF.DAT Azimuth Reference Function File
AZIMCOM.EXE Azimuth Compression Module (Auto Scaling)
User Generated Input Parameter File
Azimuth Compressed Image File with 256 gray levels
GETCFI.EXE CFI Image Extractor
CFI Image File 512x512 256 level
User Generated Input Parameter File
Raw SAR input Data File
SATPARAM.EXE Paramater and Azimuth Reference Function Generator
Auto-Generated Input Parameter File
AZIMREF.DAT Azimuth Reference Function

Figure 5 SAR Processing Flowchart

5.3.1 Parameter Setup Module, 'SATPARAM.C'

Data extracted from the raw data file header and the input parameter file will be used to compute constants that will be passed to the other modules.

The function of this module will be to extract the header and ephemeral data from the raw data file or from the user generated input parameter file. The output from this module will take the form of another parameter file holding computed constants to be passed to the other modules, and the azimuth compression reference function, to be passed to the azimuth compression module.

The range walk will be computed in metres per pulse. This can be done, as mentioned earlier, by considering the sideways movement of the earth relative to the radar, and the radar PRF. The pixel spacings in range and in azimuth are also computed using the techniques described earlier.

Computation of the azimuth reference function requires the forward ground velocity of the radar as well as the distance to the target.

These computations all require the computation of the same constants, the main one being the relative movement of the target with respect to the radar. For this reason, they have all been grouped together into a single parameter setup module. The computations required were determined by Kritzinger[16], and implemented directly from the formulae provided.

The main section of this module computes an array of distances to a target based on the relative radar and target movements. This array is then output as the azimuth reference function, and the range walk is computed directly from it. The swath velocity, or forward ground velocity, is used to compute the azimuth pixel size.

The required input parameters will be taken from the raw data header and the input parameter file, and will include:

- Average Satellite Position	[xpos,ypos,zpos]
- Average Satellite Velocity	[xvel,yvel,zvel]
- Pulse Repetition Frequency [Hz]	[prf]
- Radar Look Angle [deg]	[look]
- Slant Range to centre of swath [m]	[slant]

The output parameter file will include the following computed constants to be used by the other modules:

- Range Walk [m/pulse]	[rangewalk]
- Range Pixel Size [m]	[rangepixelsize]
- Azimuth Pixel Size [m]	[azimpixelsize]

The azimuth compression reference function will be output to a complex floating point numeric file, stored in ASCII format, called 'azimref.dat'.

5.3.2 Range Compression Module, 'RANGECOM.C'

The main functions of this module are to extract the raw data from the input data file, to compress the data in range, and then to output the compressed data to the range compressed data file. This module may be omitted in the processing if range compression is not required.

To compensate for the effects of range walk, each output range line is shifted by a given amount in the range compression module before being written out to disk.

The raw data, in the case of the SIR-B, was sampled at an intermediate frequency. This requires that it be demodulated to baseband, which is achieved with a shift of the samples in the frequency domain, prior to performing the inverse FFT.

The extraction of the range lines, the range compression and the range walk compensation operations are all combined to speed the processing and also to reduce disk space requirements.

i) Basic Processing Steps Performed

The function '**compress_range_line()**' included in this module is called to extract, demodulate and compress a single range line. This includes the following steps:

- Extract range line from the raw data file.
- Perform FFT on the extracted range line.
- Multiply by the FFT'd and conjugated range reference function as set up.
- Zero the upper half of the samples.

- Shift the samples to effectively demodulate from the Intermediate frequency to baseband.
- Perform Inverse FFT.

The Basic processing steps performed by the main section of the module may be summarised as follows.

- Extract Input Parameters from specified Parameter file.
- Generate Range Reference Function and form the conjugated FFT of it.
- Perform Sample range compressions to determine scaling constant for output file.
- Perform actual range compressions and output the data with the required range walk shift.

ii) Memory Requirements

Since each range line is compressed separately, the memory required by this module should be enough to allow a range compression to be performed on a single line. This is determined by the size of FFT chosen and the floating point format used. Normally, a 4096 point FFT is performed using complex 4 byte floating point numbers, as follows:

(4096 point) * (4 byte float) * (2 complex) = 32K

To store the range reference function, input range line and output, three times this amount is required:

3 * 32K = 96K

iii) Input Parameters and Data Files

At the start of this module, the required input parameters are extracted from the user input parameter file and from the computed parameter file set up by the parameter setup module.

The Raw data file as specified in the input parameter file is processed.

iv) Output Parameters and Data Files

The output from this module is a complex data file, with filename as specified in the input parameter file. Output Dynamic Range is 8 bits real and 8 bits imaginary, both fixed point.

5.3.2 Corner Turn Module, 'CORNER.C'

This module takes as input a complex data file that is to be rotated by 90 degrees, to allow processing in the other direction. This is required to allow for the efficient accessing of the data by the azimuth compression module.

i) Basic Processing Steps Performed

The method of corner turn used was chosen to be as fast as possible by minimising disk accesses. This is achieved by dynamically allocating as large a section of memory as possible. Input data is then read from the disk in blocks the size of available memory, as depicted in figure 6.

As can be seen in the input file diagram, the lines making up the blocks (Block Lines 1,2... as depicted in block I) are not stored sequentially and must be read in one line at a time, randomly. Once a complete block has been read into memory, it is written out one output line at a time, in the order as shown in the output file diagram. This effectively entails a memory corner turn as the output block is written to the disk. The process is repeated for as many blocks that are required to corner turn the complete input file.

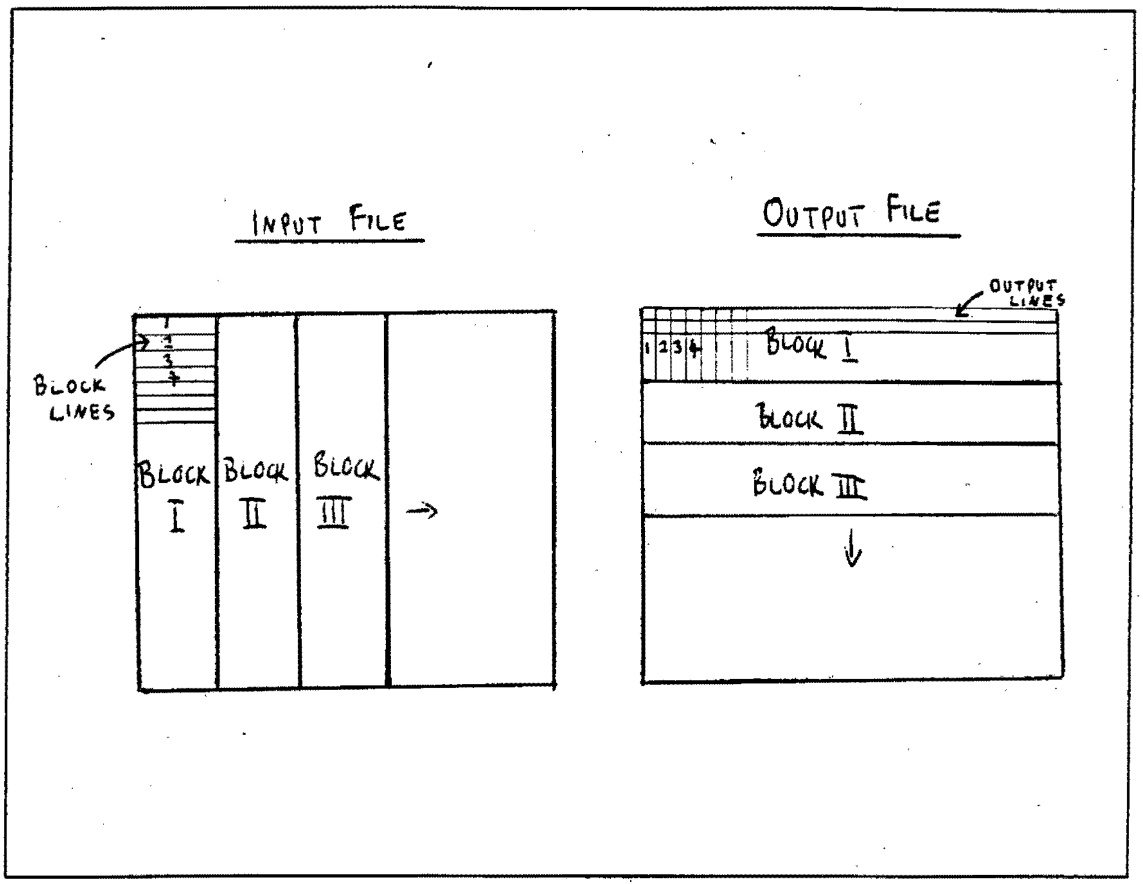


Figure 6 Corner Turning Operation

The basic processing steps for the corner turn module may be summarised as follows:

- Extract parameters from the specified parameter file.
- Allocate all available memory for corner turn.
- Perform the Corner Turn.

ii) Memory Requirements

This module requires as much free memory as possible, due to the method of corner turn used. All available memory is used, and speed is directly dependant on the amount available.

iii) Input Parameters and Data files

The input parameters are extracted from the user input parameter file, and include the size of block to corner turn. The range compressed input data file of complex numbers, is passed from the range compression module.

iv) Output Parameters and Data files

The output from the corner turn module will be a corner turned complex data file with filename as specified in the input parameter file. It will be ready to be processed in the azimuth direction.

5.3.3 Azimuth Compression Module, 'AZIMCOM.C'

This module applies the Synthetic Aperture Radar Azimuth compression to an input file. This is required to effectively sum the radar returns in the Synthetic Array.

i) Basic Processing Steps Performed

The function '**compress_azim_line()**' included in this module is called to extract and compress a single azimuth line. This includes the following steps:

- Extract azimuth line from the corner turned data file.
- Perform FFT on the extracted azimuth line.
- Multiply by the FFT'd and conjugated azimuth reference function as set up.
- Perform Inverse FFT.

The Basic processing steps performed by the main section of the module may be summarised as follows.

- Extract Input Parameters from specified Parameter file.
- Read in the azimuth reference function and form the conjugated FFT of it.
- Perform Sample azimuth compressions to determine scaling constant for output file.
- Perform actual azimuth compressions and output the data.

ii) Memory Requirements

Memory requirements are similar to that of the range compression module, as a single azimuth compression is performed at a time. Thus for an FFT size of 4096, the memory required will be 96K.

iii) Input Parameters and Data files

The reference function used to apply the azimuth compression will have been generated by the parameter setup module, and will be passed to the azimuth compression module in the form of the "Azimref.dat" file. Input parameters will also include the corner turned data file, and the file dimensions.

iv) Output Parameters and Data files

The output from this module will be the main image file, of filename as specified in the input parameter file, from which sub-images may be called up using one of the viewer or printer modules.

5.3.4 Image Extractor Module, 'GETCFI.C'

This module has been provided to extract an image of a given size from the main output image file, to allow it to be sent to one of the viewer or printer modules. The extracted output image will be of standard CFI type as defined earlier.

The Input Parameters passed from the input parameter file include the desired azimuth and range scaling factors for the output images. The position of the top left corner of the output image in the main image file may be specified on the command line.

The output image will be of fixed and standard size, so the greater the size of image selected, the lower the output resolution will be.

5.3.5 Quick Look Image Extractor, 'QUICKCFI.C'

The quick look image extractor module has been provided to extract an image directly from the range compressed and corner turned data file. Thus no azimuth compression is required prior to extracting an image with this module.

The technique used is to apply unfocussed azimuth processing while extracting the image. Thus a running average of a number of azimuth pulses is taken. The azimuth resolution obtained is not nearly as high as that obtained with focussed azimuth compression, but it allows major features of the image to be identified before full azimuth compression.

The input parameters passed from the input parameter file include the desired azimuth and range scaling factors of the output image, as well as the number of pulses to average in azimuth. The image position to extract may be specified on the command line. The output will be a standard CFI image as defined earlier.

5.4 Other Program Modules

Other modules not central to the actual processing will be mentioned here. These include modules included from elsewhere and written by others.

5.4.1 Image Viewer and Filter Modules

Image viewer and filter modules were taken from a collection of image processing and viewing utilities, developed by A. Dacks. These were developed as part of a Masters Thesis at the University of Cape Town.

The viewer module used is one that is compatible with the Standard IBM VGA format. It takes as input a standard CFI file as defined, and displays it on a standard VGA monitor using 32 gray levels.

The filter modules included a low pass image filter and median filter. These were applied to the output images to observe their effect, which will be discussed in a later section.

5.4.2 Laser Printer Module, 'PRINTCFI.C'

Producing hard copies of the output images requires either a printer capable of producing a range of gray levels, or a fine resolution printer on which gray scaling may be simulated using different dot densities. The latter approach was decided upon due to the availability of a 300 Dots per Inch Hewlett Packard Laserjet.

The module developed takes as input a standard 512 by 512 pixel CFI type file as defined earlier, and produces a 6.8 Inch by 6.8 Inch Hard Copy. The 256 gray levels of the input image are divided into 16 possible dot combinations each of which are made up with a 4 by 4 matrix of dots. The images presented later were printed using this module.

This routine is listed in Appendix C, and was developed jointly by the Author and A. Knight.

6. Software Testing

Program testing was required at all stages to maintain the correct course of development, and to allow for error location and correction. The testing may be broken down into the testing of the program as a whole, and the individual testing of modules. The individual module testing will be discussed first, followed by the complete program testing.

A number of additional programs were used during the testing stages to generate test files and to test or view output files. All programs used for testing are listed in Appendix D.

Included in these programs is one that was used to convert from the compact binary file format, that is used to store all data, to standard ASCII format. The ASCII files were then easily imported into a spreadsheet program to allow for viewing or graphing. The spreadsheet program used for this purpose was Quattro Pro Version 2 which is referred to in this section.

6.1 Individual Module Testing

The major program modules were each tested separately before being combined to form the complete program.

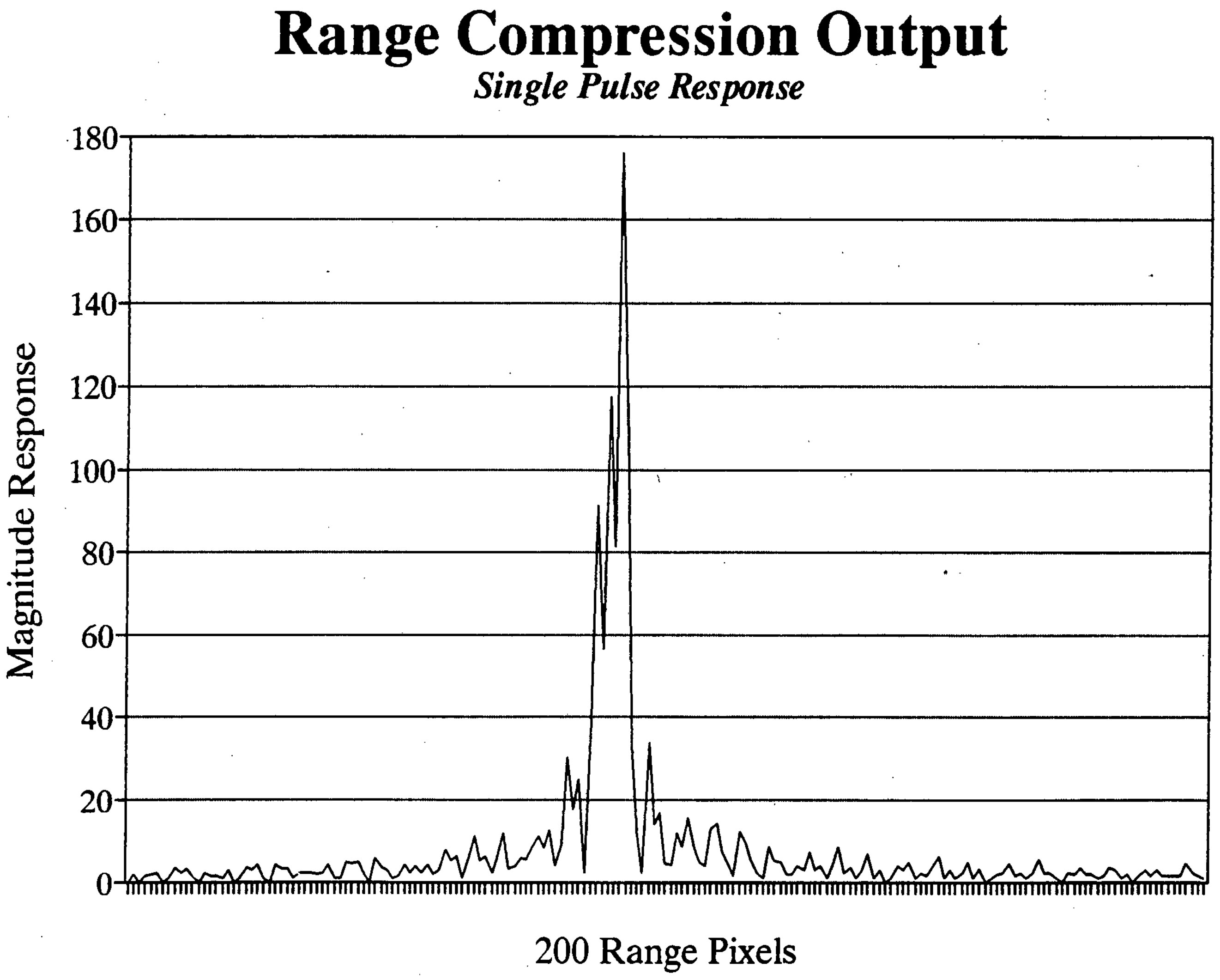
6.1.1 Range Compression Testing

The range compression module was tested by generating and inputting an uncompressed test file consisting of a single range line. This test file was stored with the same format as used by the raw data file which is normally input to this module.

A number of uncompressed test range lines were used, these included one with a single point reflector, and one with two point reflectors positioned in close proximity. The compressed output range line with a single point reflector is shown in figure 7. The resulting point target response is slightly spread out, as predicted by Kritzinger[17], due to the fact that the process of demodulation results in an oversampled signal.

The program used to generate the test file is listed in appendix D.

Figure 7 Range Compression, Single Point Target Response



6.1.2 Corner Turn Testing

The corner turn module required that the complex input file be corner turned and output to the complex output file.

A test file was initially generated holding a sequence of numbers in one row, and zeros in all other rows. This file was large enough to require being broken down into a number of smaller blocks during the corner turn. The file was then corner turned, and the output file manually inspected to ensure the correct placement of the transposed row, and the presence of zeros elsewhere.

A program to compare the input and output files to the corner turn module was also written. This program compared every number in the corner turned output file with the corresponding number in the input file. Any errors were signalled, to allow debugging of the program to take place. This test was run on the actual corner turned file used during the processing, to confirm the absence of any errors.

The programs to generate the test file, and to compare the input and output files are listed in Appendix D.

6.1.3 Azimuth Compression Testing

i) Main Module Testing

This module was tested by generating a test file of the size used during actual processing. The file contained one line with a single point target response, and another with two point target responses in close proximity. The rest of the file was filled with zeros.

The test file was then azimuth compressed and the output file inspected. The two test azimuth lines in the file were extracted, converted to ASCII and imported into the Quattro spreadsheet. The azimuth compressed output with two point targets is shown in figure 9 which also shows the effects of the windowing as discussed later. The outputs appear correctly spaced with narrow main lobes, as expected, verifying the correct operation of the azimuth compression module.

The program to generate the azimuth compression test file, is listed in Appendix D.

ii) Windowing Testing

The testing of the windowing was done in conjungtion with the testing of the azimuth compression for convenience, but the windows are used by both the range and azimuth compression modules. Each of the windows implemented were initially output to ASCII files and viewed on Quattro, to verify that they appeared correct.

The windowing of the data in the frequency domain was tested by outputting the waveforms at various stages during the processing. The azimuth compression reference was output before and after multiplication by a window in the frequency domain. This is shown in figure 8. As can be seen, the window has the effect of reducing the magnitude of the extreme positive and negative frequencies.

The effect of the window on the azimuth compression output was then tested with a number of inputs. These included a single point target and a double point target separated by a distance of 10 points. The output for the double point target is shown in figure 9 with no window, and with the Kaiser window. The slight side-lobe reduction and broadening of the main lobes on the windowed output can be observed quite clearly.

