

AD-A241 283



2

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS



DOPPLER PROCESSING OF PHASE ENCODED
UNDERWATER ACOUSTIC SIGNALS

by

Randy M. Eldred

September 1990

Thesis Advisor:

James H. Miller

Approved for public release; distribution is unlimited

91-10-3 008

91-12260



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE			5 MONITORING ORGANIZATION REPORT NUMBER(S)		
4. PERFORMING ORGANIZATION REPORT NUMBER(S)			7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School		
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School		6b OFFICE SYMBOL (If applicable) EC	7b. ADDRESS (City, State, and ZIP Code)		
6c. ADDRESS (City, State, and ZIP Code)			9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8a. NAME OF FUNDING / SPONSORING ORGANIZATION		8b OFFICE SYMBOL (If applicable)	10 SOURCE OF FUNDING NUMBERS		
8c. ADDRESS (City, State, and ZIP Code)			PROGRAM ELEMENT NO	PROJECT NO	TASK NO
					WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) DOPPLER PROCESSING OF PHASE ENCODED UNDERWATER ACOUSTIC SIGNALS					
12 PERSONAL AUTHOR(S) ELDRED, Randy M.					
13a. TYPE OF REPORT Master's Thesis		13b TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Year, Month, Day) 1990 September	
15 PAGE COUNT 108					
16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the US Government.					
17 COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	acoustic tomography; Fast Hadamard Transform; maximal-length sequences; Doppler Processing		
19 ABSTRACT (Continue on reverse if necessary and identify by block number) Travel time of an acoustic signal from transmitter to receiver provides a great deal of information about the ocean environment. Variations in the travel time of the signal may be caused by the changes in the sound speed along the path. Since sound speed is a function of pressure, temperature and salinity, measurement of this parameter in acoustic tomography provides a means to observe ocean fluctuations through the use of inverse techniques. The upcoming Heard Island Experiment will attempt to determine the feasibility of measuring global warming by measuring changes in signal travel time that may be caused by temperature changes in the world's oceans. The signals to be transmitted in this experiment are phase encoded maximal-length sequences of various lengths which are well suited to measurement of travel time. The objectives of this thesis are to provide					
20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS			21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a NAME OF RESPONSIBLE INDIVIDUAL MILLER, James H.			22b TELEPHONE (Include Area Code) 408-646-2384		22c OFFICE SYMBOL EC/Mr

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

19. cont.

a software package, in C, that will allow participation as a receiver in this experiment, and to provide a general capability to process any maximal-length sequence, transmitted at any carrier frequency and with any reasonable Doppler. A background on wave propagation, maximal-length sequences, and Doppler processing are presented in this thesis.



Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Approved for public release; distribution is unlimited.

**Doppler Processing of Phase Encoded Underwater Acoustic
Signals**

by

**Randy Michael Eldred
Lieutenant, United States Navy
B.S., University of Delaware, 1983**

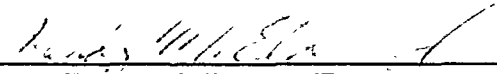
Submitted in partial fulfillment of the requirements
for the degree of

**MASTER OF SCIENCE IN ELECTRICAL
ENGINEERING**

from the

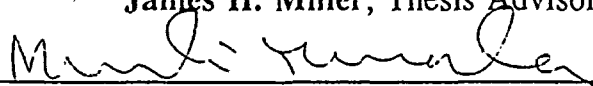
**NAVAL POSTGRADUATE SCHOOL
September 1990**

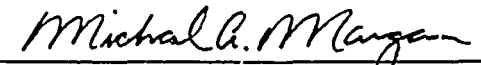
Author:


Randy Michael Eldred

Approved by:


James H. Miller, Thesis Advisor


Murali Tummala, Second Reader


Michael A. Morgan, Chairman, Department of Electrical and
Computer Engineering

ABSTRACT

Travel time of an acoustic signal from transmitter to receiver provides a great deal of information about the ocean environment. Variations in the travel time of the signal may be caused by the changes in the sound speed along the path. Since sound speed is a function of pressure, temperature and salinity, measurement of this parameter in acoustic tomography provides a means to observe ocean fluctuations through the use of inverse techniques. The upcoming Heard Island Experiment will attempt to determine the feasibility of measuring global warming by measuring changes in signal travel time that may be caused by temperature changes in the world's oceans. The signals to be transmitted in this experiment are phase encoded maximal-length sequences of various lengths which are well suited to measurement of travel time. The objectives of this thesis are to provide a software package, in C, that will allow participation as a receiver in this experiment, and to provide a general capability to process any maximal-length sequence, transmitted at any carrier frequency and with any reasonable Doppler. A background on wave propagation, maximal-length sequences, and Doppler processing are presented in this thesis.