Figure 8 Azimuth Reference Function before and after windowing

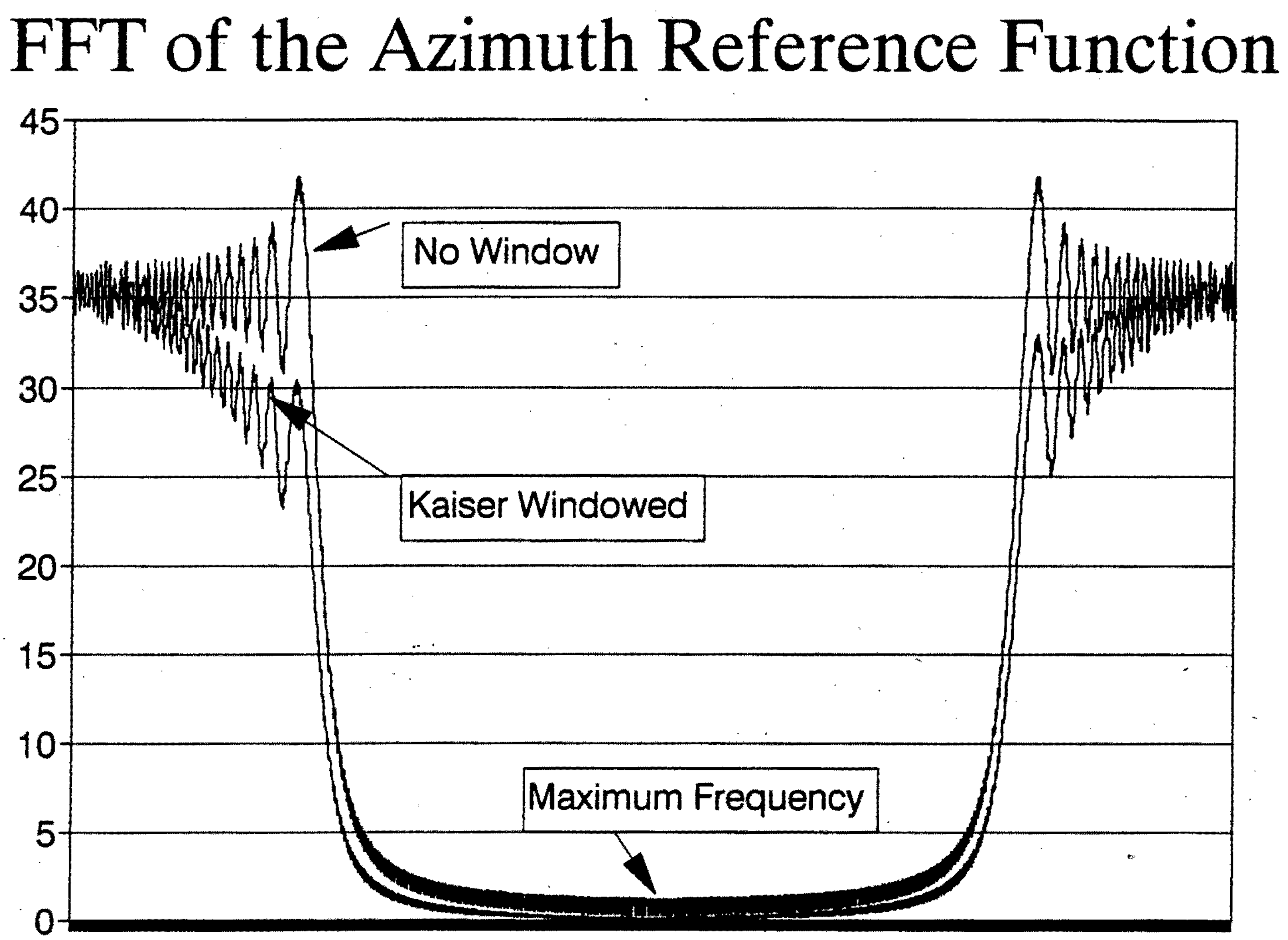
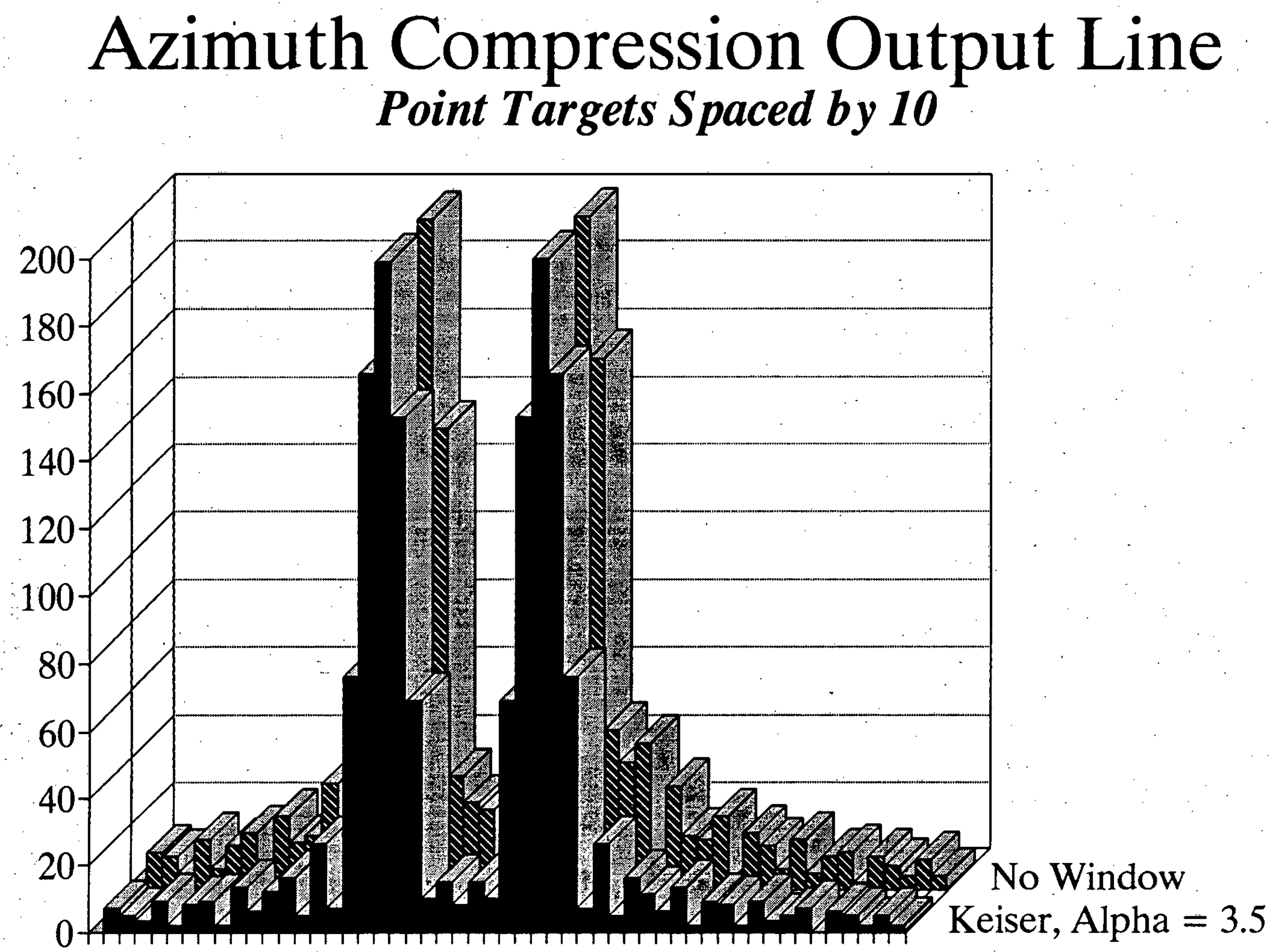


Figure 9 Azimuth Compression, Double Point Target Response



6.1.4 Testing of Fixed-Point Output Scaler

The testing of the automatic output scaler was vital to ensure that the limited 8 bit dynamic range was being used to its fullest, and that a minimum of truncation errors were produced.

A separate analysis program was written for the purpose of analysing the fixed point output data. This program reads in a file of any length made up of 8-bit bytes. An ASCII summary of the frequency of occurrence of each byte is produced, as well as mean and standard deviation values.

This ASCII output summary was then imported into Quattro to allow a graphical view of the output byte frequencies to be produced. The output data frequencies from the range and azimuth compression modules were graphed and printed as shown in figures 10 and 11.

Figure 10 Range Compression Output Distribution

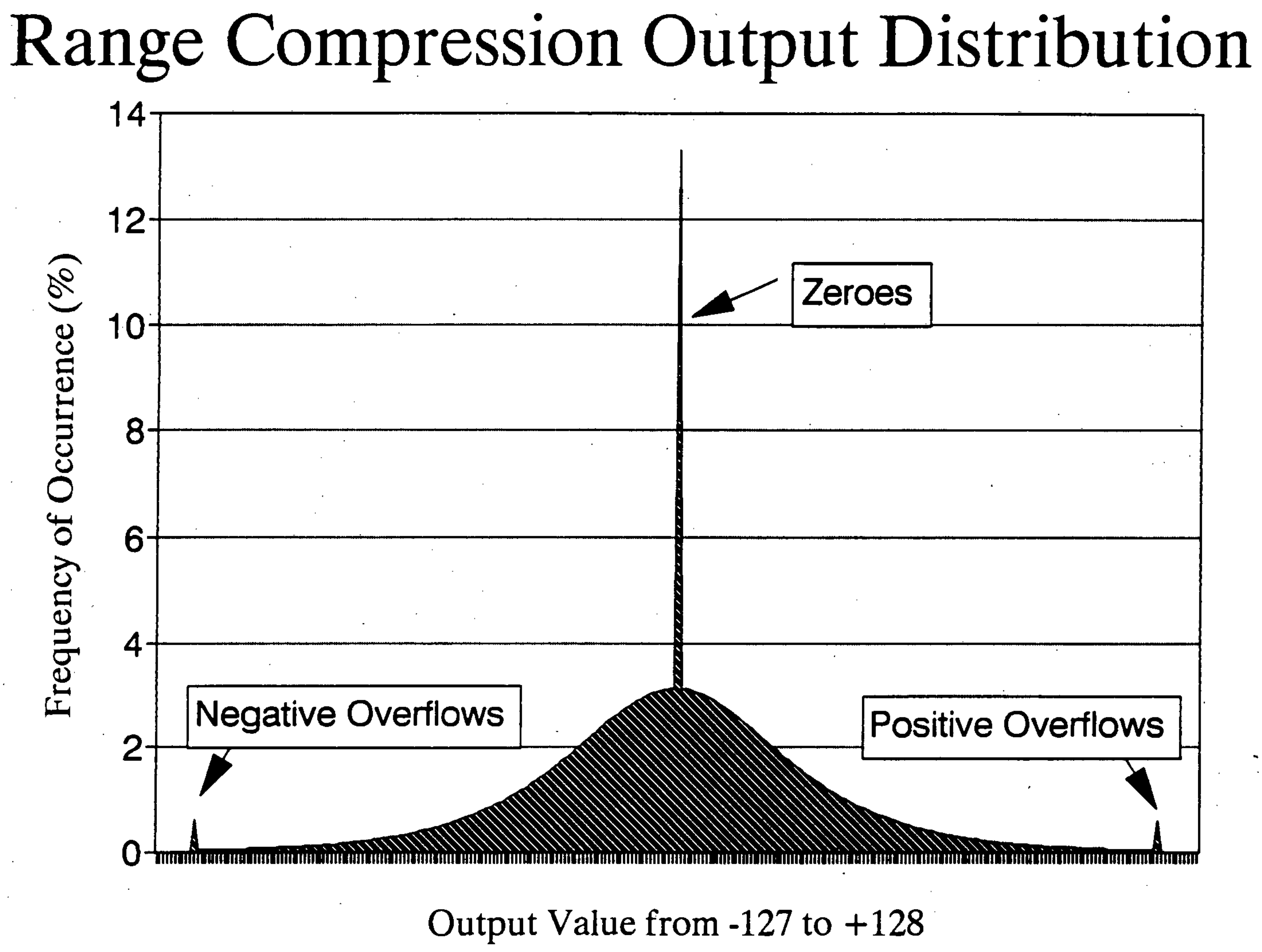
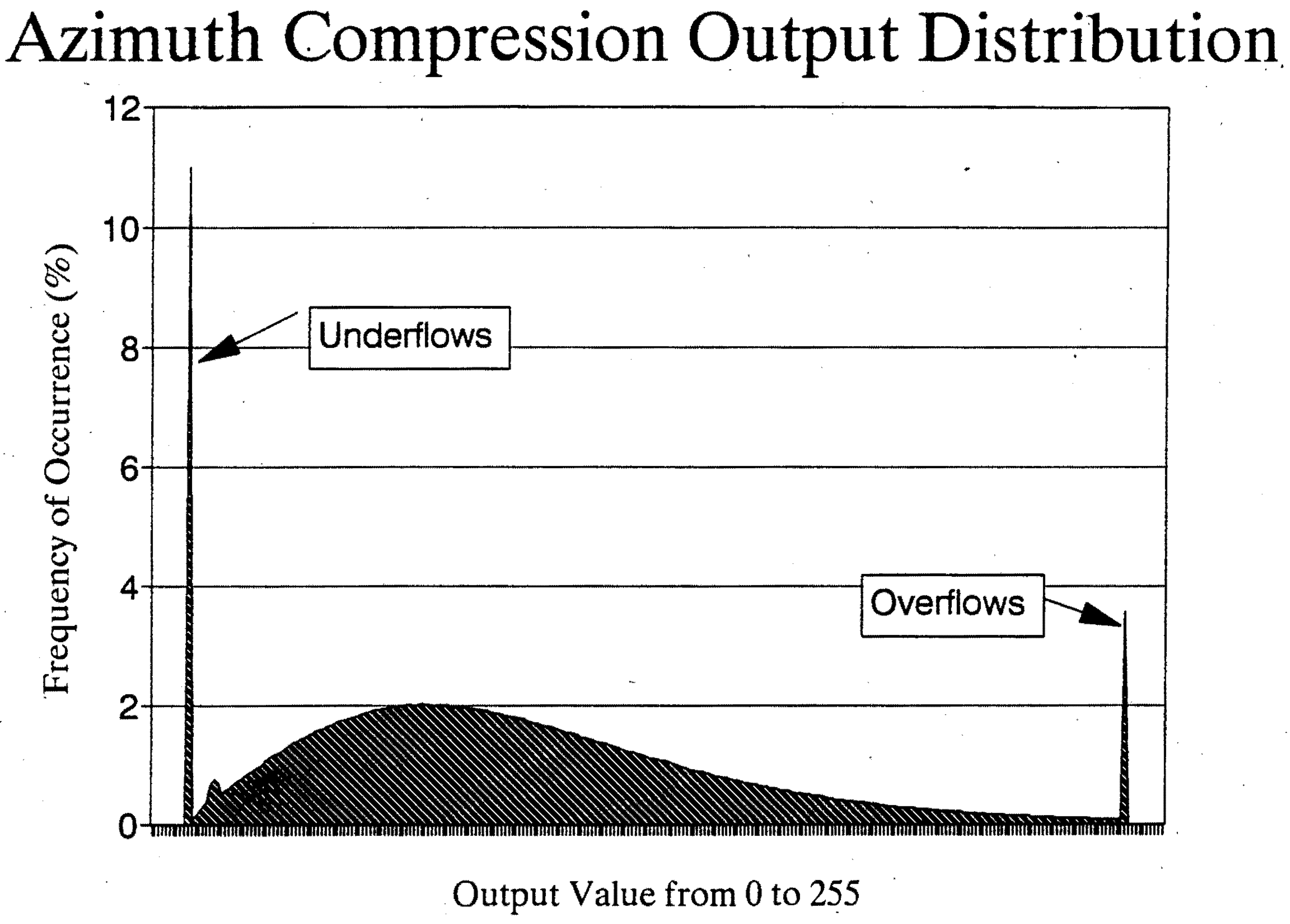


Figure 11 Azimuth Compression Output Distribution



i) Range Compression Output

The range compression output consists of signed complex numbers with each component in the range -127 to +128. The central spike on the graph represents the number of zeroes, and the spikes at the left and right extremes represent the number of negative and positive overflow errors which were truncated to the maximum or minimum value. A trade-off between the number of zeroes and the number of truncation errors was required in the output scaling. The chosen scaling as shown was decided upon after visual inspection, to provide an as even as possible distribution.

The large number of zeroes in both the range compression and azimuth compression outputs can be ascribed to the fact that zero padding was used when shifting the output data to compensate for range walk. This means that a number of zeroes were required around the actual data, so as to square it off.

ii) Azimuth Compression Output

The output from the azimuth compression module consists of unsigned bytes in the range 0 to +255. As can be seen the 8 bit range is well utilised, although a number of underflows and overflows do exist. The presence of the zeroes, or underflows, was described above, and the overflows are unavoidable using linear scaling. The overflow errors do not detract from the image quality much, as most of the values used are in the lower end of the 8 bit range, and a tapering effect toward the upper range is present.

6.2 Complete Program Testing

Complete testing was made quite difficult by the length of time required to perform all of the SAR processing steps. To reduce the time required, smaller blocks from the raw data file were processed for test purposes. The output images were then compared with the images generated by the Jet Propulsion Lab SAR processor. The effects on the image of variations in particular input parameters could then also be observed. Example images are shown in the next section.

The generation of a simulation test file to be sent through all processing stages was not thought necessary. Such a simulation file could be produced by reversing the SAR processing process as implemented. This would thus not provide testing of the algorithms, but simply of the program itself which had already undergone extensive testing on a modular basis.

7. Output Images and Post-Processing

The actual output images obtained from the various stages of the program will be presented here. Also to be mentioned here are the effects of a couple of common image enhancement procedures that were applied to the output images.

All images presented were printed on a laser printer using the routine mentioned earlier.

7.1 Azimuth Compression Output Images

Example image A, as shown in figure 12, was extracted from the main output image file. The pixels have been scaled such that they are square and approximately 10m by 10m. The figure thus represents an area of approximately 5km by 5km, since it was produced from a standard 512 by 512 pixel CFI image.

7.2 Reduction of Speckle Noise

The rapid and random intensity variation from pixel to pixel in close proximity is known as speckle. This is caused by random variations in radar reflectivity due to the roughness of the surface being mapped, and does not indicate any macro variations in radar reflectivity. A complete definition of Speckle is given by Wehner[18].

A commonly used way of reducing speckle is to apply multi-look processing, which involves the overlaying of images generated from different sections of the synthetic array. This process was used in the generation of the JPL images, but was not implemented as part of this thesis.

Alternative methods of reducing speckle and generally improving the image quality were investigated. These included low pass and median filtering of the image.

7.2.1 Low Pass Filtering

Low pass filtering was the first post-processing enhancement applied to the image. The effect is shown in figure 13. As can be seen, the overall resolution has been degraded, but the image appears far smoother and continuous.

7.2.2 Median Filtering

Median filtering was also applied to the original image in figure 12 and the result is shown in figure 14. As can be seen, the image appears far more discretised with definite areas of light and dark.

7.3 Quick Look Output Image

Output images from the Quick Look image extractor were generated and compared with those extracted from the main image file generated by the azimuth compression module. The image extracted is shown in figure 15.

Since no azimuth processing was applied, apart from the simple averaging of the radar returns, the azimuth resolution has been severely degraded. The image nevertheless still displays a definite correlation with the fully processed one, verifying the correct operation of the Quick Look image extractor.

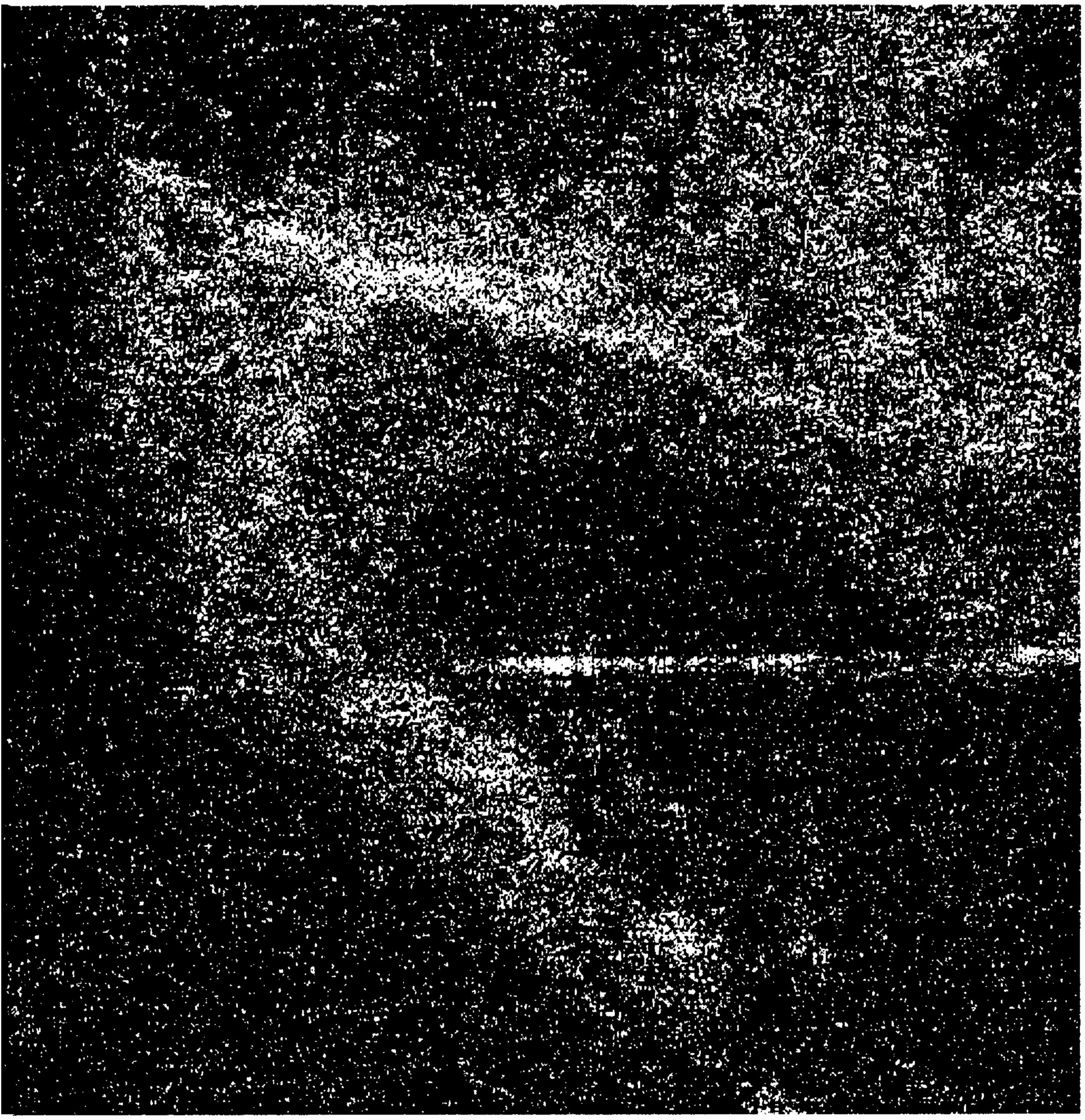


Figure 12 Full Resolution Image A

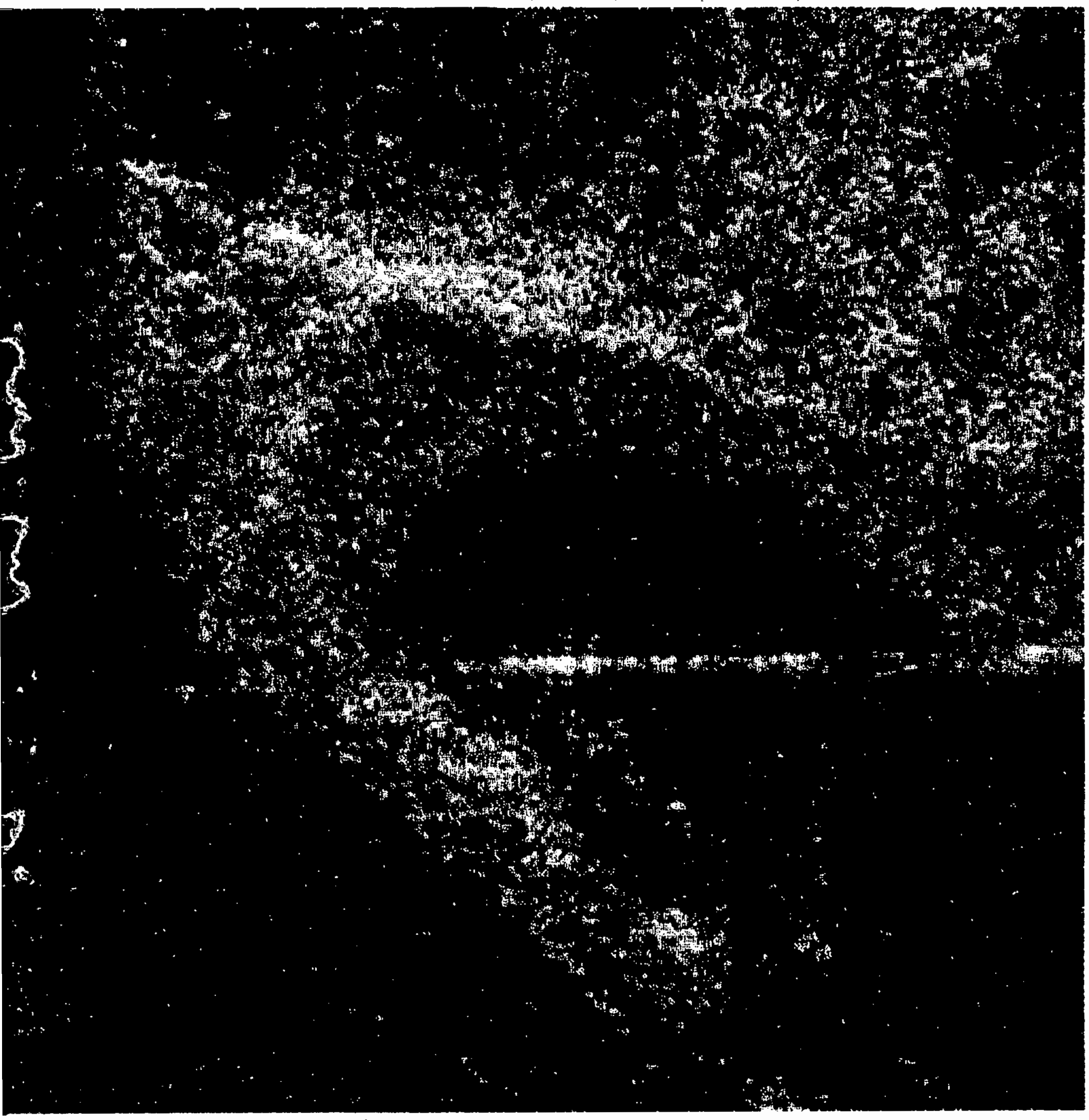


Figure 13 Low Pass Filtered Image A

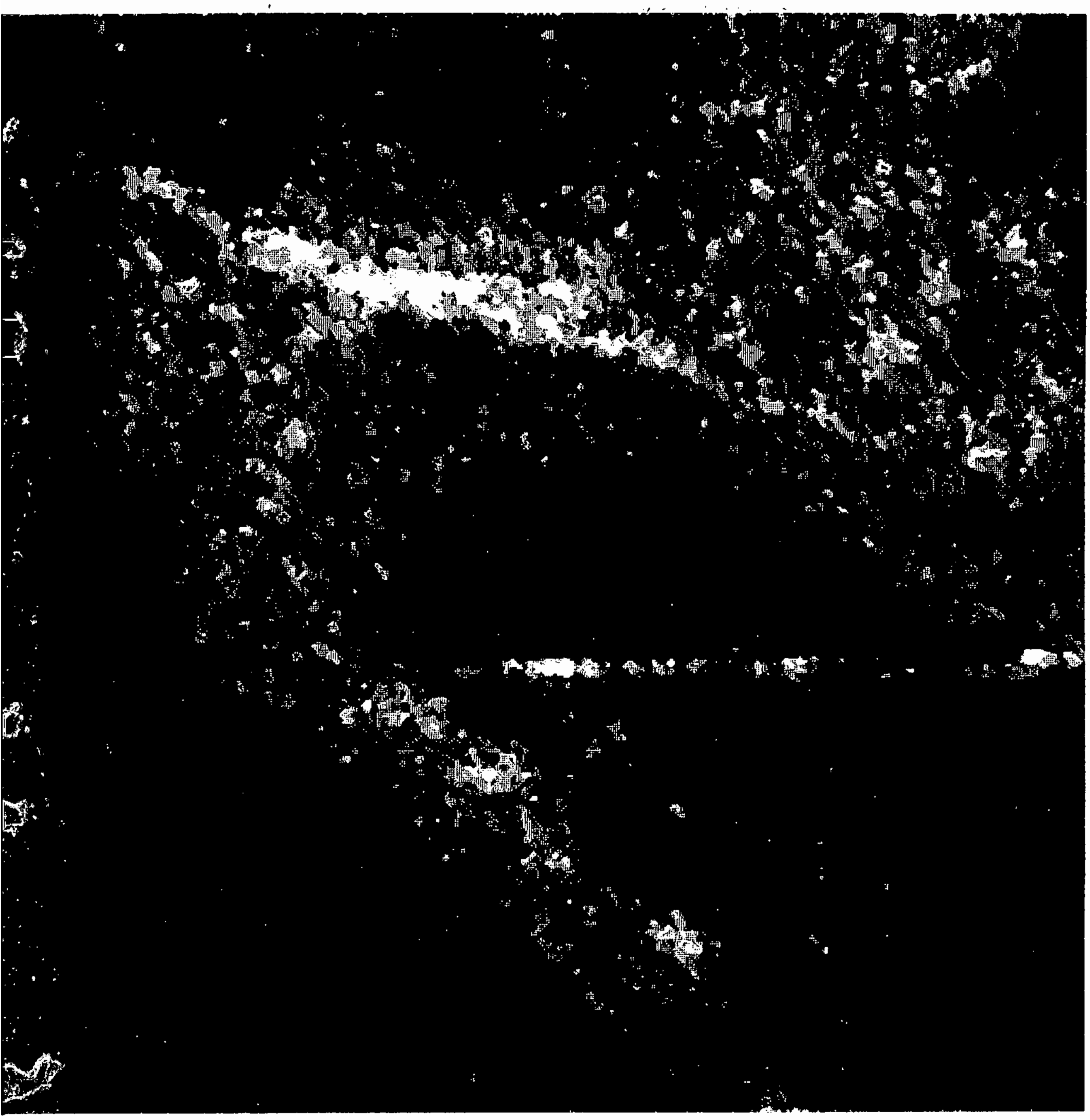


Figure 14 Median Filtered Image A

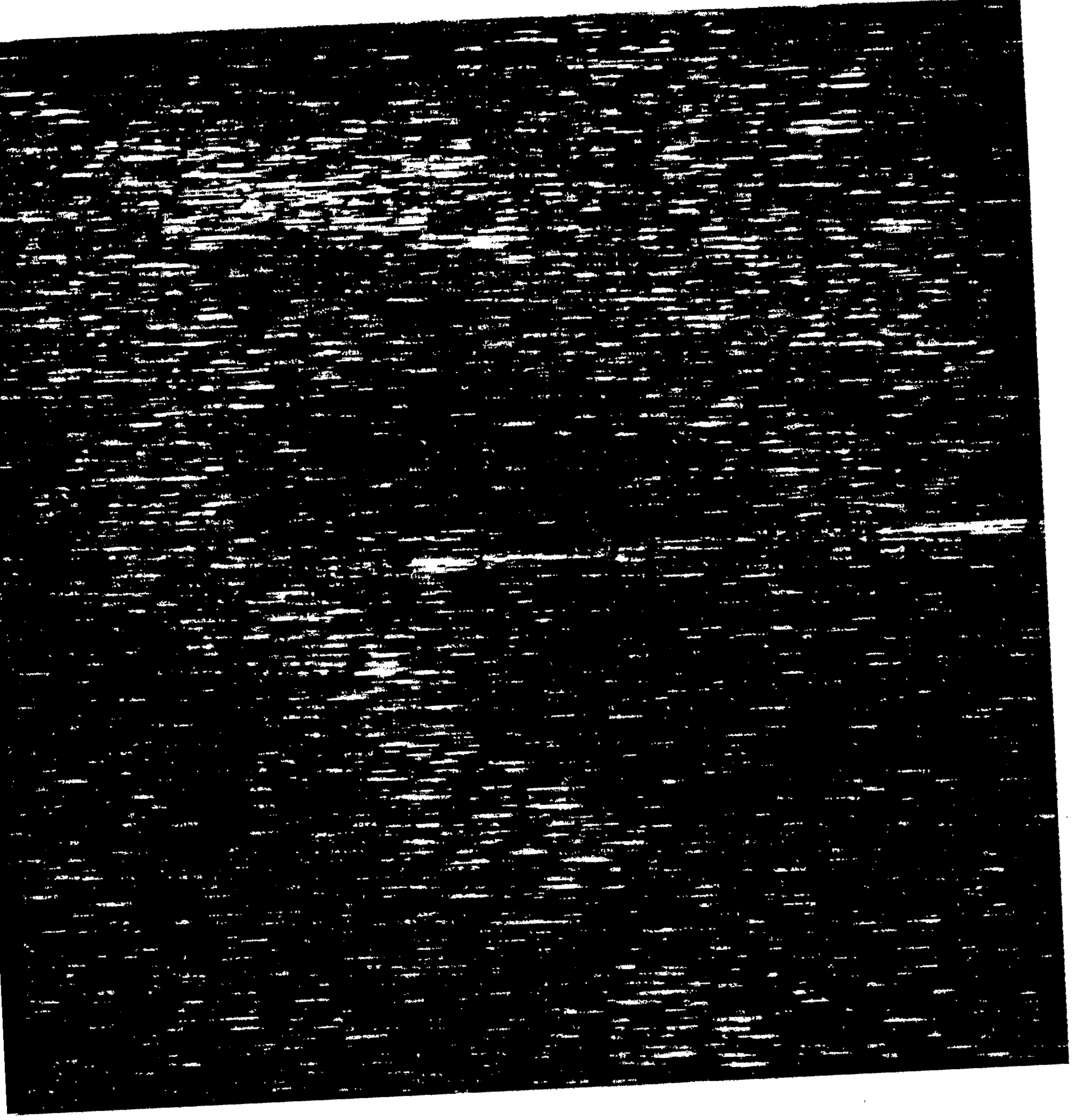


Figure 15 Quick Look of Image A

7.4 JPL SAR Processor Output Images

The JPL images were provided in the form of a large gray level file, from which standard CFI images were extracted.

For comparative purposes, a single CFI image was extracted from the JPL processed file and included here. It is shown in figure 16, and may be compared with the images in figures 17, 18 and 19. The first of these images was extracted directly from the azimuth compressed image file, and the others are low pass and median filtered version of this.

Both the low pass and median filter approaches, appear to provide a similar effect to multi-look processing when compared with the JPL generated image. Experimentation with different types of filter could probably provide better results.

The JPL processing also includes compensation for range curvature, which allows the use of a longer synthetic array without introducing phase correction errors. This, apart from the multi-look processing, is probably another reason for the JPL image appearing more defined.

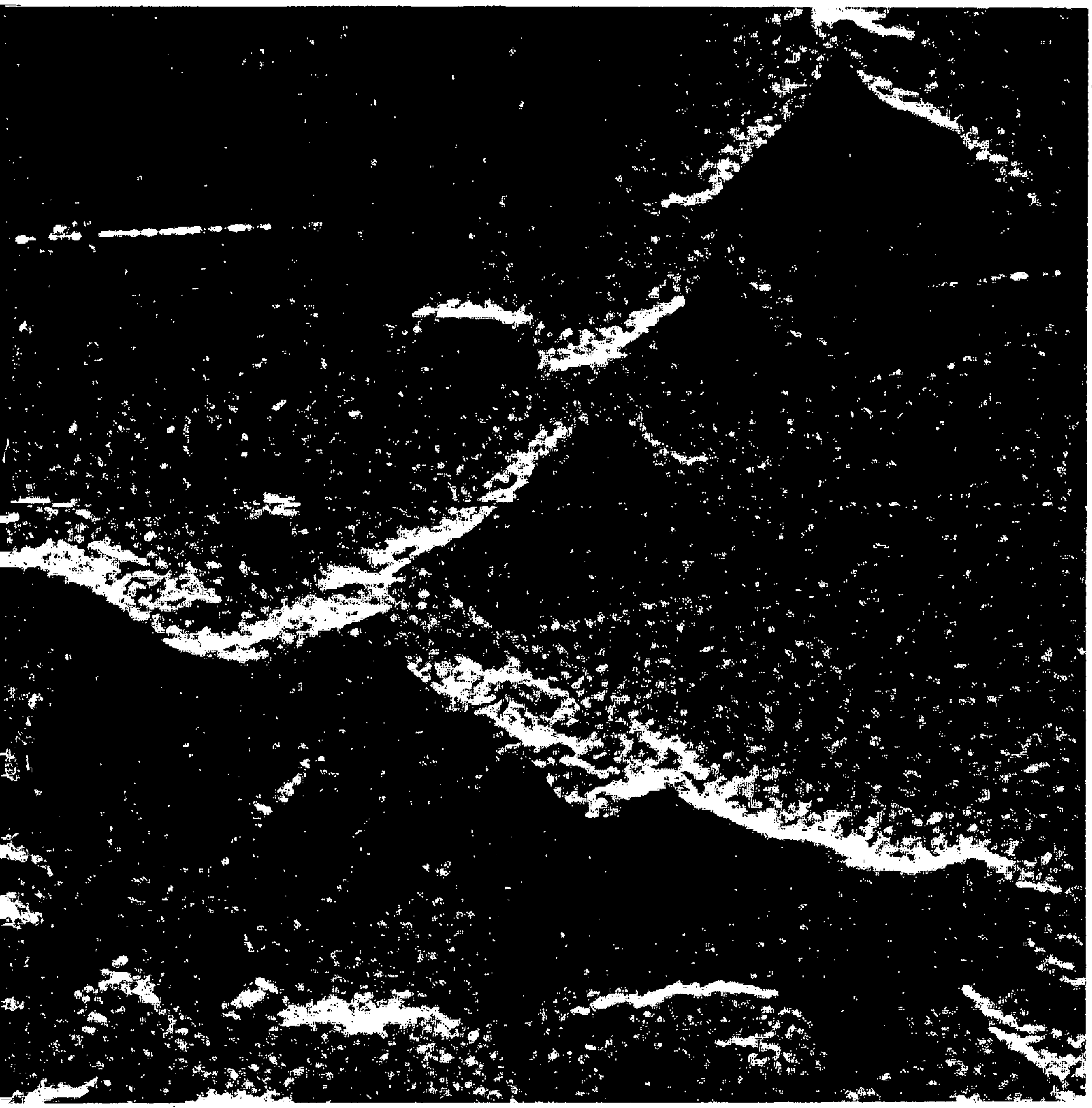


Figure 16 JPL processed Image B

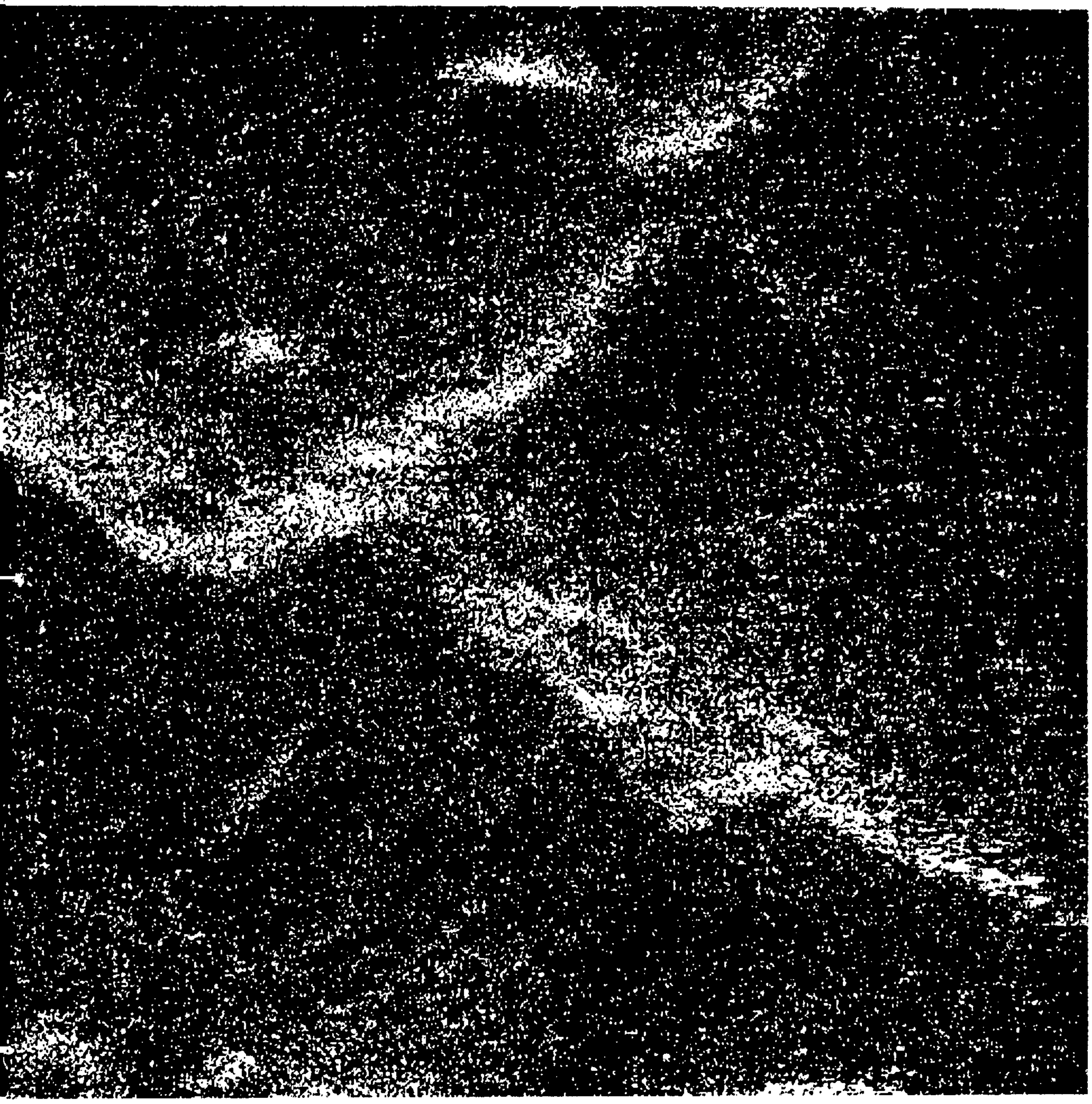


Figure 17 Full Resolution Image B

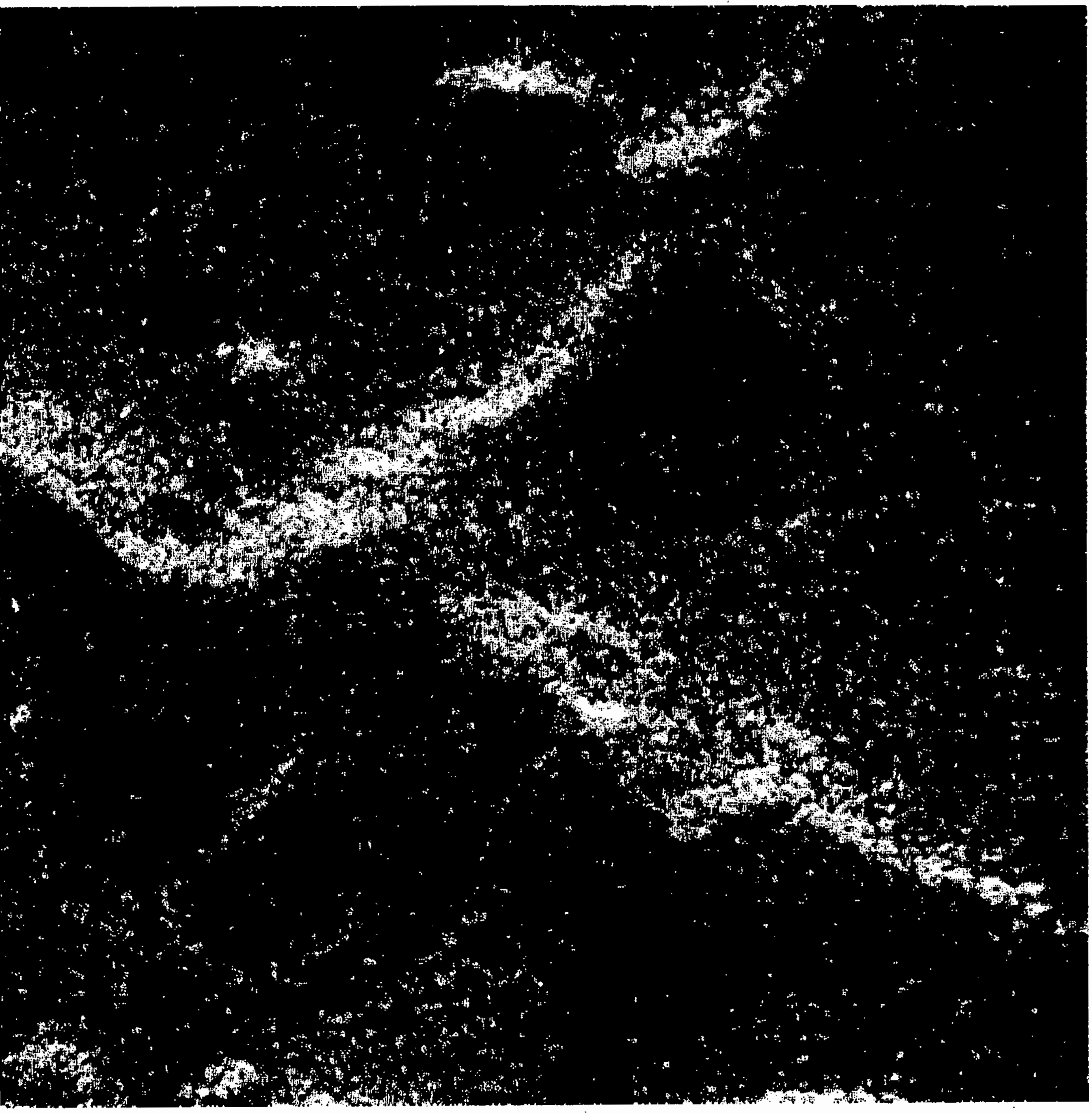


Figure 18 Low Pass Filtered Image B

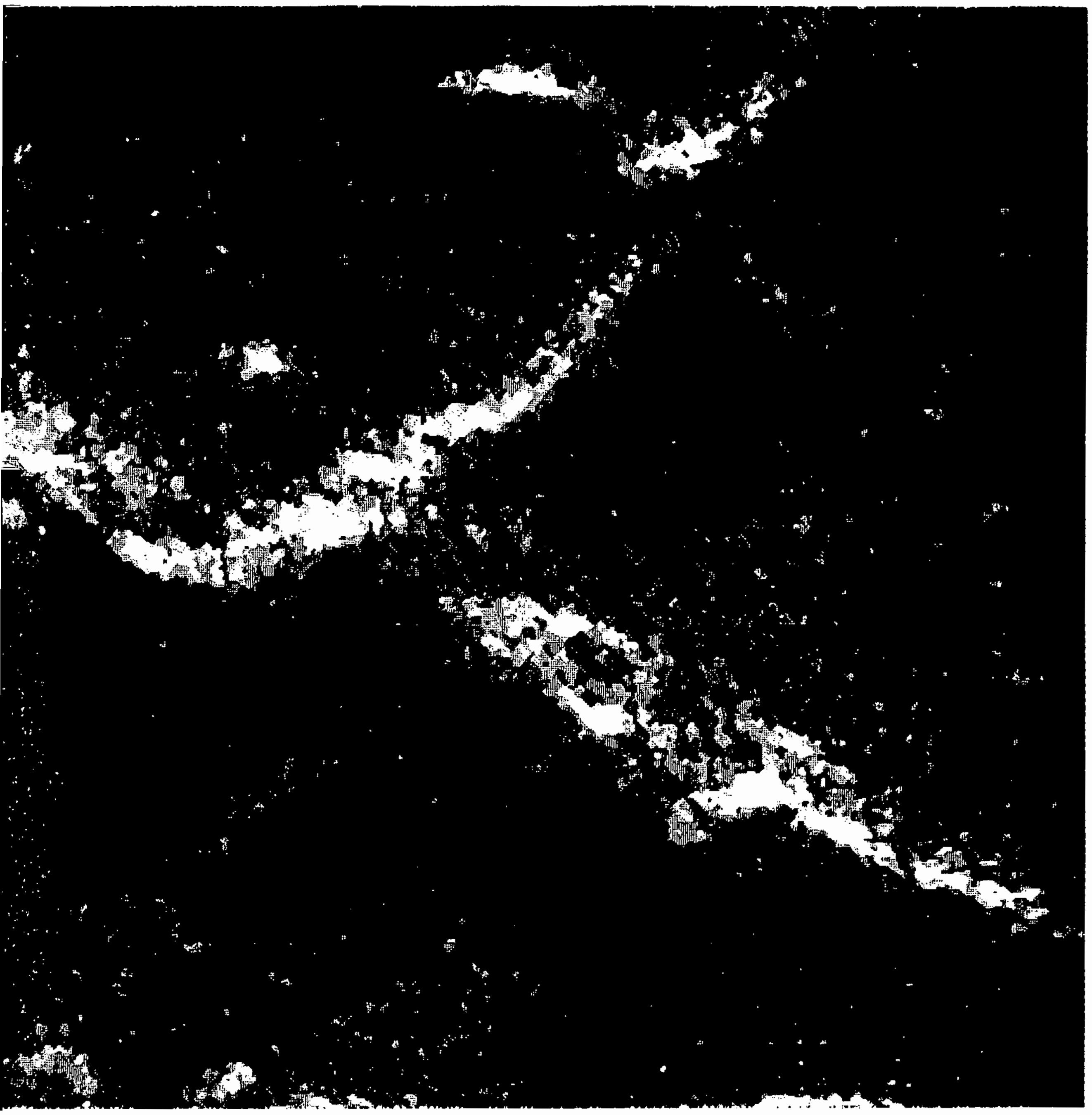


Figure 19 Median Filtered Image B

7.5 Output Image Observations

The radar images produced were compared with topographical maps and a number of initial observations were made.