TABLE OF CONTENTS

I. INTRODUCTION.....	1
A. THESIS SUMMARY	1
B. THE HEARD ISLAND EXPERIMENT	3
II. ACOUSTIC WAVE PROPAGATION.....	6
A. THE WAVE EQUATIONS	6
1. Plane Wave Equation	6
2. Spherical Wave Equation	7
B. RAYPATH DETERMINATION.....	8
1. Sound Speed	9
2. Snell's Law	9
3. Travel Time	12
III. M-SEQUENCES AND HADAMARD PROCESSING.....	13
A. INTRODUCTION.....	13
B. SHIFT REGISTER FUNDAMENTALS	14
C. THE FAST HADAMARD TRANSFORM.....	17
IV. TIME-VARYING DOPPLER PROCESSING	24
A. PHASE ENCODING OF M-SEQUENCES.....	24
B. SIGNAL PROCESSING WITH ZERO DOPPLER.....	24
C. SIGNAL PROCESSING WITH DOPPLER.....	28
V. RESULTS AND CONCLUSIONS.....	34
A. RESULTS	34
B. CONCLUSIONS.....	39

C. ADDITIONAL WORK.....	39
APPENDIX A	41
APPENDIX B.....	56
A. MAIN PROGRAM.....	56
1. MACROFILE.....	56
2. SEQREM.....	56
B. INITIALIZATION PROGRAMS.....	68
1. INIT_HAD.....	68
2. FWD_HAD.....	69
3. REV_HAD.....	72
4. GET_FLT_COEF.....	73
C. DEMODULATION AND FILTERING.....	74
1. DEMODULATE	74
2. HIGHPASS.....	76
3. LOWPASS1	78
4. LOWPASS2.....	80
5. LOWPASS3.....	82
D. FAST HADAMARD TRANSFORM (FHT).....	85
1. HADAMARD.....	85
E. MAGNITUDE AND PHASE.....	86
1. MAG_PHASE	86
F. UTILITY PROGRAMS	87
1. SET_FILTER.....	87
2. APLOT.....	88
3. DOPLOT.....	90
4. MGEN.....	91
5. MAKEFILE.....	93
6. EXAMPLE FILTER COEFFICIENT FILE	93

REFERENCES.....	95
INITIAL DISTRIBUTION LIST.....	98

ACKNOWLEDGEMENTS

This work may not have been completed without the help of a select group of individuals. First, I thank my thesis advisor, James H. Miller, for his never ending patience and help when I was confused, and for always being available when a problem cropped up. For Kurt Metzger and Ted Birdsall, I would like to express my appreciation for the papers and initial code they provided as a starting point in this thesis. Also, I thank my parents, Deane and Charla Eldred, for their support during those starting years, which without I would have never started. I would like to thank my in-laws, Harold and Florence Plympton, for their encouragement and pride in me. And most of all, I thank my wife Michele for her love, and undying support. She kept me going, even when I didn't want to go.

I. INTRODUCTION

A. THESIS SUMMARY

The objective of this thesis is to develop a program in the C programming language [Ref. 1] that can process acoustic signals used in ocean acoustic tomography. The goal of tomography signal processing is the precise measurement of acoustic travel time, the integral along the raypath of inverse sound speed. Travel time is a fairly well understood function of temperature, salinity, and pressure. Once variations in the travel time are known, as well as the travel times for multiple arrival paths, the fluctuations of the ocean can be determined from mathematical inverse techniques [Ref. 2].