The first observation that was made was the presence of bright lines running in the azimuth direction at a number of places on both the JPL processed and UCT processed images. These were thought to be strong radar reflectors such as metallic fences or maybe power lines. The fact that they were only visible when running in the azimuth direction, indicates the possibility that the reflections were from regular extended targets lying parallel to the direction of travel of the radar. These targets would need to be regular, since no reflections were received when not parallel to the direction of travel of the radar.

Another observation made in a specific area on one of the images, was the presence of a number of strong radar reflectors, which were regularly spaced and formed a straight line. These could quite possibly be metallic pylons, which would probably be spaced in this way, and would certainly produce strong radar reflections. Figure 20 shows this image, with the reflectors clearly visible.

The observations mentioned here were merely noted from a glance at the output images, and further investigations would certainly be necessary to define them properly.

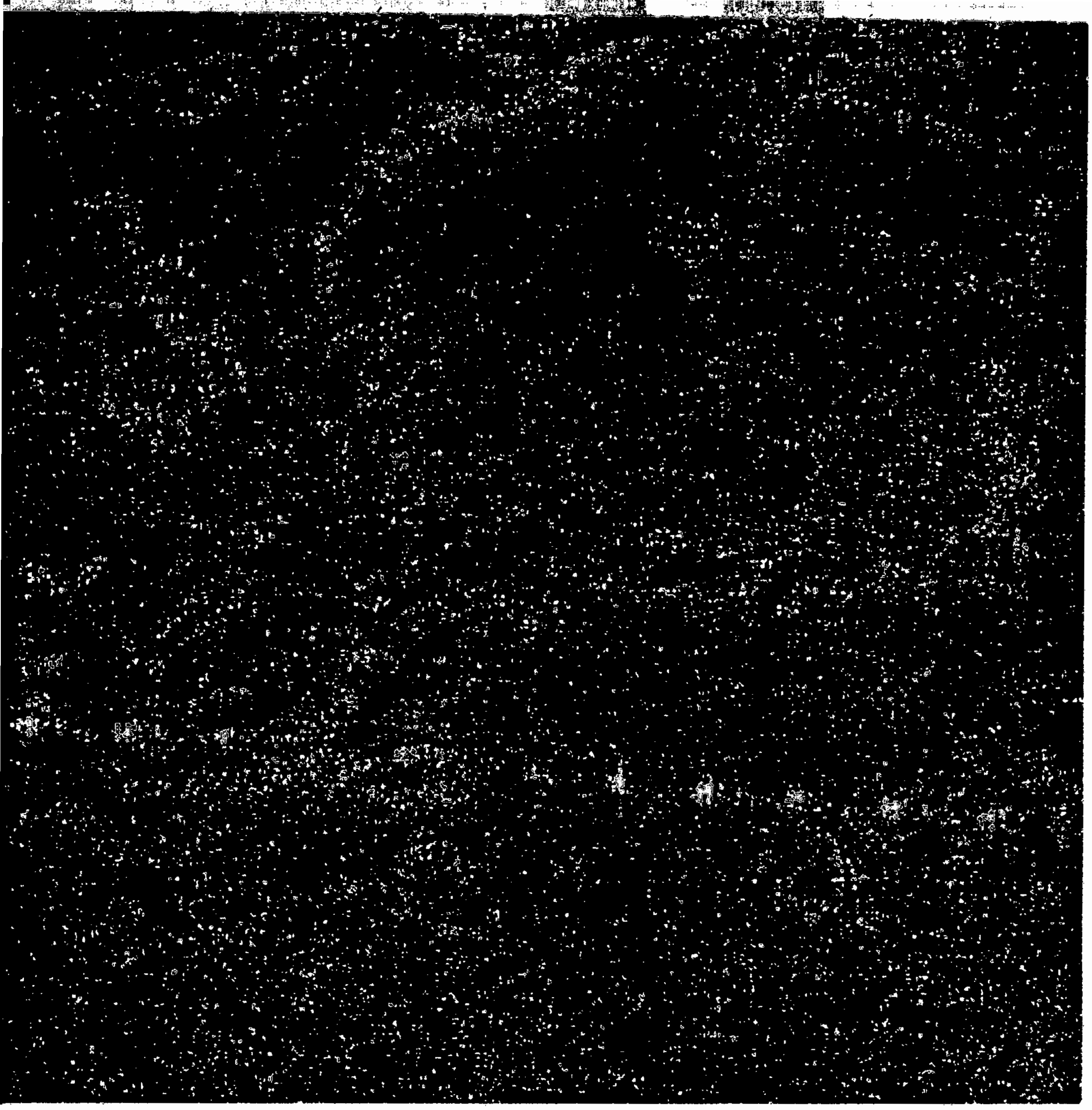


Figure 20 Point Reflectors on Image

8. Program Extensions

A number of improvements and extensions, which are beyond the scope of this thesis, can be made to the basic software presented here. These will be discussed briefly in this section.

8.1 Multi-Look Processing

The purpose of multi-look processing is to reduce the effect of speckle, which is a problem that was defined earlier.

Multi-look processing involves the splitting of the length of the synthetic array into a number of sections, effectively forming a number of sub-apertures. This allows each sub-aperture to be used to produce an image of the same area. The separate images are then combined by incoherently summing them to form the final image.

Each sub-aperture will effectively be pointing in slightly different directions which allows the target to pass through each sub-aperture which will be able to resolve it looking at it from slightly different angles. These slightly different angles are then great enough to produce different speckle effects in each of the multi-look images. The incoherent summing of the images then allows an average of the looks to be produced, reducing the speckle effect.

Multi-look processing could be implemented in the azimuth compression module, by splitting up the synthetic array and forming a number of multi-look images. The processing time overhead would not be increased, as the size of FFT required would be reduced by the number of looks required.

8.2 Dynamic Range of Output

The dynamic range of the output data files produced by each of the modules is limited by its fixed point nature. The reason for using a fixed point format was to economise on disk usage. A compact floating point format may alternatively be devised and used. This would obviously require an investigation into the effects of dynamic range and accuracy on the final image quality.

Another solution could be to apply a Compander algorithm to the data. This would compress the dynamic range before outputting to the mass storage device, and then expand again to the full dynamic range when reading in the data. This can probably be done quite simply using some sort of logarithmic scale.

9. Conclusions and Future Work

The software developed in this thesis performed as it was designed to do, and met with the specifications required.

The images produced compared favourably with those produced by the JPL SAR processor, but may be improved by implementing other processing enhancements. These could include multi-look and the lengthening of the synthetic array. This would also require range curvature and depth of focus compensation.

The speed of processing may be improved by running the software on a different hardware platform, or re-developing the software to run in a parallel processing environment. Parallel approaches to SAR processing have been proposed by Franceschetti and others[19] [20].

This thesis has been a successful initial implementation of a digital SAR processor, and has provided a useful starting point for future SAR processing work in South Africa.

10. References

1 Elachi, Charles **Spaceborne Radar Remote Sensing: Applications and Techniques**. IEEE Press, 1988. Imaging Applications: Pages 12-50.

2 Elachi, Charles **Spaceborne Radar Remote Sensing: Applications and Techniques**. IEEE Press, 1988. Optical Processing: Pages 145-151.

3 Stremler, Ferrel G. **Introduction to Communications Systems.** 2nd Ed. Addison-Wesley, 1982, Pages 406-409: Matched Filtering.

4 Kritzinger, P.J. (**Masters Thesis to be submitted**), Department of Electrical Engineering, University of Cape Town, 1991.

5 Elachi, Charles **Spaceborne Radar Remote Sensing: Applications and Techniques**. IEEE Press, 1988.

6 Kritzinger, P.J. (**Masters Thesis to be submitted**), Department of Electrical Engineering, University of Cape Town, 1991, Section 2.6.2: Digital Demodulation.

7 Kritzinger, P.J. (**Masters Thesis to be submitted**), Department of Electrical Engineering, University of Cape Town, 1991, Section 3.4: Unfocussed Azimuth Processing.

8 Kritzinger, P.J. (**Masters Thesis to be submitted**), Department of Electrical Engineering, University of Cape Town, 1991, Section 3.10: Depth of Focus.

9 Franceschetti, G. and Schirinzi, G. **A SAR Processor Based on Two-Dimensional FFT Codes.** IEEE Trans. AES Vol. 26, No. 2, March 1990, Pages 356-365.

10 Di Cenzo, A. **A New Look at Nonseparable Synthetic Aperture Radar Processing.** IEEE Trans. AES Vol. 24, No. 3, May 1988, Pages 218-223.

11 Kritzinger, P.J. (**Masters Thesis to be submitted**), Department of Electrical Engineering, University of Cape Town, 1991, Section 3.9.5: Range Curvature.

12 Barber, B.C. **Review Article: Theory of Digital imaging from orbital synthetic-aperture radar**. Int. J. Remote Sensing, 1985, Vol. 6, No. 7, pp. 1009-1057

13 Harris, Fredric J. **On the use of Windows for Harmonic Analysis with the Discrete Fourier Transform.** IEEE proceedings, Vol. 66, No. 1, January 1978.

14 Barber, B.C. **Review Article: Theory of Digital imaging from orbital synthetic-aperture radar**. Int. J. Remote Sensing, 1985, Vol. 6, No. 7, pp. 1009-1057

15 Roberts, Richard A. and Mullis, Clifford T. **Digital Signal Processing.** Addison-Wesley, 1987. Power-of-Two FFT algorithms: Pages 148-152.

16 Kritzinger, P.J. **(Masters Thesis to be submitted)**, Department of Electrical Engineering, University of Cape Town, 1991, Appendices B3 and B4.

17 Kritzinger, P.J. **(Masters Thesis to be submitted)**, Department of Electrical Engineering, University of Cape Town, 1991, Section 3.3.3.

18 Wehner D.R. **High Resolution Radar**. Norwood, MA. Artech House, 1987. Speckle: pages 231-235.

19 Franceschetti G., Pascazio V., Schirinzi G., Mazzeo A., Mazzocca, N. **A Distributed Parallel Processor for Precision SAR Imaging.** International Geoscience And Remote Sensing Symposium (IGARSS) 1990, Pages 1303-1307.

20 Franceschetti G., Pascazio V., Schirinzi G., Mazzeo A., Mazzocca, N. **An Efficient SAR Parallel Processor.** IEEE Trans. AES Vol. 27, No. 2, March 1991, Pages 343-352.

11. Bibliography

Barber, B.C. **Review Article: Theory of Digital imaging from orbital synthetic-aperture radar**. Int. J. Remote Sensing, 1985, Vol. 6, No. 7, pp. 1009-1057

Borland, **Turbo C Reference Manual**, Ver 2.00

Brigham E.O. **The Fast Fourier Transform and its Applications**. Englewood Cliffs, New Jersey: Prentice Hall, 1988.

Cimino J.B., Holt B., Richardson A.H. **The Shuttle Imaging Radar B (SIR-B) Experiment Report**. Jet Propulsion Laboratory, California. NASA JPL Pub. 88-2, 1988.

Curlander, John C. **Performance of the SIR-B Digital Image Processing Subsystem.** IEEE Trans, Vol. GE-24, No. 4, July 1986. pp. 649-651

Elachi, Charles **Spaceborne Radar Remote Sensing: Applications and Techniques**. IEEE Press, 1988.

Geckinli, Nezih C. and Yavuz, Davras. **Some Novel Windows and a Concise Tutorial Comparison of Window Families**, IEEE Trans, Vol. ASSP-26, No. 6, December 1978. pp. 501-507.

Harris, Fredric J. **On the use of Windows for Harmonic Analysis with the Discrete Fourier Transform**. IEEE proceedings, Vol. 66, No. 1, January 1978.

Hovanessian, S.A. **Radar System Design and Analysis**. Artech House Inc, 1984.

Kritzinger, P.J. **(Masters Thesis to be submitted)**, Department of Electrical Engineering, University of Cape Town, 1991

Munson, David C. and Visentin, Robert L. **A Signal Processing View of Strip-Mapping Synthetic Aperture Radar**. IEEE Trans. Acoustics, Speech and Signal Processing, VOL 37 No 12, December 1989

Roberts, Richard A. and Mullis, Clifford T. **Digital Signal Processing**. Addison-Wesley, 1987.

Sommerville, Ian. **Software Engineering**. 3rd Edition, Addison-Wesley 1989

Stremler, Ferrel G. **Introduction to Communications Systems**. 2nd Ed. Addison-Wesley, 1982.

Wehner D.R. **High Resolution Radar**. Norwood, MA. Artech House, 1987.

APPENDIX A

Main SAR Program Files

SATPARAM.C - Generate Processing Parameters
RANGECOM.C - Perform Range Compression
CORNER.C - Perform Corner Turn
AZIMCOM.C - Perform Azimuth Compression

```
/***************************************************************************/
/* MODULE:       SATPARAM.C                                                */
/*                                                                         */
/* Functions:    main() - Parameter Generation procedure                   */
/*                                                                         */
/* Author:       Simon Welsh                                               */
/* Date:         01/09/91                                                  */
/***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <alloc.h>
#include "param.h"
#include "matrix.h"

#define pi 3.141592654
#define lightspeed 3e8

/***************************************************************************/
/* main()                                                                  */
/* Program to generate paramaters for the other modules                    */
/*                                                                         */
/* Input Parameters: argc - Number of Command Line parameters              */
/*                   *argv[1] - Name of Input Parameter File               */
/*                                                                         */
/* Output Files: AZIMREF.DAT - Azimuth Reference Function                  */
/*               SAT.PAR - Other Generated Parameters                      */
/***************************************************************************/

main(int argc, char *argv[])
{
FILE    *azimrefout;
FILE    *azimrangeout;
FILE    *inparamfile;
FILE    *satparamfile;

double  rangescale;             /* Scaling Factor for rangecom */

char    found;
char    rawname[30];
char    rawtype[30];
int     samples;
double  prf,lambda;
float   rangebinsize;   /* Size of Rangebin to determine range walk in bins */
float   sampfreq;       /* Sampling frequency to compute rangebinsize */

double  satpos[3];      /* x,y,z Satellite Positiion */
double  satvel[3];      /* x,y,z Satellite Velocity */
double  slant,earthradius,look;
double  b,c;

double  dsath;
double  P[3][3];
double  angle;
double  intercept[3];
double  targetpos[3];   /* x,y,z Target Position */
double  targetsph[3];    /* Spherical Target Position */
double  dtargetsph[3];   /* Spherical Target Velocity */
double  rotvect[3];      /* Earth Rotation Vector */
double  targetvel[3];    /* x,y,z Target Velocity */
double  swathvel[3];     /* x,y,z Swath Velocity */
double  swathvelmag;     /* Swath Velocity Magnitude */
double  azimuthpixel;    /* Azimuth Pixel Size */
double  rotaxis[3];
double  Q[3][3];
double  satsph[3];       /* Spherical Satellite Position */
double  dsatsph[3];      /* Spherical Satellite Velocity */

double  ti;
int     i;
double  A[3];
double  T[3];
double  S[3];
double  *R,*real,*imag;
double  rvect[3], rmag;
float   rangewalk;

/******************************************************************************/
/* Check input parameters:                                                    */
/******************************************************************************/

if (argc!=2)
{ printf("SATPARAM paramfile\n");
  return(1); }

/******************************************************************************/
/* Open the input parameter file for reading:                                 */
/******************************************************************************/

inparamfile = fopen(argv[1],"r");
if (inparamfile==NULL)
{ printf("%s Input Parameter File Not Found - Aborting\n",argv[1]);
  return(1); }

/******************************************************************************/
/* Set up input Constants, read from input parameter file:                    */
/******************************************************************************/

satpos[0] = getfloatparam("xpos",inparamfile,&found);
if (!found) return(1);
satpos[1] = getfloatparam("ypos",inparamfile,&found);
if (!found) return(1);
satpos[2] = getfloatparam("zpos",inparamfile,&found);
if (!found) return(1);
satvel[0] = getfloatparam("xvel",inparamfile,&found);
if (!found) return(1);
satvel[1] = getfloatparam("yvel",inparamfile,&found);
if (!found) return(1);
satvel[2] = getfloatparam("zvel",inparamfile,&found);
if (!found) return(1);
prf = getfloatparam("prf",inparamfile,&found);
if (!found) return(1);
lambda = getfloatparam("lambda",inparamfile,&found);
```

```
if (!found) return(1);
samples = getfloatparam("samples",inparamfile,&found);
if (!found) return(1);
look = getfloatparam("look",inparamfile,&found);
if (!found) return(1);
earthradius = getfloatparam("earthradius",inparamfile,&found);
if (!found) return(1);
sampfreq = getfloatparam("sampfreq",inparamfile,&found);
if (!found) return(1);
getstringparam("rawname",rawname,inparamfile,&found);
if (!found) return(1);
getstringparam("rawtype",rawtype,inparamfile,&found);
if (!found) return(1);

/*****************************************************************************/
/* Allocate memory from the heap:                                            */
/*****************************************************************************/

R    = malloc( sizeof(double) * samples );
real = malloc( sizeof(double) * samples );
imag = malloc( sizeof(double) * samples );

/*****************************************************************************/
/* If slant range parameter not specified, then compute the slant range      */
/* based on the radar height and look angle:                                 */
/*****************************************************************************/

/* Convert look angle from degrees to radians: */

look = look * pi/180;

slant = getfloatparam("slant",inparamfile,&found);
if (!found)
{ b = -2 * magnitude(satpos) * cos(look);
  c = pow(magnitude(satpos), 2) - pow(earthradius, 2);
  slant = (-b - sqrt(pow(b,2) - 4 * c)) / 2;
  printf("Slant Range to Swath Center Computed As: %g\n",slant); }

/* Compute rangebinsize from the sampling frequency: */

rangebinsize = (lightspeed / ( sampfreq * 2 )) / sin(look) ;

/*****************************************************************************/
/* Find Satellite and Target positions and velocities in spherical Co-ords   */
/*****************************************************************************/

dsath = mult( satvel, satpos ) / magnitude(satpos);

constdiv( satpos, P[0], magnitude(satpos) );
constdiv( satvel, P[2], magnitude(satvel) );
crossprod( P[2], P[0], P[1] );
printf("\nP Matrix: \n");
show( P[0] );
show( P[1] );
show( P[2] );
```

```
angle = acos((pow(slant,2) - pow(magnitude(satpos),2) - pow(earthradius,2)) / (-
2 * earthradius * magnitude(satpos)));

intercept[0] = earthradius * cos(angle);
intercept[1] = earthradius * sin(angle);
intercept[2] = 0;
matmult( P, intercept, targetpos );

targetsph[0] = magnitude(targetpos);
targetsph[1] = atan(targetpos[1] / targetpos[0]);
targetsph[2] = acos(targetpos[2] / magnitude(targetpos));
printf("\ntargetsph:\n");
show(targetsph);

dtargetsph[0] = 0;
dtargetsph[1] = (2*pi)/24/60/60;
dtargetsph[2] = 0;
printf("\ndtargetsph:\n");
show(dtargetsph);

crossprod( satpos, satvel, rotaxis );
constdiv( rotaxis, rotaxis, magnitude(rotaxis) );

constdiv( satpos, Q[0], magnitude(satpos) );
constdiv( rotaxis, Q[2], 1);
crossprod( Q[2], Q[0], Q[1] );
printf("\nQ Matrix:\n");
show( Q[0] );
show( Q[1] );
show( Q[2] );

satsph[0] = magnitude(satpos);
satsph[1] = 0;
satsph[2] = pi/2;
printf("\nsatsph:\n");
show(satsph);

dsatsph[0] = dsath;
dsatsph[1] = magnitude(satvel) / magnitude(satpos);
dsatsph[2] = 0;
printf("\ndsatsph:\n");
show(dsatsph);

/* Compute the Swath Velocity: */
/* Target Velocity = dtargetsph crossprod Target Position */

rotvect[0] = 0;
rotvect[1] = 0;
rotvect[2] = (2*pi)/24/60/60;
crossprod(rotvect, targetpos, targetvel);
printf("\ndsatsph:\n");
show(targetvel);

vectsub( satvel, targetvel, swathvel);
swathvelmag = magnitude(swathvel);      /* Swath Velocity Magnitude */
azimuthpixel = swathvelmag / prf;       /* Compute Pixel Size */
```

```
/*****************************************************************************/
/* Spherical co-ordinates of satellite and target have been set up.         */
/* Now generate an array of distances between satellite and target:         */
/*****************************************************************************/

for(i=0; i<samples; i++)
{
  ti = (-(samples/2)*(1/prf)+(i+1)*(1/prf));
  S[0] = (satsph[0]+dsatsph[0]*ti) * sin(satsph[2]) *
cos(satsph[1]+dsatsph[1]*ti);
  S[1] = (satsph[0]+dsatsph[0]*ti) * sin(satsph[2]) *
sin(satsph[1]+dsatsph[1]*ti);
  S[2] = (satsph[0]+dsatsph[0]*ti) * cos(satsph[2]);
  matmult( Q, S, A );

  T[0] = targetsph[0] * sin(targetsph[2]) * cos(targetsph[1] + dtargetsph[1] *
ti);
  T[1] = targetsph[0] * sin(targetsph[2]) * sin(targetsph[1] + dtargetsph[1] *
ti);
  T[2] = targetsph[0] * cos(targetsph[2]);

  vectsub( A, T, rvect );
  R[i] = magnitude(rvect);
}

/*****************************************************************************/
/* The Azimuth Reference Function is generated from the array of ranges:    */
/* It is written out to the azimref.dat file to be read in by the           */
/* azimuth compression module.                                              */
/*****************************************************************************/

azimrefout=fopen("azimref.dat","w");
azimrangeout=fopen("azmrange.dat","w");
rmag = R[(samples-1)/2];
for(i=0; i<samples; i++)
{ R[i] = 2 * (R[i] - rmag);
  real[i] = cos(-2*pi*R[i]/lambda);
  imag[i] = sin(-2*pi*R[i]/lambda);
  printf("%d %g %g %g\n",i,R[i],real[i],imag[i]);
  fprintf(azimrefout, "%g %g\n", real[i],imag[i]);
  fprintf(azimrangeout,"%g\n",R[i]);
}
close(azimrefout);

rangewalk = abs(R[samples-1] - R[0]) / (float) samples;
printf("Rangewalk: %g m\n", rangewalk);

/*****************************************************************************/
/* Open SAT.PAR parameter file for writing and write out new parameters: */
/*****************************************************************************/

satparamfile=fopen("SAT.PAR","w");
putfloatparam("xpos",satpos[0],satparamfile);
putfloatparam("ypos",satpos[1],satparamfile);
putfloatparam("zpos",satpos[2],satparamfile);
putfloatparam("xvel",satvel[0],satparamfile);
putfloatparam("yvel",satvel[1],satparamfile);
putfloatparam("zvel",satvel[2],satparamfile);
putfloatparam("prf",prf,satparamfile);
putfloatparam("look",look,satparamfile);
putfloatparam("rangewalk",rangewalk,satparamfile);
putfloatparam("rangepixel",rangebinsize,satparamfile);
putfloatparam("azimuthpixel",azimuthpixel,satparamfile);
putfloatparam("slant",slant,satparamfile);
close(satparamfile);
close(inparamfile);
return(0);

}
```

```
/***************************************************************************/
/* MODULE:      RANGECOM.C                                                 */
/*                                                                         */
/* Functions:   compress_range_line() - Compress Single Range Line         */
/*              main() - Compress Raw file of range lines                  */
/*                                                                         */
/* Author:      Simon Welsh                                                */
/* Date:        01/09/91                                                   */
/***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "sar.h"
#include "extract.h"
#include "param.h"
#include "window.h"

/***************************************************************************/
/* compress_range_line()                                                   */
/* Compress a single range line                                            */
/*                                                                         */
/* Input Parameters: xreal - Pointer to real output store                  */
/*                   ximag - Pointer to imag output store                  */
/*                   sinlook - Pointer to Sin lookup table for FFT         */
/*                   coslook - Pointer to Cos lookup table for FFT         */
/*                   rrefreal - Pointer to Range Reference Function (real) */
/*                   rrefimag - Pointer to Range Reference Function (imag) */
/*                   azimline - Rangeline to be compressed                 */
/*                   rangefftsize - Size of FFT to use                     */
/*                   totbins - Total rangebins extracted from raw file     */
/*                   centreshift - Shift before IFFT                       */
/*                   satdat - Pointer to raw input data file               */
/*                                                                         */
/* Steps Performed:                                                        */
/*                                                                         */
/* Read in a real range line from the raw data file.                       */
/* Perform FFT on the input range line.                                    */
/* Multiply FFT of the range line by FFT of the reference function.        */
/* Zero the upper half of the samples.                                     */
/* Shift the samples, effectively demodulating to baseband.                */
/* Perform Inverse FFT on the result.                                      */
/*                                                                         */
/***************************************************************************/

compress_range_line(float xreal[],float ximag[],float rrefreal[],float
rrefimag[],float sinlook[],float coslook[], int azimline, int rangefftsize, int
*totbins, float centreshift, FILE *satdat)
{
int     i;

  for(i=0;i<rangefftsize;xreal[i]=0,ximag[i]=0,i++);
  extract(satdat, azimline, xreal, totbins);
  fft(xreal,ximag,sinlook,coslook,rangefftsize,1);
  multiply(xreal,ximag,rrefreal,rrefimag,rangefftsize);
```

```
/* Zero the upper half of the samples: */

  for(i=rangefftsize/2; i<rangefftsize; xreal[i]=0, ximag[i]=0, i++);

/* TEST CODE: Determine the position of the maximum in frequency domain: */
/*
  maxval = 0;
  for(i=0; i<rangefftsize/2; i++ )
  { if (!(i%8)) fftmag=0;
    fftmag += (xreal[i] * xreal[i] + ximag[i] * ximag[i]);
    if ( fftmag > maxval)
    { maxval = fftmag;
      maxpos = i-4; } }
  printf("MaxPos: %d ",maxpos);
  avemaxpos += maxpos;
  printf("AvePos: %d ",avemaxpos / (azimline - startazim + 1));
*/

/* Shift the samples so as to centre them at the fft origin,    */
/* Thus demodulating to baseband:                               */

  for(i=0; i<centreshift; i++ )
  { xreal[rangefftsize-centreshift+i] = xreal[i];
    xreal[i] = xreal[centreshift+i];
    ximag[rangefftsize-centreshift+i] = ximag[i];
    ximag[i] = ximag[centreshift+i]; }

/* Perform the inverse FFT: */

  fft(xreal,ximag,sinlook,coslook,rangefftsize,-1);
}

/**************************************************************************/
/* main()                                                                 */
/* Start of main program to compress range lines                          */
/*                                                                        */
/* Steps Performed:                                                       */
/*                                                                        */
/* Extract Input Parameters.                                              */
/* Create Range Reference Function.                                       */
/* Perform FFT on range reference Function and Conjugate it.              */
/*                                                                        */
/* Perform Sample Range Compressions to determine output scaling factor   */
/* Perform actual range compressions with the computed output scaling     */
/* factor, and shift the output samples to compensate for range walk.     */
/*                                                                        */
/* Output data to the status file                                         */
/**************************************************************************/

main(int argc, char *argv[])
{

/* File Pointer Declarations: */

FILE    *fpout;
FILE    *satdat;
FILE    *inparamfile;
FILE    *satparamfile;
FILE    *status;
FILE    *test;

/* Input parameter variable declarations: */

char    found;          /* Indicate if parameter was found */

int     rangefftsize;   /* Size of the FFT, closest to rangesamples */
int     startazim;      /* Starting Azimuth Pulse */
int     azimtot;        /* Number of azimuth pulses to process */
int     rangebins;      /* Number of range compressed output rangebins */
float   rangescale;     /* rangescale for output */
char    rawname[50];    /* Name of the raw data file */
char    rngname[50];    /* Output Range Compressed Filename */
char    statusname[50]; /* Name of the status file */

float   pulselength;    /* Reference Pulse Length */
float   sampfreq;       /* Sampling Frequency */
float   centreshift;    /* Shift to centre samples before inverse FFT */
float   ifreq;          /* Intermediate Frequency */
float   maxfreq;
float   sweep;          /* Chirp Sweep Range */

float   rangewalk;      /* Rangewalk in metres per pulse */
float   rangebinsize;   /* Size of Rangebin to determine range walk in bins */
int     windowtype;     /* Type of Window to apply */

int     totbins;        /* Number of bins in raw rangeline, set by extract */

/* Declare Dynamic Heap Memory pointers: */

float   *rrefreal;
float   *rrefimag;
float   *xreal;
float   *ximag;
float   *sinlook;
float   *coslook;

/* Variables for determining Scaling Factor: */

double  peakmag;
double  fftmag;
double  avepeakmag;

/* Other Program variable declarations: */

int     N,i,dup;
float   realout, imagout;
float   maxval;
int     maxpos;
long    avemaxpos;

int     walkshift;      /* Current bin shift for range walk compensation */
int     maxwalkshift;   /* Maximum Range Walk bin shift */
int     azimline;
long    starttime, endtime, outstart, outend;
long    outerror, zeroerror, maxouterror, maxzeroerror, outpos, zeropos;
float   rangelinemean, minmag, maxmag, magout;

unsigned char   buffer;
float           value;

/* Status output variables */

float   totmean;

/* Range Reference Function variables: */

float   gain,tm;
float   ipos;
int     pos;                    /* Position in Array */

/**************************************************************************/
/* Check input parameters:                                                */
/**************************************************************************/

if (argc!=2)
{ printf("RANGECOM paramfile\n");
  return(1); }

/***********************************************************************/
/* Open input parameter files INPUT.PAR and SAT.PAR:                   */
/***********************************************************************/

inparamfile = fopen(argv[1],"r");
if (inparamfile==NULL)
{ printf("%s Input Parameter File Not Found - Aborting\n",argv[1]);
  return(1); }

satparamfile = fopen("SAT.PAR","r");
if (satparamfile==NULL)
{ printf("SAT.PAR Input Parameter File Not Found - Aborting\n");
  return(1); }

/***********************************************************************/
/* Read in parameter values from the parameter files:                  */
/***********************************************************************/

getstringparam("rawname",rawname,inparamfile,&found);
if (!found) return(1);
getstringparam("rngname",rngname,inparamfile,&found);
if (!found) return(1);
getstringparam("statusname",statusname,inparamfile,&found);
if (!found) return(1);

startazim = getfloatparam("startpulse",inparamfile,&found);
if (!found) return(1);
azimtot = getfloatparam("pulses",inparamfile,&found);
if (!found) return(1);
rangebins = getfloatparam("rangebins",inparamfile,&found);   /* Output 2494 */
if (!found) return(1);
rangefftsize = getfloatparam("rangefftsize",inparamfile,&found);
if (!found) return(1);
```

```
windowtype = getfloatparam("window",inparamfile,&found);
if (!found) return(1);

pulselength = getfloatparam("pulselength",inparamfile,&found);
if (!found) return(1);
sampfreq = getfloatparam("sampfreq",inparamfile,&found);
if (!found) return(1);
ifreq = getfloatparam("ifreq",inparamfile,&found);
if (!found) return(1);
sweep = getfloatparam("sweep",inparamfile,&found);
if (!found) return(1);

rangewalk = getfloatparam("rangewalk",satparamfile,&found);
if (!found) return(1);
rangebinsize = getfloatparam("rangepixel",satparamfile,&found);
if (!found) return(1);

fclose(inparamfile);
fclose(satparamfile);

/**************************************************************************/
/* Open the status file for writing. Existing data is cleared.            */
/**************************************************************************/

status  = fopen(statusname,"w");
fprintf(status,"Start of the Range Compression Module.\n");
fprintf(status,"======================================\n");
fprintf(status,"\n");
fflush(status);

/**************************************************************************/
/* Allocate memory and check for out of memory error:                     */
/**************************************************************************/

rrefreal = malloc( rangefftsize * sizeof(float) );
rrefimag = malloc( rangefftsize * sizeof(float) );
xreal    = malloc( rangefftsize * sizeof(float) );
ximag    = malloc( rangefftsize * sizeof(float) );
sinlook  = malloc( rangefftsize * sizeof(float) );
coslook  = malloc( rangefftsize * sizeof(float) );

if (coslook==NULL)
{ fprintf(status,"ERROR: Range Compression Memory Allocation - Aborting\n");
  fclose(status);
  return(1); }

/*****************************************************************************/
/* Open the satdat raw data file and range compressed output file:           */
/*****************************************************************************/

satdat = fopen(rawname,"rb");
if (satdat==NULL)
{ fprintf(status,"ERROR: Opening Raw data file: %s - Aborting\n",rawname);
  fclose(status);
  return(1); }

fpout = fopen(rngname,"wb");
```

```
/*******************************************************************************/
/* Send Useful Information to ASCII data file:                                 */
/*******************************************************************************/

fprintf(status,"Output File %s\n",rngname);
fprintf(status,"Range Walk Compensation: %g metres per pulse\n", rangewalk);
fprintf(status,"Starting azimuth pulse: %d\n", startazim);
fprintf(status,"Total azimuth pulses to process: %d\n", azimtot);
fprintf(status,"Range bins output per azimuth line: %d\n", rangebins);
fprintf(status,"Window type: %d\n",windowtype);

fprintf(status,"\n");
fflush(status);

/****************************************************************************/
/* Set up the range reference function:                                     */
/* It is left justified.                                                    */
/****************************************************************************/

gain = sweep / pulselength;      /* Sweep Gain */
tm   = 1 / sampfreq;             /* Time between samples */
maxfreq = ifreq + sweep / 2;     /* Compute maxfreq of reference */
centreshift = ifreq / sampfreq * rangefftsize;

if ((int) (pulselength * sampfreq) >= rangefftsize)
{ fprintf(status,"ERROR: Rangefftsize must be greater than range reference
pulse\n");
  fclose(status);
  return(1); }

printf("Generating Range Reference Function.\n");
for(ipos=0;ipos<rangefftsize;rrefreal[ipos]=0,rrefimag[ipos]=0,ipos++);

test = fopen("rangeref.dat","w");
for(ipos=0; ipos <= (int) (pulselength * sampfreq) ; ipos++ )
{ rrefreal[ipos] = cos(2*pi*(maxfreq*tm*ipos-0.5*gain*tm*ipos*tm*ipos));
  fprintf(test,"%g\n",rrefreal[ipos]);
}
fclose(test);

/*****************************************************************************/
/* Perform FFT on the reference array.                                       */
/* Conjugate the FFT of the reference function.                              */
/*****************************************************************************/

loadsincos(sinlook,coslook,rangefftsize);
fft(rrefreal,rrefimag,sinlook,coslook,rangefftsize,1);
for(i=0;i<rangefftsize;rrefimag[i]=-rrefimag[i],i++);
window(rrefreal,rrefimag,rangefftsize,windowtype);

/*****************************************************************************/
/* Perform Sample Range Compressions to determine output scaling constant:   */
/* Every 32nd azimuth pulse is range compressed.                             */
/*****************************************************************************/

avepeakmag = 0;
```

```
for(azimline=startazim; azimline < startazim+azimtot; azimline+=32)
{
compress_range_line(xreal,ximag,rrefreal,rrefimag,sinlook,coslook,azimline,range
fftsize,&totbins,centreshift,satdat);

/* Determine the position of the peak: */

  peakmag = 0;
  for(i=0; i<rangebins; i++ )
  { fftmag = (xreal[i] * xreal[i] + ximag[i] * ximag[i]);
    if (fftmag > peakmag)
      peakmag = fftmag; }

  avepeakmag += peakmag;
  printf("Peak Magnitude: %lg\n",peakmag);
}

avepeakmag/=(azimtot/32);
rangescale = 150.0 / sqrt(avepeakmag/2);  /* Origionally 134.0 */
fprintf(status,"Output Scaling Factor: %g\n",rangescale);
fflush(status);

/*************************************************************************/
/* The reference correlation function has now been set up.               */
/* Start of loop to process multiple range lines:                        */
/*************************************************************************/

maxouterror = 0;
maxzeroerror = 0;
maxmag=0;
minmag=10000;
avemaxpos=0;
totmean=0;

time(&starttime);
for(azimline=startazim; azimline < startazim + azimtot; azimline++)
{ time(&outstart);

compress_range_line(xreal,ximag,rrefreal,rrefimag,sinlook,coslook,azimline,range
fftsize,&totbins,centreshift,satdat);

/*************************************************************************/
/* Output the compressed range line to the output binary file:           */
/* Unsigned bytes are used (0 to 255).                                   */
/*************************************************************************/

  outerror = 1;
  zeroerror = 1;
  rangelinemean = 0;

/* Compute the current range walk shift in rangebins: */

  maxwalkshift = azimtot / (rangebinsize / rangewalk);
  walkshift = (azimline - startazim) / (rangebinsize / rangewalk);
```

```
/***********************************************************************/
/* Insert leading zeros for range walk compensation:                   */
/***********************************************************************/

  for(i = 0; i < (maxwalkshift-walkshift); )
  { buffer = (unsigned char) 128;
    fwrite(&buffer, 1, 1, fpout);
    fwrite(&buffer, 1, 1, fpout);
    i++; }

/***********************************************************************/
/* Output the data:                                                    */
/***********************************************************************/

  for(i = rangebins-1-maxwalkshift; i>=0; )
  { realout = (xreal[i] * rangescale);
    imagout = (ximag[i] * rangescale);
    magout  = sqrt(realout*realout + imagout*imagout);
    rangelinemean += magout;
    if (magout>maxmag) maxmag=magout;
    if (magout<minmag) minmag=magout;

    buffer = (unsigned char) (realout+128);
    if (realout >  127) buffer = (unsigned char)  255;
    if (realout < -127) buffer = (unsigned char)  0;
    fwrite(&buffer, 1, 1, fpout);

    buffer = (unsigned char) (imagout+128);
    if (imagout >  127) buffer = (unsigned char)  255;
    if (imagout < -127) buffer = (unsigned char)  0;
    fwrite(&buffer, 1, 1, fpout);

    i--;
    if ((imagout > 127) || (imagout < -127) || (realout > 127) || (realout < -
127))
      outerror++;
    if (!((signed char)imagout) && !((signed char)realout))
      zeroerror++;
  }

/***********************************************************************/
/* Pad out with zeros for range walk compensation:                     */
/***********************************************************************/

  for(i=0; i<walkshift; )
  { buffer = (unsigned char) 128;
    fwrite(&buffer, 1, 1, fpout);
    fwrite(&buffer, 1, 1, fpout);
    i++; }

/***********************************************************************/
/* Display useful data on the screen:                                  */
/***********************************************************************/

  printf("\n");
  printf("Current Range Walk Correction: %d\n", walkshift);
  printf("Out of Range errors: %ld    Percentage: %ld\n", outerror,
outerror*100/rangebins );
  printf("        Zero errors: %ld    Percentage: %ld\n", zeroerror,
zeroerror*100/rangebins );
  printf(" Ave Out Mag: %g ", (rangelinemean / rangebins) );

  printf(" Max Out Mag: %g ", maxmag );
  printf(" Min Out Mag: %g\n", minmag );
  if (outerror > maxouterror) {maxouterror = outerror; outpos=azimline; }
  if (zeroerror > maxzeroerror) {maxzeroerror = zeroerror; zeropos=azimline; }
  time(&outend);
  printf("Time Elapsed: %d sec  ", outend-outstart);
  printf("Time Remaining: %d min\n",(azimtot * (outend-starttime)/(azimline-
startazim+1))/60 );
  totmean += (rangelinemean/rangebins);
}
time(&endtime);

/*****************************************************************************/
/* Close all files and terminate the program:                                */
/*****************************************************************************/

fprintf(status,"Maximum Walk Shift in Range Bins: %d\n", maxwalkshift);
fprintf(status,"Maximum out of range errors: %ld at %ld \n", maxouterror,
outpos);
fprintf(status,"Maximum zero errors: %ld at %ld \n", maxzeroerror, zeropos);
fprintf(status,"Mean Output Magnitude: %g\n", (totmean / azimtot) );
fprintf(status,"Average Max Position before IFFT: %d\n", (avemaxpos / azimtot)
);
fprintf(status,"Processing Time: %d min\n", (endtime - starttime)/60 );
fprintf(status,"Range Compression Module Complete\n\n");

fclose(status);
fclose(fpout);
fclose(satdat);
return(0);

}
```

```
/**************************************************************************/
/* MODULE:       CORNER.C                                                 */
/*                                                                        */
/* Functions:    main() - Apply corner turn                               */
/*                                                                        */
/* Author:       Simon Welsh                                              */
/* Date:         01/09/91                                                 */
/**************************************************************************/

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <alloc.h>
#include <math.h>
#include "param.h"

/**************************************************************************/
/* main()                                                                 */
/* Program to Apply corner turn to input file and send output to output   */
/* file. Read input file randomly and write output sequentially.          */
/*                                                                        */
/* Input Parameters: argc - Number of Command Line parameters             */
/*                   *argv[1] - Name of Input Parameter File              */
/**************************************************************************/

main(int argc, char *argv[])
{
FILE    *status;
FILE    *fpin;
FILE    *fpout;
FILE    *inparamfile;

char    found;
long    azimtot;
long    rangebins;
char    rngname[50];    /* Range Compressed Input filename */
char    corname[50];    /* Corner Turned Output filename */
char    statusname[50]; /* Status filename */

long    starttime, endtime, blockstart, blockend;

unsigned char huge *store;
unsigned char huge *storepos;
unsigned char huge *temp;
unsigned char far *inblock;

int     row, charsize;
long    N, size, m, azim, block, i, i2, blocksize, curblocksize, test;
char    *termn;

charsize = sizeof(char);

/****************************************************************************/
/* Check input parameters:                                                  */
/****************************************************************************/

if (argc!=2)
```

```
{ printf("CORNER paramfile\n");
  return(1); }

/*************************************************************************/
/* Open input parameter file and read size parameters:                  */
/*************************************************************************/

inparamfile = fopen(argv[1],"r");
if (inparamfile==NULL)
{ printf("%s Input Parameter File Not Found - Aborting\n",argv[1]);
  return(1); }

getstringparam("rngname",rngname,inparamfile,&found);
if (!found) return(1);
getstringparam("corname",corname,inparamfile,&found);
if (!found) return(1);
getstringparam("statusname",statusname,inparamfile,&found);
if (!found) return(1);

azimtot = getfloatparam("pulses",inparamfile,&found);
if (!found) return(1);
rangebins = getfloatparam("rangebins",inparamfile,&found);
if (!found) return(1);

/*************************************************************************/
/* Open status file and write relevant details:                          */
/*************************************************************************/

status=fopen(statusname,"a");
fprintf(status,"Start of the Corner Turn Module.\n");
fprintf(status,"================================\n\n");

fprintf(status,"Input File: %s\n",rngname);
fprintf(status,"Output File: %s\n",corname);
fprintf(status,"Corner Turn Dimensions: %ld by %ld (2 byte
complex)\n",azimtot,rangebins);

/*************************************************************************/
/* Allocate Maximum Dynamic Heap Memory:                                 */
/*************************************************************************/

printf("Available Far Heap Memory: %ld\n", farcoreleft());
blocksize = ((farcoreleft() - 5000) / (2 * (azimtot + 1)));
if (blocksize > rangebins) blocksize = rangebins;
printf("Blocksize chosen: %ld\n", blocksize);
inblock = farmalloc( (unsigned long) blocksize * 2 );
store = farmalloc( (unsigned long) azimtot * blocksize * 2 );
if (store == NULL)
{ fprintf(status,"ERROR - Cannot Allocate Enough Memory for corner turn\n");
  fclose(status);
  return(1); }
printf("Available Far Heap Memory: %ld\n", farcoreleft());

fprintf(status,"Available Memory Allocated: %ld bytes\n", (azimtot+1) *
blocksize * 2 );
```

```
/*****************************************************************************/
/* Open the input and output files and test for errors:                      */
/*****************************************************************************/

fpin  = fopen(rngname,"rb");
fpout = fopen(corname,"wb");

if (fpin==NULL)
{ fprintf(status,"ERROR - %s corner turn input file not found -
Aborting\n",rngname);
  fclose(status);
  return(1); }

rewind(fpin);

/*****************************************************************************/
/* Start of Main Block Read/Transpose/Write Loop:                            */
/*****************************************************************************/

time(&starttime);
time(&blockstart);
for(block=0; block < (int) (0.999 + (float)rangebins/(float)blocksize) ; )
{ printf("Current Block Number: %ld\n", block);

/* Current Block size may be smaller than the full block size on last block: */

  if (blocksize*(block+1) > rangebins)
    curblocksize = rangebins - block*blocksize;
  else
    curblocksize = blocksize;

/* Read in and Transpose the current block in memory: */

  for(azim=0; azim<azimtot; )
  { fseek(fpin, 2*((rangebins-block*blocksize-curblocksize) + (azim*rangebins))
,SEEK_SET);
    fread(inblock, 2*curblocksize, 1, fpin);

    storepos = store + azim*2;
    for(i=0; i<2*curblocksize; )
    { *storepos = (unsigned char) inblock[ 2*curblocksize - i - 2 ];    /* Real
Componant */
      storepos++;
      *storepos = (unsigned char) inblock[ 2*curblocksize - i - 1 ];    /* Imag
Componant */
      storepos--;
      storepos+=(azimtot*2);
      i+=2; }
    azim++;
  }

/* Debug Code: */

/*
  storepos = store;
  for(i=0; i<azimtot*blocksize*2; )
  { printf("%2X  ", *storepos);
```

```
    if (!((i+1)%16)) printf("\n");
    i++;
    storepos++; }
*/

/* Write out the transposed block to the output file: */

   for(i=0; i<azimtot; i++)
   { fwrite(store + i * curblocksize * 2, 1, curblocksize * 2, fpout); }

/* Increment the block number and display current status: */

  block++;
  blockend = blockstart;
  time(&blockstart);
  printf("Block Time Elapsed: %ld seconds\n",(blockstart-blockend));
  printf("Time Remaining: %ld minutes\n", (blockstart-
blockend)*(rangebins/blocksize-block)/60 );
  printf("\n");
}

/* Write terminating data to the summary file: */

time(&endtime);
fprintf(status,"Corner Turn Time Taken: %ld minutes\n",(endtime-starttime)/60);
fprintf(status,"Corner Turn Module completed.\n\n");

/* Close all files and terminate the program: */

fclose(fpin);
fclose(fpout);
fclose(inparamfile);
fclose(status);
return(0);

};
```

```
/****************************************************************************/
/* MODULE:      AZIMCOM.C                                                   */
/*                                                                          */
/* Functions:   compress_azim_line() - Compress Single Azimuth line         */
/*              main() - Compress corner turned file of azimuth lines       */
/*                                                                          */
/* Author:      Simon Welsh                                                 */
/* Date:        01/09/91                                                    */
/****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "sar.h"
#include "param.h"
#include "window.h"

/****************************************************************************/
/* compress_azim_line()                                                      */
/* Compress a single azimuth line                                            */
/*                                                                           */
/* Input Parameters: xreal - Pointer to real output store                    */
/*                   ximag - Pointer to imag output store                    */
/*                   sinlook - Pointer to Sin lookup table for FFT           */
/*                   coslook - Pointer to Cos lookup table for FFT           */
/*                   azimrefreal - Pointer to Azimuth Reference Fn (real)   */
/*                   azimrefimag - Pointer to Azimuth Reference Fn (imag)   */
/*                                                                           */
/*                   aziminputline - Pointer to tempory input storage        */
/*                   azimtot - Total azimuth points to process               */
/*                   azimfftsize - Size of FFT to use                        */
/*                   fpin - Pointer to input data file                       */
/*                                                                           */
/* Steps Performed:                                                          */
/*                                                                           */
/* Read in an azimuth line from the data file.                               */
/* Perform FFT on the input azimuth line.                                    */
/* Multiply FFT of azimuth line by FFT of the azimuth reference function.   */
/* Perform Inverse FFT on the result.                                        */
/*                                                                           */
/****************************************************************************/

compress_azim_line(float xreal[],float ximag[],float azimrefreal[],float
azimrefimag[],float sinlook[],float coslook[],unsigned char aziminputline[],int
azimfftsize,int azimtot,FILE *fpin)
{
int     i;

/* Read in the input data from the file: */

  for(i=0;i<azimfftsize;xreal[i]=0,ximag[i]=0,i++);
  fread(aziminputline, 2, azimtot, fpin);
  for(i=0; i<azimtot; )
  { xreal[i] = (float)aziminputline[i*2]   - 128.0;
    ximag[i] = (float)aziminputline[i*2+1] - 128.0;
    i++; }
```

```
/* Perform forward FFT and multiply: */

  fft(xreal,ximag,sinlook,coslook,azimfftsize,1);
  multiply(xreal,ximag,azimrefreal,azimrefimag,azimfftsize);

/* Perform the inverse FFT: */

  fft(xreal,ximag,sinlook,coslook,azimfftsize,-1);
}

/****************************************************************************/
/* main()                                                                   */
/* Start of main program to compress azimuth lines                          */
/*                                                                          */
/* Steps Performed:                                                         */
/*                                                                          */
/* Extract Input Parameters.                                                */
/* Read in Azimuth Reference Function.                                      */
/* Perform FFT on azimuth reference Function and Conjugate it.              */
/*                                                                          */
/* Perform Sample Azimuth Compressions to determine output scaling factor   */
/* Perform Actual Azimuth Compressions with the computed output scaling     */
/* factor, and output to the main image file                                */
/*                                                                          */
/* Output data to the status file                                           */
/****************************************************************************/

main(int argc, char *argv[])
{
FILE    *fpin;
FILE    *fpout;
FILE    *status;
FILE    *inparamfile;
FILE    *azimrefdata;

char    corname[50];     /* Corner File output filename */
char    azmname[50];     /* Azimuth Compression output filename */
char    statusname[50]; /* Status filename */

/* Input parameter variable declarations: */

char    found;           /* Indicate if parameter was found */
int     azimfftsize;     /* Size of the FFT, closest to rangesamples */
int     azimtot;         /* Number of azimuth pulses to process */
int     rangebins;       /* Number of range compressed output rangebins */
float   azimscale;       /* Scaling factor for Azimuth Compression output */
int     windowtype;      /* Type of Window to apply */

/* Declare Pointers to Dynamic Heap Memory: */

float           *azimrefreal;
float           *azimrefimag;
float           *xreal;
float           *ximag;
float           *sinlook;
float           *coslook;
```

```
unsigned char   *aziminputline;

int      N,i,err,points;
float    realout, imagout;
int      rangeline, startrange;
long     starttime, endtime, outstart, outend;
long     outerror, zeroerror, maxouterror, maxzeroerror, outpos, zeropos;
float    mean, minmag, maxmag, magout;

unsigned char   buffer;
float           value;

/* Variables for determining Scaling Factor: */

double  peakmag;
double  fftmag;
double  avepeakmag;

/****************************************************************************/
/* Check input parameters:                                                  */
/****************************************************************************/

if (argc!=2)
{ printf("AZIMCOM paramfile\n");
  return(1); }

/**************************************************************************/
/* Open input parameter files:                                            */
/**************************************************************************/

inparamfile = fopen(argv[1],"r");
if (inparamfile==NULL)
{ printf("%s Input Parameter File Not Found - Aborting\n",argv[1]);
  return(1); }

/**************************************************************************/
/* Read in parameter values from the parameter files:                     */
/**************************************************************************/

getstringparam("corname",corname,inparamfile,&found);
if (!found) return(1);
getstringparam("azmname",azmname,inparamfile,&found);
if (!found) return(1);
getstringparam("statusname",statusname,inparamfile,&found);
if (!found) return(1);

azimtot = getfloatparam("pulses",inparamfile,&found);
if (!found) return(1);
rangebins = getfloatparam("rangebins",inparamfile,&found);   /* Output 2494 */
if (!found) return(1);
azimfftsize = getfloatparam("azimfftsize",inparamfile,&found);
if (!found) return(1);
windowtype = getfloatparam("window",inparamfile,&found);
if (!found) return(1);
```

```
/**************************************************************************/
/* Open the status file:                                                  */
/**************************************************************************/

status  = fopen(statusname,"a");
fprintf(status,"Start of the Azimuth Compression Module.\n");
fprintf(status,"========================================\n\n");
fflush(status);

/*****************************************************************************/
/* Allocate Dynamic Heap memory:                                             */
/*****************************************************************************/

aziminputline    = malloc( azimtot * 2 );
azimrefreal      = malloc( azimfftsize * sizeof(float) );
azimrefimag      = malloc( azimfftsize * sizeof(float) );
xreal            = malloc( azimfftsize * sizeof(float) );
ximag            = malloc( azimfftsize * sizeof(float) );
sinlook          = malloc( azimfftsize * sizeof(float) );
coslook          = malloc( azimfftsize * sizeof(float) );

if (coslook==NULL)
{ fprintf(status,"ERROR - Azimuth Compression Memory Allocation Error\n");
  fclose(status);
  return(1); }

/*****************************************************************************/
/* Open the input and output data files:                                     */
/*****************************************************************************/

printf("Input File:  %s\n",corname);
printf("Output Data: %s\n",azmname);

fpin  = fopen(corname,"rb");
fpout = fopen(azmname,"wb");

if (fpin==NULL)
{ fprintf(status,"ERROR: Azimuth Compression Cannot open %s - Aborting\n",
corname);
  fclose(status);
  return(1); }

rewind(fpin);

/*****************************************************************************/
/* Send Useful Information to data file:                                     */
/*****************************************************************************/

fprintf(status,"Input Data file %s\n",corname);
fprintf(status,"Output Data file %s\n",azmname);
fprintf(status,"Window type: %d\n",windowtype);
fflush(status);
```

```
/****************************************************************************/
/* Obtain Azimuth Reference Array from data file.                           */
/****************************************************************************/

printf("Extracting Azimuth Reference\n");
azimrefdata = fopen("azimref.dat","r");
if (azimrefdata==NULL)
{ fprintf(status,"ERROR: Cannot Open Azimuth Compression Reference file\n");
  fclose(status);
  return(1); }

for(i=0; i<azimfftsize; azimrefreal[i]=0, azimrefimag[i]=0, i++);
i=0;
while (  (fscanf(azimrefdata,"%g", &azimrefreal[i]) != EOF) &&
(fscanf(azimrefdata,"%g", &azimrefimag[i]) != EOF) )
  i++;
points=i;
for(i=0; i<points; printf("%d:  %g %g\n", i, azimrefreal[i], azimrefimag[i] ),
i++);

/****************************************************************************/
/* Perform FFT on the Azimuth reference array.                              */
/* Conjugate the FFT of the azimuth reference function.                     */
/* Multiply by the selected window function                                 */
/****************************************************************************/

printf("Points: %d\n",points);
loadsincos(sinlook,coslook,azimfftsize);
fft(azimrefreal,azimrefimag,sinlook,coslook,azimfftsize,1);
for(i=0;i<azimfftsize;azimrefimag[i]=-azimrefimag[i],i++);
window(azimrefreal,azimrefimag,azimfftsize,windowtype);

/****************************************************************************/
/* Determine scaling constant for output values based on average of peaks:  */
/****************************************************************************/

avepeakmag = 0;
for(rangeline=0; rangeline < rangebins; rangeline+=32)
{ printf("EXTRACT %d ", rangeline);

compress_azim_line(xreal,ximag,azimrefreal,azimrefimag,sinlook,coslook,aziminput
line,azimfftsize,azimtot,fpin);

/* Determine the position of the peak: */

  peakmag = 0;
  for(i=0; i<azimtot; i++ )
  { fftmag = (xreal[i] * xreal[i] + ximag[i] * ximag[i]);
    if (fftmag > peakmag)
      peakmag = fftmag; }

  avepeakmag += sqrt(peakmag);
  printf("Peak Magnitude: %lg\n",sqrt(peakmag));
}

avepeakmag/=(rangebins/32);
azimscale = 300.0 / avepeakmag ;
```

```
fprintf(status,"Output Scaling Factor: %g\n",azimscale);
fflush(status);

/****************************************************************************/
/* The reference azimuth correllation function has now been set up.        */
/* Start of loop to process multiple azimuth lines:                         */
/****************************************************************************/

maxouterror = 0;
maxzeroerror = 0;
time(&starttime);

for(rangeline=0; rangeline < rangebins; rangeline++)
{ time(&outstart);
  printf("EXTRACT %d ", rangeline);

compress_azim_line(xreal,ximag,azimrefreal,azimrefimag,sinlook,coslook,aziminput
line,azimfftsize,azimtot,fpin);

/****************************************************************************/
/* Output the compressed azimuth line to the output binary file:            */
/* Unsigned bytes are used (0 to 255).                                      */
/****************************************************************************/

  outerror = 1;
  zeroerror = 1;
  mean = 0;
  maxmag=0;
  minmag=10000;

  printf("Outputing ");
  for(i=0; i<azimtot; )
  { magout = azimscale * sqrt(xreal[i] * xreal[i] + ximag[i] * ximag[i]);
    mean += magout;
    if (magout>maxmag) maxmag=magout;
    if (magout<minmag) minmag=magout;

    buffer = (unsigned char) (magout);
    if (magout >  255) buffer = (unsigned char) 255;
    fwrite(&buffer, 1, 1, fpout);

    i++;
    if (magout > 255)
      outerror++;
    if (!((signed char)magout))
      zeroerror++;
  }

/* Display useful data on the screen: */

  printf("\n");
  printf("Out of Range errors: %ld    Percentage: %ld\n", outerror,
outerror*100/azimtot );
  printf("        Zero errors: %ld    Percentage: %ld\n", zeroerror,
zeroerror*100/azimtot );
  printf(" Ave Out Mag: %g ", (mean / azimtot) );
  printf(" Max Out Mag: %g ", maxmag );
```

```
   printf(" Min Out Mag: %g\n", minmag );
   if (outerror > maxouterror) {maxouterror = outerror; outpos=rangeline; }
   if (zeroerror > maxzeroerror) {maxzeroerror = zeroerror; zeropos=rangeline; }
   time(&outend);
   printf("Time Elapsed: %d sec  ", outend-outstart);
   printf("Time Remaining: %d min\n",((rangebins-rangeline) * (outend-
starttime)/(rangeline+1))/60 );
}
time(&endtime);

/*************************************************************************/
/* Write useful data to the status file:                                 */
/*************************************************************************/

fprintf(status,"Maximum out of range errors: %ld at %ld \n", maxouterror,
outpos);
fprintf(status,"Maximum zero errors: %ld at %ld \n", maxzeroerror, zeropos);
fprintf(status,"Number of Azimuth Reference points: %d\n",points);
fprintf(status,"Processing Time: %d min\n", (endtime - starttime)/60 );
fprintf(status,"Azimuth Compression Module Completed.\n\n");

/*************************************************************************/
/* Close all files and terminate the program:                            */
/*************************************************************************/

fclose(status);
fclose(fpout);
fclose(fpin);
fclose(inparamfile);
return(0);

}
```

APPENDIX B

Included Sub-Modules

EXTRACT.H - Extract Raw radar data
WINDOW.H - Provide window functions
SAR.H - Various Processing Routines
PARAM.H - Extract input parameters
MATRIX.H - Provide matrix utilities

```
/**************************************************************************/
/* MODULE:       EXTRACT.H                                                */
/*                                                                        */
/* Functions:    extract() - Extact Compacted Range Line from the raw     */
/*                            data file.                                  */
/*                                                                        */
/* Author:       Simon Welsh                                              */
/* Date:         01/09/91                                                 */
/**************************************************************************/

/**************************************************************************/
/* extract()                                                              */
/* Function to extract a single raw range line from the raw data file:    */
/*                                                                        */
/* Input Parameters:  satdat - pointer to raw data file                   */
/*                    rangeline - rangeline to extract                    */
/*                    range[] - array in which to place rangeline         */
/*                                                                        */
/* Output Parameters: totbins - Number of rangebins extracted             */
/*                                                                        */
/**************************************************************************/

#define reclen 770

extract(FILE *satdat, unsigned long rangeline, float range[], int *totbins)
{
char            buffer[4];
unsigned long   word32[reclen];
char            *ptr;
int             err, i, num, n;
unsigned char   raw[5];

printf("EXTRACTING %ld ",rangeline);

/* Open Input File */

if (satdat == NULL)
  printf("%s","Satdat File Open Error");
rewind(satdat);

/* Initialisation for main rangeline read loop: */

fseek( satdat, (rangeline * reclen * 4), SEEK_SET);
err = fread(word32, 1, 6, satdat);       /* Read In 6 bytes into buffer */

/* Read Next Record into word32 array: */

  ptr = word32;
  for (n=0; n<reclen; n++)
  { err = fread(buffer, 1, 4, satdat);    /* Read In rangeline */
    for (i=3; i>=0; *(ptr++) = buffer[i--]);
  }

/* Display Fixed Header Data for current Record: */
/*
    printf ("\nLine: %ld\n", (word32[5]>>8)&0xffffff);
    printf ("Bits per sample: %d\n", (word32[5]>>5)&0x03+3);
    printf ("Clock select: %f MHz\n", (word32[5]&0x20) ? 30.353 : 45.530);
    printf ("Spare rangeline count: %d\n", (word32[5]&0xf));
    printf ("Serial   state   Vector Bits 0-15: %X\n", (word32[9]>>16)&0xffff);
    printf ("Serial attitude Vector Bits 16-31: %X\n", (word32[9])&0xffff);
*/
/* Display All Header Words (32Bits) in record: */
/*
   for(num=0; num<13; num++)
   { printf("%4d: ", num);
    for (i=32;i>0;printf("%d",(word32[num]>>--i)&1));
    printf (" %lx\n", word32[num]);
   }
*/

/* Expand 6 bit Raw Data samples into range array: */
/* Samples are centred about zero. */

    for(num=13; num<696; num++)
    { for(i=0;i<5; raw[i] = (word32[num]>>(6*(4-i++)))&0x3f );
      for(i=0;i<5; range[(num-13)*5+i] = (float) (raw[i]-32), i++ );
    }

/* Zero the errors: */
/*
    for(i=2482; i<=2487; range[i]=0, i++);
*/
    *totbins = 3415;    /* Number of raw rangebins in record */
}
/***************************************************************************/
/* MODULE:       WINDOW.H                                                 */
/*                                                                        */
/* Additional modules required: MATH.H                                    */
/*                                                                        */
/* Functions:    fact() - return factorial                                */
/*               I0() - Return modified Bessel                            */
/*               window() - multiply input array by window                */
/*                                                                        */
/* Author:       Simon Welsh                                              */
/* Date:         01/09/91                                                 */
/***************************************************************************/

#define pi 3.141592654
#define alpha 1.5

/***************************************************************************/
/* Function to return the factorial of x:                                 */
/***************************************************************************/

long fact(x)
{
long    i;
long    fact;

fact=1;
for(i=1; i<=x; i++)
  fact*=i;
return(fact);
```

```
}

/*************************************************************************/
/* Function to return an approximation to the Bessel Function:          */
/*************************************************************************/

double I0( double x )
{
long    k;
double  sum;

sum = 0;
for(k=0; k<10; k++)
  sum += pow( (pow( (double) x/2, (double) k) / (double) fact(k)), 2);

return(sum);
}

/*************************************************************************/
/* window()                                                              */
/* Procedure to multiply complex input array by a window function:       */
/*                                                                       */
/* Input Params: real - pointer to real array to be multiplied by window */
/*               imag - pointer to imag array to be multiplied by window */
/*               N - Length of real and imag arrays                      */
/*               type - type of window with which to multiply            */
/*                                                                       */
/*************************************************************************/

window(float real[], float imag[], int N, int type)
{
FILE    *test;
int     n;
float   w;
double  x;

test = fopen("Window.Dat","w");

for(n=0; n<N; n++)
{ switch (type) {

/* No Windowing: */

  case 0:
  {
  w = 1;
  break;
  }

/* Raised Cosine Window: */

  case 1:
  {
  w = 0.5 * (1.0 - cos(2 * n * pi / N));
  fprintf(test,"%g\n",w);
  break;
  }
/* Keiser - Bessel Window: */

  case 2:
  {
  x = pi * alpha * sqrt( 1.0 - pow (((double) n - ((double) N / 2 )) / ((double)
N / 2),2)) ;
  w = (float) (IO(x) / IO(pi * alpha));
  fprintf(test,"%g\n",w);
  break;
  }

/* Cosine on pedestal Window: */

  case 3:
  {
  w = 0.5 + 0.5 * sin(n * pi / N);
  fprintf(test,"%g\n",w);
  break;
  }

/* Multiply the input real and imaginary componants by the window: */

  }
  real[(n + N/2) % N] *= w;
  imag[(n + N/2) % N] *= w; }

  fclose(test);
}
```

```
/****************************************************************************/
/* MODULE:      SAR.H                                                      */
/*                                                                         */
/* Additional modules required: MATH.H                                     */
/*                                                                         */
/* SAR library file                                                        */
/* This file contains the following externally callable functions:         */
/*                                                                         */
/* Functions: bitrev() - Perform Bit reversal for FFT function             */
/*            fft() - Perform forward or inverse fft                       */
/*            loadsincos() - Set up the Sin and Cos tables                 */
/*            multiply() - Multiply two complex numbers                    */
/*            sendout() - Output a test file                               */
/*                                                                         */
/* Author:      Simon Welsh                                                */
/* Date:        01/09/91                                                   */
/****************************************************************************/

/****************************************************************************/
/* bitrev()                                                                */
/* Function to perform a bit reversal on a single integer:                 */
/****************************************************************************/

long bitrev(long J, int NU)
{
long    J2,y,i;
y=0;
for(i=0;i<NU;i++)
{ J2=J/2;
  y=y*2+(J-2*J2);
  J=J2; }
return(y);
};

/****************************************************************************/
/* fft()                                                                   */
/* Procedure to perform an FFT:                                            */
/*                                                                         */
/* Input Parameters: xreal - Pointer to real array for input / output      */
/*                   ximag - Pointer to imaginary array for input / output */
/*                   sinlook - Pointer Sin lookup table                    */
/*                   coslook - Pointer Cos lookup table                    */
/*                   N - Number of points in the FFT                       */
/*                   sign - (1=Forward FFT)   (-1=Inverse FFT)             */
/*                                                                         */
/****************************************************************************/

fft(float xreal[], float ximag[], float sinlook[], float coslook[], int N, int
sign)
{
double          c,s;
float           treal;
float           timag;
int             NU,NU1,L,I,M,N2,K,i,KN2;

NU=log(N)/log(2);
N2=N/2;
NU1=NU-1;
K=0;

/* Conjugate Input for inverse FFT: */

if (sign==-1)
{ for(i=0; i<N; ximag[i]=-ximag[i], i++);
  printf("IFFT "); }
else
  printf("FFT ");

for(L=1;L<=NU;L++ )
{ do
  { for(I=1;I<=N2;I++ )
    { M = (K/(1<<NU1));
      KN2=K+N2;
      treal=xreal[KN2]*coslook[M] + ximag[KN2]*sinlook[M];
      timag=ximag[KN2]*coslook[M] - xreal[KN2]*sinlook[M];
      xreal[KN2]=xreal[K]-treal;
      ximag[KN2]=ximag[K]-timag;
      xreal[K]+=treal;
      ximag[K]+=timag;
      K++; }
    K+=N2;
    }
  while(K<N-1);
  K=0;
  NU1-=1;
  N2/=2;
}

/* Perform Bit Reversal: */

for(K=0;K<N;K++)
{
  I=bitrev(K,NU);
  if (I>K)
  { treal=xreal[K];
    timag=ximag[K];
    xreal[K]=xreal[I];
    ximag[K]=ximag[I];
    xreal[I]=treal;
    ximag[I]=timag;
  }
}

/* Conjugate and scale if IFFT: */

if (sign==-1)
{ for(i=0;i<N;ximag[i]=-ximag[i], i++ );
  for(i=0;i<N; xreal[i]/=(float)N, ximag[i]/=(float)N, i++ ); }

};
```

```
/****************************************************************************/
/* loadsincos()                                                             */
/* Procedure to load the sin and cos lookup tables:                         */
/*                                                                          */
/* Input Parameters: sinlook - Pointer Sin lookup table                     */
/*                   coslook - Pointer Cos lookup table                     */
/*                   N - Number of points in the FFT                        */
/*                                                                          */
/****************************************************************************/

#define pi 3.141592654

loadsincos(float sinlook[], float coslook[], int N)
{
int     i;
long    NU,P;
double  arg;

NU=log(N)/log(2);

printf("Loading Sin/Cos Lookup Tables.\n");
for(i=0;i<N;i++ )
{ P = bitrev(i,NU);
  arg = 2 * pi * (float)i / (float)N;
  sinlook[P]=sin(arg);
  coslook[P]=cos(arg); }
};

/****************************************************************************/
/* multiply()                                                               */
/* Procedure to multiply two complex arrays:                                */
/*                                                                          */
/* Input Parameters: real1 - Pointer to first real array                    */
/*                   imag1 - Pointer to first imaginary array               */
/*                   real2 - Pointer to second real array                   */
/*                   imag2 - Pointer to second imaginary array              */
/*                   N - Number of points in the arrays to multiply         */
/*                                                                          */
/* Output parameters: real1 - Points to the real product                    */
/*                    imag1 - Points to the imaginary product               */
/****************************************************************************/

multiply(float real1[], float imag1[], float real2[], float imag2[], int N)
{
int     i;
float   tempreal;

printf("MULT ");
for(i=0;i<N;i++)
{ tempreal = real1[i] * real2[i] - imag1[i] * imag2[i];
  imag1[i] = real1[i] * imag2[i] + imag1[i] * real2[i];
  real1[i] = tempreal; }
};
```

```
/*************************************************************************/
/* sendout()                                                             */
/* Procedure to output a test file:                                      */
/*                                                                       */
/* Input Parameters: real - Pointer to real array                        */
/*                   imag - Pointer to imaginary array                   */
/*                   N - Number of points in the array                   */
/*                   filename - filename of output file                  */
/*************************************************************************/

sendout( float real[], float imag[], int N, char *filename)
{
FILE    *fptr;
int     i;

fptr = fopen(filename,"w");
for(i=0; i<N; i++)
  fprintf(fptr, "%g %g %g\n",real[i], imag[i],
sqrt(real[i]*real[i]+imag[i]*imag[i]));

fclose(fptr);

}
```

```
/***************************************************************************/
/* MODULE:       PARAM.H                                                   */
/*                                                                         */
/* Functions: getfloatparam() - Get float parameter from parameter file    */
/*            getstringparam() - Get string parameter from parameter file */
/*            putfloatparam() - Put float parameter into parameter file    */
/*                                                                         */
/* Author:       Simon Welsh                                               */
/* Date:         01/09/91                                                  */
/***************************************************************************/

#include <string.h>

/***************************************************************************/
/* getfloatparam()                                                         */
/* Get a floating point parameter from the parameter file                  */
/*                                                                         */
/* Input Parameters:  getname[] - name of parameter                        */
/*                    paramfile - name of parameter file                   */
/*                                                                         */
/* Output Parameters: found - 1=Parameter Found / 0=Parameter Not Found    */
/*                                                                         */
/*                    return value = parameter value                       */
/***************************************************************************/

float getfloatparam( char getname[], FILE *paramfile, char *found)
{
  float value;
  int   i;
  char  paramname[30];

  value=0;
  rewind(paramfile);
  *found = 0;
  while (fscanf(paramfile, "%s %g",paramname,&value)!=EOF)
  { *found = !stricmp( getname, paramname );
    if (*found) break; }

  if (!(*found))
    printf("ERROR - %s Parameter not found\n",getname);
  else
    printf("%s = %g\n",paramname, value);

  return(value);
}
```

```
/***************************************************************************/
/* getstringparam()                                                        */
/* Get a string parameter from the parameter file                          */
/*                                                                         */
/* Input Parameters:  getname[] - name of parameter                        */
/*                    stringparam[] - array in which to place string       */
/*                    paramfile - name of parameter file                   */
/*                                                                         */
/* Output Parameters: found - 1=Parameter Found / 0=Parameter Not Found    */
/*                                                                         */
/***************************************************************************/

getstringparam( char getname[], char stringparam[], FILE *paramfile, char
*found)
{
  int   i;
  char  paramname[30];

  rewind(paramfile);
  *found = 0;
  while (fscanf(paramfile, "%s %s",paramname,stringparam)!=EOF)
  { *found = !stricmp( getname, paramname );
    if (*found) break; }

  if (!(*found))
    printf("ERROR - %s Parameter not found\n",getname);
  else
    printf("%s = %s\n",paramname, stringparam);

  return;
}

/***************************************************************************/
/* putfloatparam()                                                         */
/* Put a floating point parameter into a parameter file                    */
/*                                                                         */
/* Input Parameters:  putname[] - name of parameter                        */
/*                    value - value to assign to parameter                 */
/*                    paramfile - name of parameter file                   */
/*                                                                         */
/***************************************************************************/

putfloatparam( char putname[], float value, FILE *paramfile)
{
  printf("Parameter name: %s\n",putname);
  printf("Parameter Value: %g\n",value);

  fprintf(paramfile,"%s %g\n", putname, value);
}
```

```
/*************************************************************************/
/* MODULE:       MATRIX.H                                                */
/*                                                                       */
/* Functions:    magnitude() - Return Magnitude of input vector          */
/*               crossprod() - Compute cross product of 2 vectors        */
/*               constdiv() - Divide vector by a constant                */
/*               constmult() - Multiply vector by a constant             */
/*               vectsub() - Perform Vector Subtraction                  */
/*               vectcopy() - Copy vector                                */
/*               show() - Display vector                                 */
/*               matmult() - Multiply matrix by vector                   */
/*               mult() - Multiply vector by vector                      */
/*                                                                       */
/* Author:       Simon Welsh                                             */
/* Date:         01/09/91                                                */
/*************************************************************************/

#define vsize 3

/*************************************************************************/
/* magnitude()                                                           */
/* compute magnitude of input vector                                     */
/*                                                                       */
/* Input Parameters: vec - input vector                                  */
/* returns magnitude                                                     */
/*************************************************************************/

double magnitude( double vec[] )
{
  int i;
  double       sum;
  sum=0;
  for(i=0; i<vsize; i++)
    sum += vec[i]*vec[i];
  return( sqrt(sum) );
}

/*************************************************************************/
/* crossprod()                                                           */
/* Put cross-product of vectors a and b in c: (3-d only)                 */
/*************************************************************************/

crossprod( double a[], double b[], double c[] )
{
  c[0] = (a[1]*b[2] - a[2]*b[1]);
  c[1] = (a[2]*b[0] - a[0]*b[2]);
  c[2] = (a[0]*b[1] - a[1]*b[0]);
}

/*************************************************************************/
/* constdiv()                                                            */
/* Divide vector a by constant x and put result in vector b:             */
/*************************************************************************/

constdiv( double a[], double b[], double x )
{
  int i;
  for(i=0; i<vsize; i++)
    b[i] = a[i] / x;
}

/***************************************************************************/
/* constmult()                                                             */
/* Multiply vector by constant to produce vector:                          */
/***************************************************************************/

constmult( double a[], double b[], double x )
{
  int i;
  for(i=0; i<vsize; i++)
    b[i] = a[i] * x;
}

/***************************************************************************/
/* vectsub()                                                               */
/* Subtract Vector from Vector to produce vector:                          */
/***************************************************************************/

vectsub( double a[], double b[], double c[] )
{
  int i;
  for(i=0; i<vsize; i++)
    c[i] = a[i] - b[i];
}

/***************************************************************************/
/* vectcopy()                                                              */
/* Copy Vector A to Vector B:                                              */
/***************************************************************************/

vectcopy( double a[], double b[] )
{
  int i;
  for(i=0; i<vsize; i++)
    b[i] = a[i];
}

/***************************************************************************/
/* show()                                                                  */
/* Display Vector a:                                                       */
/***************************************************************************/

show( double a[] )
{
  int i;
  for(i=0; i<vsize; i++)
    printf( "%g ", a[i] );
  printf("\n");
}
```

```
/**************************************************************************/
/* matmult()                                                              */
/* Multiply matrix by vector to produce vector:                           */
/**************************************************************************/

matmult( double a[vsize][vsize], double b[vsize], double c[vsize])
{
  int row,col;
  for(row=0; row<vsize; row++)
  { c[row]=0;
    for(col=0; col<vsize; col++)
      c[row] += ((b[col]) * (a[col][row]));
  }
}

/**************************************************************************/
/* mult()                                                                 */
/* Multiply vector by vector to produce constant                          */
/**************************************************************************/

double mult( double a[], double b[] )
{
  int i;
  double c;
  c = 0;
  for(i=0; i<vsize; i++)
    c += (a[i] * b[i]);
  return( c );
}
```

APPENDIX C

Additional Modules

GETCFI.C - Extract CFI image
QUICKCFI.C - Extract unfocussed CFI image
PRINTCFI.C - Print CFI image on HP Laserjet
STAT.C - Provide Statistics for input file
REFORMAT.C - Reformat ASCII file
BTOA.C - Convert Binary file to ASCII

```
/***************************************************************************/
/* MODULE:      GETCFI.C                                                   */
/*                                                                         */
/* Functions:   main() - Extract CFI image from main image file            */
/*                                                                         */
/* Author:      Simon Welsh                                                */
/* Date:        01/09/91                                                   */
/***************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <conio.h>
#include "param.h"

#define cfisize 512

/***************************************************************************/
/* main()                                                                  */
/* Program to extract CFI image from azimuth compressed data file          */
/* Input Parameters: argc - Number of Command Line parameters              */
/*                   *argv[1] - Name input parameter file                  */
/*                   *argv[2] - Starting azimuth pulse for image           */
/*                   *argv[3] - Staring rangebin for image                 */
/***************************************************************************/

main(int argc, char *argv[])
{
FILE    *cfiout;
FILE    *azimcom;
FILE    *test;
FILE    *inparamfile;

char    azmname[50];     /* Azimuth Compression output filename */
char    outfile[50];     /* Output filename */
char    docfile[50];     /* Output document name */

int     startrange, endrange, startazim, endazim, linetot, i, err;
int     pos;
long    starttime, endtime;
float   outpix;
float   rangeline;

unsigned char   *range;
unsigned char   *outpixarray;

/* Input parameters: */

char    found;
int     azimtot;
float   azimave;         /* 1.64 Output every azimave azimuth pixels */
float   rangeave;        /* 1.00 Output every rangeave range pixels */
```

```
/****************************************************************************/
/* Check that three input parameters exist:                                 */
/****************************************************************************/

if (argc < 4)
{ printf("SYNTAX: getcfi paramfile startpulse startbin\n");
  return; }

/****************************************************************************/
/* Open input parameter files INPUT.PAR and SAT.PAR:                        */
/****************************************************************************/

inparamfile = fopen(argv[1],"r");
if (inparamfile==NULL)
{ printf("%s Input Parameter File Not Found - Aborting\n", argv[1]);
  return; }

getstringparam("azmname",azmname,inparamfile,&found);
if (!found) return;
azimtot = getfloatparam("pulses",inparamfile,&found);
if (!found) return;
azimave = getfloatparam("cfiazimsize",inparamfile,&found);
if (!found) return;
rangeave = getfloatparam("cfirangesize",inparamfile,&found);
if (!found) return;

/****************************************************************************/
/* Allocate memory for one azimuth line:                                    */
/****************************************************************************/

range  = malloc( cfisize * 2 * azimave);
outpixarray = malloc( cfisize );
if (outpixarray == NULL)
{ printf("Memory Allocation Error\n");
  return; }

/****************************************************************************/
/* Open the azimcom processed data file: */
/****************************************************************************/

azimcom = fopen(azmname,"rb");

sscanf(argv[2], "%d", &startazim);
sscanf(argv[3], "%d", &startrange);
printf("Starting Azimuth pulse: %d\n", startazim);
printf("    Starting Range bin: %d\n", startrange);
sprintf(outfile, "%d%d.cfi",startazim, startrange );
sprintf(docfile, "%d%d.doc",startazim, startrange );

printf("%s\n",outfile);
cfiout = fopen(outfile,"wb");

/****************************************************************************/
/* Open ASCII file and output useful data:                                  */
/****************************************************************************/

test = fopen(docfile,"w");
```

```
fprintf(test,"Azimuth compressed input filename: %s\n",azmname);
fprintf(test,"CFI Output Filename:  %d%d.cfi\n",startazim, startrange );
fprintf(test,"Starting Azimuth line: %d\n", startazim);
fprintf(test,"  Starting Range line: %d\n", startrange);
fprintf(test,"Azimuth Ave: %g\n",azimave);
fprintf(test,"  Range Ave: %g\n",rangeave);
fclose(test);

/***************************************************************************/
/* Start of the main CFI output loop:                                      */
/***************************************************************************/

time(&starttime);
endrange = startrange + (long) (cfisize * rangeave);
for( rangeline = (float)startrange; (long)rangeline < endrange; rangeline +=
rangeave)
{ fseek( azimcom, ((long)rangeline * (long)azimtot + (long)startazim),
SEEK_SET);
  err=fread(range, 1, (long) (cfisize * azimave), azimcom);
  for(i=0; i<cfisize; i++)
    outpixarray[i] = range[i*azimave];
  fwrite(outpixarray, cfisize, 1, cfiout);
  time(&endtime);
  printf("Time Remaining: %d sec  \r", (endrange-(long)rangeline) * (endtime-
starttime) / ((long)rangeline-startrange+1) );
}

/*****************************************************************************/
/* Close all files and terminate the program:                                */
/*****************************************************************************/

fclose(cfiout);
fclose(azimcom);

}
```

```
/*************************************************************************/
/* MODULE:      QUICKCFI.C                                               */
/*                                                                       */
/* Functions:   main() - Produce Unfocussed Image from Corner Turned File */
/*                                                                       */
/* Author:      Simon Welsh                                              */
/* Date:        01/09/91                                                 */
/*************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "param.h"

#define cfisize 512
#define scale 10.0

/*************************************************************************/
/* main()                                                                */
/* Produce Unfocussed CFI Image from Corner Turned File                  */
/*                                                                       */
/* Input Parameters: argc - Number of Command Line parameters            */
/*                   *argv[1] - Name input parameter file                */
/*                   *argv[2] - Starting azimuth pulse for image         */
/*                   *argv[3] - Staring rangebin for image               */
/*************************************************************************/

main(int argc, char *argv[])
{
FILE    *cfiout;
FILE    *test;
FILE    *azim;
FILE    *inparamfile;

char    outfile[50];    /* Output filename */
char    docfile[50];    /* Output document name */
char    corname[50];    /* Input filename */

unsigned char   *inputarray;
unsigned char   *outpixarray;
char            *runavereal;     /* The running average real store */
char            *runaveimag;     /* The running average imag store */
char            *xreal;          /* Floating Point real componant sample */
char            *ximag;          /* Floating Point imaginary componant sample */

int     i;
int     startazim;
int     startrange;
int     rangeline;
long    realave,imagave;
float   outmag;                  /* Magnitude of single sample */

unsigned char   cfibuf;          /* Output CFI buffer */
long    starttime,endtime;

/* Input parameters: */
```

```
char    found;
int     azimtot;
int     azimave;
int     azimsize;
int     rangesize;

/****************************************************************************/
/* Check that 4 input parameters exist:                                     */
/****************************************************************************/

if (argc < 4)
{ printf("SYNTAX: quickcfi paramfile startpulse startbin\n");
  return; }

printf("\n\nQuick Look CFI Image Generator\n\n");

/****************************************************************************/
/* Open input parameter files:                                              */
/****************************************************************************/

inparamfile = fopen(argv[1],"r");
if (inparamfile==NULL)
{ printf("%s Input Parameter File Not Found - Aborting\n", argv[1]);
  return; }

getstringparam("corname",corname,inparamfile,&found);
if (!found) return;
azimtot = getfloatparam("pulses",inparamfile,&found);
if (!found) return;
azimave = getfloatparam("quickazimave",inparamfile,&found);
if (!found) return;
azimsize = getfloatparam("quickazimsize",inparamfile,&found);
if (!found) return;
rangesize = getfloatparam("quickrangesize",inparamfile,&found);
if (!found) return;

/****************************************************************************/
/* Allocate memory for one azimuth line:                                    */
/****************************************************************************/

inputarray = malloc( cfisize * 2 * azimsize);
xreal = malloc( cfisize * azimsize);
ximag = malloc( cfisize * azimsize);
runavereal = calloc( azimave, 1 );
runaveimag = calloc( azimave, 1 );
outpixarray = malloc( cfisize );
if (outpixarray == NULL)
{ printf("Memory Allocation Error\n");
  return; }

/****************************************************************************/
/* Open the azimcom processed data file: */
/****************************************************************************/

azim = fopen(corname,"rb");
```

```
sscanf(argv[2], "%d", &startazim);
sscanf(argv[3], "%d", &startrange);
printf("Starting Azimuth pulse: %d\n", startazim);
printf("    Starting Range bin: %d\n", startrange);
sprintf(outfile, "Q%d%d.cfi",startazim, startrange );
sprintf(docfile, "Q%d%d.doc",startazim, startrange );

printf("Generating Output File: %s\n\n",outfile);
cfiout = fopen(outfile,"wb");

/***************************************************************************/
/* Open ASCII file and output useful data:                                 */
/***************************************************************************/

test = fopen(docfile,"w");
fprintf(test,"Range compressed and corner turned input filename: %s\n",corname);
fprintf(test,"CFI Output Filename:  %d%d.qfi\n",startazim, startrange );
fprintf(test,"Starting Azimuth line: %d\n", startazim);
fprintf(test,"  Starting Range line: %d\n", startrange);
fprintf(test,"Azimuth Ave: %d\n",azimave);
fprintf(test,"Range Size: %d\n",rangesize);
fprintf(test,"Azimuth Size: %d\n",azimsize);
fclose(test);

/***************************************************************************/
/* Start of the main CFI output loop:                                      */
/***************************************************************************/

time(&starttime);
for(rangeline=startrange; rangeline < (int) (startrange + cfisize * rangesize);
rangeline += (int)rangesize)
{
  fseek(azim, (((long)rangeline * (long)azimtot + (long)startazim) * 2),
SEEK_SET);

/* Read in one azimuth line of range compressed data into array: */
/* Transfer to real and imaginary arrays and centre about zero: */

  fread(inputarray, 1, (int) (cfisize * azimsize * 2), azim);
  for(i=0; i< cfisize * azimsize ; )
  { xreal[i] = (char) (inputarray[i*2]-128);
    ximag[i] = (char) (inputarray[i*2+1]-128);
    i++; }

/* Output a running averaged azimuth line: */

  realave=0;
  imagave=0;
  for(i=0; i<azimave; runavereal[i]=0, runaveimag[i]=0, i++ );
  for(i=0; i < (int) (cfisize * azimsize) ; i++ )
  {
    realave += (long) xreal[i];
    imagave += (long) ximag[i];
    realave -= (long) runavereal[i%azimave];
    imagave -= (long) runaveimag[i%azimave];
    runavereal[i%azimave] = xreal[i];
    runaveimag[i%azimave] = ximag[i];
    if (!(i%azimsize))
    { outmag = (long) ((float)scale / (float)azimave * sqrt
        ((float)realave * (float)realave + (float)imagave * (float)imagave ));
      if (outmag > 255)
        outpixarray[i/azimsize] = 255;
      else
        outpixarray[i/azimsize] = (unsigned char) outmag;
    }
  }

  fwrite(outpixarray, cfisize, 1, cfiout);
  time(&endtime);
  printf("Line: %4d   Time Remaining: %d min  \r", (rangeline-startrange+1) /
rangesize, ((cfisize * rangesize) - (rangeline-startrange+1)) * (endtime-
starttime) / (rangeline-startrange+1) / 60  );

}

fcloseall();
}
```

```
/*************************************************************************/
/* MODULE:       PRINTCFI.C                                             */
/*                                                                       */
/* Functions:    main() - Generate HP Laserjet Print file for CFI image  */
/*                                                                       */
/* Author:       Simon Welsh                                             */
/* Date:         01/09/91                                                */
/*************************************************************************/

#include <stdio.h>

/*************************************************************************/
/* main()                                                                */
/* Program to Generate an HP Laserjet Print file for a given CFI image   */
/* Input Parameters: argc - Number of Command Line parameters            */
/*                   *argv[1] - Input CFI image filename                 */
/*                   *argv[2] - Print Scaling Factor (Optional)          */
/*************************************************************************/

main (int argc, char *argv[])

{
  char filename[50];
  FILE *infile, *prn;
  int row, subrow, col, err, byte1, byte2;
  float scale;
  unsigned char rowbuffer [512], outbyte;

  char tile [16][4] = {
    {0, 0, 0, 0},                /* Gray Level 0 */
    {0, 0, 0, 4},                /* Gray Level 1 */
    {0, 4, 0, 1},                /* Gray Level 2 */
    {8, 1, 0, 4},                /* Gray Level 3 */
    {1, 8, 1, 2},                /* Gray Level 4 */
    {8, 1, 10, 4},               /* Gray Level 5 */
    {1, 10, 4, 10},              /* Gray Level 6 */
    {10, 5, 2, 5},               /* Gray Level 7 */

    {5, 10, 13, 10},             /* Gray Level 8 */
    {14, 5, 11, 5},              /* Gray Level 9 */
    {7, 14, 5, 11},              /* Gray Level 10 */
    {14, 7, 14, 13},             /* Gray Level 11 */
    {7, 14, 15, 11},             /* Gray Level 12 */
    {15, 11, 15, 14},            /* Gray Level 13 */
    {15, 15, 15, 11},            /* Gray Level 14 */
    {15, 15, 15, 15} };          /* Gray Level 15 */

/***************************************************************************/
/* Check for input parameters:                                             */
/***************************************************************************/

  if ((argc!=2) && (argc!=3))
  { printf("SYNTAX: printcfi filename scaling  (.CFI extension assumed) \n");
    return; }
/****************************************************************************/
/* Read in input parameters and open files:                                 */
/****************************************************************************/

  printf("\n\nLaserjet Series II CFI image printer\n\n");
  sprintf(filename, "%s.cfi", argv[1]);
  infile = fopen (filename, "rb");
  printf ("Input file: %s\n", filename);
  rewind(infile);
  sprintf(filename, "%s.prn", argv[1]);
  prn = fopen (filename, "wb");
  printf ("Output print file: %s\n",filename);
  printf ("Output print file should be copied to prn with /b option\n\n");

/****************************************************************************/
/* Set output scaling if specified on command line:                         */
/****************************************************************************/

  if (argc==3)
  { printf("Output Scaling Factor: %s\n",argv[2]);
    sscanf(argv[2],"%g",&scale); }
  else
    scale = 1.0;

/****************************************************************************/
/* Generate HP Laserjet compatible print file:                              */
/****************************************************************************/

  fprintf (prn, "\x1b*rB\x1b*t300R");
  for (row = 0; row < 512; row++)
    {
      printf("Row %3d\b\b\b\b\b\b\b",row);
      fread (rowbuffer, 1, 512, infile);
      for (subrow = 0; subrow < 4; subrow++)
        {
          fprintf (prn, "\x1b*r0A\x1b*b256W");
          for (col = 0; col < 256; col++)
            { byte1 = (rowbuffer[col*2]*scale)/16;
              byte2 = (rowbuffer[col*2+1]*scale)/16;
              outbyte = tile [ byte1<256 ? byte1 : 255 ][subrow] << 4;
              outbyte |= tile [ byte2<256 ? byte2 : 255 ][subrow];
              fprintf (prn, "%c", ~outbyte);
            }
          fprintf (prn, "\x1b*rB");
        }
    }
  fprintf (prn, "\x1b*rB\x0a\x0d");

/****************************************************************************/
/* Close input and output files:                                            */
/****************************************************************************/

  fclose(prn);
  fclose(infile);
}
```

```
/***************************************************************************/
/* MODULE:      STAT.C                                                    */
/*                                                                        */
/* Functions:   main() - Produce Statistics for input binary file         */
/*                                                                        */
/* Author:      Simon Welsh                                               */
/* Date:        01/09/91                                                  */
/***************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define bufsize 2048            /* (Maximum) input buffer size */

/***************************************************************************/
/* main()                                                                 */
/* Program to generate statistic file for given input binary file         */
/* Input Parameters: argc - Number of Command Line parameters             */
/*                   *argv[1] - Name of Input binary file                 */
/*                   *argv[2] - Name of output ASCII statistics file      */
/***************************************************************************/

main(int argc, char *argv[])
{
FILE    *infile;
FILE    *outfile;
unsigned char   buffer[bufsize];
unsigned long   stat[256];
unsigned long   i;
unsigned long   totread;
unsigned int    num;
float   mean;
float   sumsq;
float   sum;
float   sdev;

printf("\n\nStatistic Analysis of Byte File\n\n");

/***************************************************************************/
/* Test for correct number of parameters:                                 */
/***************************************************************************/

if (argc < 3)
{ printf("SYNTAX: STAT inputfile outputfile\n");
  return; }

/***************************************************************************/
/* Open the input and output files:                                       */
/***************************************************************************/

infile = fopen(argv[1],"rb");
if (infile==NULL)
{ printf("Input File Not Found: %s\n",argv[1]);
  return; }

outfile = fopen(argv[2],"w");
printf("Input Data File: %s\n",argv[1]);
printf("Output Statistic File: %s\n",argv[2]);

/*************************************************************************/
/* Loop through until end of file:                                      */
/*************************************************************************/

for(i=0; i<256; stat[i] = 0, i++);

rewind(infile);
printf("\nReading Input File\n");
totread=0;
sum=0;
sumsq=0;
do
{ num=fread(buffer,1,bufsize,infile);
  for(i=0; i<num; )
  { sum += (unsigned float) (buffer[i]);
    sumsq += (unsigned float) (buffer[i] * buffer[i]);
    (stat[buffer[i++]])++;  }
  totread+=num;
  printf("Bytes Read: %ld\r",totread); }
while (num==bufsize);

mean = sum/totread;
sdev = sqrt(abs(( sumsq - (sum*sum)/totread ) / (totread-1)));

printf("\n\nWriting Output File\n");
for(i=0; i<256; i++ )
  fprintf(outfile,"%ld %g%% %g\n",i, (100 * (float)stat[i] / (float)totread),
((float)stat[i]/((float)totread/256)) );
fprintf(outfile,"\n\nComputed Statistics:\n\n");
fprintf(outfile,"Samples: %ld\n",totread);
fprintf(outfile,"Sum: %g\n",sum);
fprintf(outfile,"Sum of Squares: %g\n",sumsq);
fprintf(outfile,"Mean: %g\n",mean);
fprintf(outfile,"Standard deviation: %g\n",sdev);

fclose(infile);
fclose(outfile);

}
```

```
/***************************************************************************/
/* MODULE:      REFORMAT.C                                                 */
/*                                                                         */
/* Functions:   main() - Reformat Input ASCII file to Output ASCII file    */
/*                       with a given number of numbers per line          */
/*                                                                         */
/* Author:      Simon Welsh                                                */
/* Date:        01/09/91                                                   */
/***************************************************************************/

#include <stdio.h>
#include <stdlib.h>

/***************************************************************************/
/* main()                                                                  */
/* Program reformat ASCII input file                                       */
/* Input Parameters: argc - Number of Command Line parameters              */
/*                   *argv[1] - Name of Input ASCII file                   */
/*                   *argv[2] - Name of output ASCII file                  */
/*                   *argv[3] - Number of numbers to output per line       */
/***************************************************************************/

main(int argc, char *argv[])
{
FILE    *infile, *outfile;
int     num,i;
double  value;

if (argc!=4)
{ printf("SYNTAX: reformat inputfile outputfile num\n");
  return(1); }

infile = fopen(argv[1], "r");
outfile = fopen(argv[2], "w");
sscanf( argv[3], "%d", &num);
printf("Outputting %d values per line.\n",num);

if (infile==NULL)
{ printf("ERROR - Input file not found\n");
  return(2); }

i=0;
while(fscanf(infile, "%lg", &value)!=EOF)
{ fprintf(outfile," %lg",value);
  if (!(++i%num)) fprintf(outfile,"\n");
}

fprintf(outfile,"\n");
fcloseall();

}
```

```
/****************************************************************************/
/* MODULE:      BTOA.C                                                      */
/*                                                                          */
/* Functions:   main() - Convert input binary file to ASCII output file     */
/*                                                                          */
/* Author:      Simon Welsh                                                 */
/* Date:        01/09/91                                                    */
/****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/****************************************************************************/
/* main()                                                                   */
/* Input Parameters: argc - Number of Command Line parameters               */
/*                   *argv[1] - Name of Input binary file                   */
/*                   *argv[2] - Name of output ASCII file                   */
/*                   *argv[3] - Number of ASCII numbers output per line     */
/****************************************************************************/

main(int argc, char *argv[])
{
FILE *fpin,*fpout;

unsigned char value;

int     count;
int     num;

if (argc < 4)
{ printf("BTOA inputfile outputfile numperline\n");
  return; }

sscanf(argv[3],"%d",&num);

printf("Input Binary File: %s\n Output ASCII File: %s\n",argv[1], argv[2]);
printf("%d Numbers per line\n",num);

fpin  = fopen(argv[1],"rb");
if (fpin==NULL)
{ printf("Input File Not Found\n");
  return; }
fpout = fopen(argv[2],"w");
rewind(fpin);

count=0;
while (fread(&value, sizeof(value), 1, fpin) != NULL)
{ fprintf(fpout, "%d ", value);
  if (!((count+1)%num)) fprintf(fpout, "\n");
  count++;
  }

fclose(fpout);
fclose(fpin);
}
```

APPENDIX D

Test Programs

RNGFILE.C - Create Range Compression Test file
AZMFILE.C - Create Azimuth Compression Test file
CORFILE.C - Create Corner Turn Test file
TESTCOR.C - Test the corner turn routine

```
/*************************************************************************/
/* MODULE:      RNGFILE.C                                               */
/*                                                                       */
/* Functions:   main() - Generate Range Compression Test File           */
/*                                                                       */
/* Author:      Simon Welsh                                             */
/* Date:        01/09/91                                                */
/*************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include "param.h"

#define rangebins 3415
#define azimtot 4096
#define points 923
#define testpos 512
#define sumnum 1
#define dist 10

/*************************************************************************/
/* Start of Program to generate test input file for range compression    */
/*************************************************************************/

main(int argc, char *argv[])
{
FILE            *infile;
FILE            *outfile;
unsigned char   testbyte;
unsigned char   *buffer;
unsigned char   *outbuf;
int             *testline;

int             i,j;
float           *rangerefreal;
float           *rangerefimag;

printf("\n\nGenerating Test File for Range Compression\n\n");

/*************************************************************************/
/* Allocate Memory:                                                      */
/*************************************************************************/

rangerefreal = malloc( points * sizeof(float) );
rangerefimag = malloc( points * sizeof(float) );
testline = malloc( 2 * sizeof(int) * rangebins );
buffer   = malloc( 2 * rangebins );
outbuf   = malloc( 2 * rangebins );
if (outbuf==NULL)
{ printf("Memory Allocation Error\n");
  return; }

/*************************************************************************/
/* Open the input and output files:                                      */
/*************************************************************************/

infile = fopen("RANGEREF.DAT","r");
if (infile==NULL)
{ printf("RANGEREF.DAT Input File Not Found: \n");
  return; }

for(i=0; i<points; rangerefreal[i]=0, rangerefimag[i]=0, i++);
i=0;
while (  (fscanf(infile,"%g", &rangerefreal[i]) != EOF) )
  i++;

outfile = fopen("range.dat","wb");
if (outfile==NULL)
  return;

/***************************************************************************/
/* Loop through until end of file:                                         */
/***************************************************************************/

/* Set up the blank buffer: */

for(i=0; i<rangebins; buffer[i]=128.0, i++);

/* Clear the test buffer: */

for(i=0; i<rangebins; testline[i]=0, i++);

/* Set up the test azimuth line with an array of point targets: */

for(i=0; i<points; )
{ for(j=512; j<512+(sumnum*dist); j+=dist )
  { testline[(j+i)]   += (rangerefreal[i] * 32.0 / sumnum); }
  i++; }

/* Transfer Test line to output buffer centred at +32: */

for(i=0; i<rangebins; )
{ outbuf[i] = testline[i] + 32;
  if (testline[i] >= 32) outbuf[i] = 63;
  if (testline[i] <= -32) outbuf[i] = 0;
  i++; }

/* Output test line: */

fwrite(outbuf, 1, rangebins, outfile);

/* Close files and terminate program: */

fclose(infile);
fclose(outfile);

}
```

```
/**************************************************************************/
/* MODULE:      AZMFILE.C                                                  */
/*                                                                         */
/* Functions:   main() - Generate Test file for azimuth compression        */
/*                                                                         */
/* Author:      Simon Welsh                                                */
/* Date:        01/09/91                                                   */
/**************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include "param.h"

#define rangebins 2494
#define azimtot 4096
#define points 512
#define testpos 1500
#define sumnum 2
#define dist 10

/***************************************************************************/
/* Start of Program to generate test input file for azimuth compression    */
/***************************************************************************/

main(int argc, char *argv[])
{
FILE            *infile;
FILE            *outfile;
unsigned char   testbyte;
unsigned char   *buffer;
unsigned char   *outbuf;
int             *testline;

int             i,j;
float           *azimrefreal;
float           *azimrefimag;

printf("\n\nGenerating Test File for Azimuth Compression\n\n");

/***************************************************************************/
/* Allocate Memory:                                                        */
/***************************************************************************/

azimrefreal = malloc( points * sizeof(float) );
azimrefimag = malloc( points * sizeof(float) );
testline = malloc( 2 * sizeof(int) * azimtot );
buffer   = malloc( 2 * azimtot );
outbuf   = malloc( 2 * azimtot );
if (outbuf==NULL)
{ printf("Memory Allocation Error\n");
  return; }

/***************************************************************************/
/* Open the input and output files:                                        */
/***************************************************************************/

infile = fopen("AZIMREF.DAT","r");
```

```
if (infile==NULL)
{ printf("AZIMREF.DAT Input File Not Found: \n");
  return; }

for(i=0; i<points; azimrefreal[i]=0, azimrefimag[i]=0, i++);
i=0;
while (  (fscanf(infile,"%g", &azimrefreal[i]) != EOF) && (fscanf(infile,"%g",
&azimrefimag[i]) != EOF) )
  i++;

outfile = fopen("azim.dat","wb");
if (outfile==NULL)
  return;

/*************************************************************************/
/* Loop through until end of file:                                       */
/*************************************************************************/

/* Set up the blank buffer: */
for(i=0; i<azimtot*2; buffer[i]=128.0, i++);

/* Clear the test buffer: */
for(i=0; i<azimtot*2; testline[i]=0, i++);

/* Set up the test azimuth line with an array of point targets: */

for(i=0; i<points; )
{ for(j=1024; j<1024+(sumnum*dist); j+=dist )
  { testline[(j+i)*2]   += (azimrefreal[i] * 127.0 / sumnum);
    testline[(j+i)*2+1] += (azimrefimag[i] * 127.0 / sumnum); }
  i++; }


/* Set up the test azimuth line with an array of sweep targets: */
/*
for(i=1024; i<1024+sumnum; )
{ testline[(i)*2]   = (127.0);
  testline[(i)*2+1] = (127.0);
  i++; }
*/
/* Transfer Test line to output buffer centred at +128: */
for(i=0; i<azimtot*2; )
{ outbuf[i] = testline[i] + 128;
  if (testline[i] >= 127) outbuf[i] = 255;
  if (testline[i] <= -128) outbuf[i] = 0;
  i++; }

/* Output test line: */
fwrite(outbuf, 1, azimtot*2, outfile);

/* Output blank lines: */
/*
for(i=1; i<testpos; fwrite(buffer, 1, azimtot*2, outfile), i++);
*/

/* Output test line: */
/*
```

```
fwrite(outbuf, 1, azimtot*2, outfile);
*/

/* Output blank lines: */
/*
for(i=testpos; i<rangebins-1; fwrite(buffer, 1, azimtot*2, outfile), i++);
*/

/* Close files and terminate program: */
fclose(infile);
fclose(outfile);

}
```

```
/*************************************************************************/
/* MODULE:      CORFILE.C                                                */
/*                                                                       */
/* Functions:   main() - Generate Corner Turn Test File                  */
/*                                                                       */
/* Author:      Simon Welsh                                              */
/* Date:        01/09/91                                                 */
/*************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>

/*************************************************************************/
/* Program to generate corner turn test file:                            */
/*************************************************************************/

#define maxazim   1024  /* 4096 Maximum number of azimuth lines */
#define maxrange  512   /* 3415 Maximum number of rangebins */

main(int argc, char *argv[])
{
FILE    *fpout;
unsigned char *testbuf, *blank;
long    azim,i;

fpout  = fopen("rangecom.dat","wb");

testbuf = malloc( maxrange * 2 );
blank   = malloc( maxrange * 2 );
for(i=0; i<maxrange*2; blank[i]=0, i++);
for(i=0; i<maxrange*2; testbuf[i] = (i/2)%256, testbuf[i+1] = (i/2)%256, i+=2 );

fwrite(testbuf, 2, maxrange, fpout);
for(azim=1; azim<maxazim; )
{ fwrite(blank, 2, maxrange, fpout);
  azim++;
}

fclose(fpout);
};
```

```
/***************************************************************************/
/* MODULE:       TESTCOR.C                                                 */
/*                                                                         */
/* Functions:    main() - Test the corner turn program by comparing the    */
/*                        input and output files.                          */
/*                                                                         */
/* Author:       Simon Welsh                                               */
/* Date:         01/09/91                                                  */
/***************************************************************************/

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <alloc.h>
#include <math.h>

/***************************************************************************/
/* Start of the main program:                                              */
/***************************************************************************/

#define maxazim   4096  /* 4096 Maximum number of azimuth lines */
#define maxrange  3415  /* 3415 Maximum number of rangebins */

main(int argc, char *argv[])
{
FILE    *corner, *rangecom;
unsigned char *test, *test2;

long    rangeline, testazim;
int     error;

/***************************************************************************/
/* Allocate memory and Open files:                                         */
/***************************************************************************/

test = malloc( maxazim * 2 );
test2 = malloc( maxazim * 2 );
if (test2==NULL) return;

corner  = fopen("azim.dat","rb");
rangecom = fopen("rangecom.dat","rb");
if (corner==NULL) return;
if (rangecom==NULL) return;
rewind(corner);
rewind(rangecom);

/***************************************************************************/
/* Loop to compare all azimuth lines:                                      */
/***************************************************************************/

for(testazim = 0; testazim<maxazim; testazim++)
{
  printf("\nAzimuth Line: %ld ",testazim);
  for(rangeline=0; rangeline < maxrange; )
  {
    fseek(rangecom, (2 * ((long)testazim * (long)maxrange + (long)rangeline)),
SEEK_SET);
```

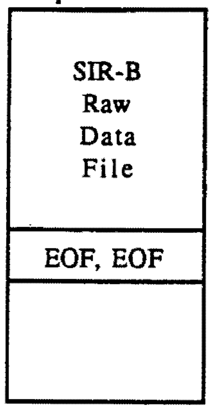
```
    fread(test2, 1, 2, rangecom);
    fseek(corner, (2 * ((long)testazim + (long)maxazim * ((long)maxrange - 1 -
(long)rangeline))), SEEK_SET);
    fread(test, 2, 1, corner);

/*
    printf("%2X  %2X  ", test2[0], test2[1]);
    printf("%2X  %2X  ", test[0], test[1]);
*/

    if (!(rangeline%100)) printf("%d ", rangeline);
    if ((test[0]!=test2[0]) || (test[1]!=test2[1]))
    { printf("\nERROR found in Corner Turn File\n");
      printf("Azimuth Line: %ld    Rangeline: %ld\n\n",testazim,rangeline);
      return; }

    rangeline++;
  }
}

/***************************************************************************/
/* Close files and terminate program:                                      */
/***************************************************************************/

fclose(corner);
fclose(rangecom);
};
```

APPENDIX E

SIR-B Raw Data Format

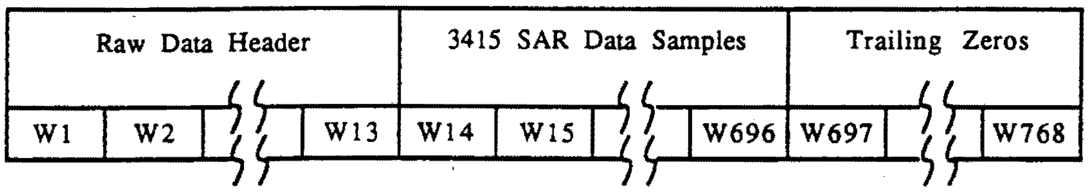
SIR-B Raw Data Format

The SIR-B raw data sets you have requested are stored on two 9-track, 6250 bpi computer compatible tapes (CCT). There is one file on each raw data tape. The tapes are labelled with the data take number and the time of the data.



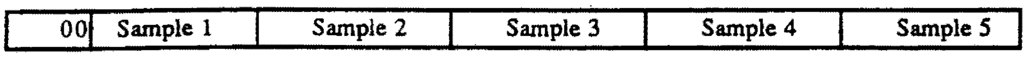
SIR-B raw data tape

Each record on the tape is a range line of the raw data set and contains 768 words *(a word is defined to be 32-bits)*. There are 21K records *(i.e., 21504 records)* in each file. The first 13 words of the raw data record consist of header information. The actual raw data begins at the 14th word. Documentation on the raw data header is attached.



One raw data record

Each range line contains 3415 SAR data samples followed by 288 bytes of trailing zeros. The raw data for data take X1-035.60 were digitized using 6 bits/sample. Five samples are packed, right-justified within each 32-bit word. The two most significant bits of each word are zeros.



One 32-bit word containing 5 raw data samples -- "packed" format