One method in which travel time can be measured is through the use of explosive and implosive devices. These crude tools, however, are not exactly repeatable. A better method that has been found employs the use of maximal-length sequences as a phase modulation for bandpass signals. Maximal-length sequences (m-sequences) or pseudo-random noise are well suited for this application because of their deterministic nature, correlation properties, and simplicity.

The goal of the work described in this thesis was to be able to measure arrival time by performing replica cross-correlation of a transmitted acoustic signal that has been phase-encoded using m-sequences, and to be able to detect that signal over any reasonable Doppler shift. Specifically, the programming package should be able to:

1. Detect any phase encoded m-sequence transmitted with any carrier frequency.
2. Scan over any reasonable Doppler range at any Doppler bin spacing.
3. Provide filtering capability with any type filter.
4. Employ fast algorithms to minimize processing time.
5. Be well documented and portable for future transfer to a dedicated machine for real time processing.

The body of this thesis is structured as follows:

Chapter II. Acoustic Wave Propagation.

Chapter III. M-Sequences and Hadamard Processing.

Chapter IV. Time-Varying Doppler Processing.

Chapter V. Results and Conclusions.

Chapter II presents an introduction to acoustic wave propagation in the ocean environment. It includes plane wave propagation, spherical wave propagation, the dependencies of sound speed on temperature and other parameters. Acoustic raypath determination is discussed.

The third chapter is an introduction to general shift registers and m-sequences. The Fast Hadamard Transform is reviewed to provide the background necessary to understand the programming package.

Chapter IV presents the signal processing aspects of the thesis. It first presents basic phase encoding using m-sequences, and then develops the treatment of a signal with zero Doppler. And finally, the nonzero Doppler case is presented.

Appendix A contains the results from all Doppler considerations. Appendix B is the source code for the thesis work as well as some supplementary

programs. Supplementary programs include: plotting routines using VAX NCAR Graphics [Ref. 3], a routine for generating shift register states and corresponding m-sequences in a (1,-1) format for viewing before processing, if desired. Finally, a MATLAB program that can be used for generating filter coefficients for a Butterworth bandpass filter and a Chebychev lowpass filter. The filter coefficients are stored to a file that can be read by the main program SEQREM.

Documentation of the source code should be sufficient for the typical user to understand the general operation of the program without much difficulty, and is built using standard C functions and coding to minimize any portability problems.

B. THE HEARD ISLAND EXPERIMENT

The first direct application of this thesis is planned for the beginning of 1991. From January 23 - February 4, 1991 an experiment to determine the feasibility of measuring global warming using underwater acoustic signals is to take place and is currently designated the Heard Island Experiment [Ref. 4]. It is hoped that changes in the temperature of the Earth's ecosystem can be measured by observing changes in travel time of an acoustic signal over distances which include one or more oceans. Travel time is related to water temperature, as well as other parameters, so that a change in travel time might be related to warming of the Earth's atmosphere, which would also cause a overall warming of the world's oceans. If these two parameters can be related, then existence of global warming might be determined.

The signals will be transmitted from the vicinity of Heard Island in the southern Indian Ocean with raypaths that reach several receiving sites in the Indian Ocean, the Atlantic, and the Pacific as shown in Figure 1.1. The

transmitting vessel will be the m/v Cory Chouest. Preliminary ray traces have shown that one possible reception site is the Monterey Peninsula [Ref. 5]. Participation is intended by the Naval Postgraduate School, and is one motivation for this thesis. Of the many signals that will be transmitted, one is an m-sequence of 255 digits with a Q of 5, and a carrier frequency of 57 Hz, where Q is the ratio of digit frequency to carrier frequency and is normally selected to be an integer number. A carrier of 57 Hz was chosen because of its long propagation distance and also to help distinguish it from the common frequencies 50 and 60 Hz that are generated by a large number of power plants and machinery world wide.

Detailed signal specifications are contained in Reference 4, but some of the more general data are:

1. HLF4LL source.
2. Output power: 209 dB.
3. Signal to be transmitted using m-sequences

$$s(t) = A \cos(2\pi f_c t + M(t)\psi)$$

where $f_c = 57$ Hz.

$M(t)$ is the m-sequence.

$\psi = 45$ degrees, is the modulation angle.

4. Maximum of 3-5 knots to maintain way.

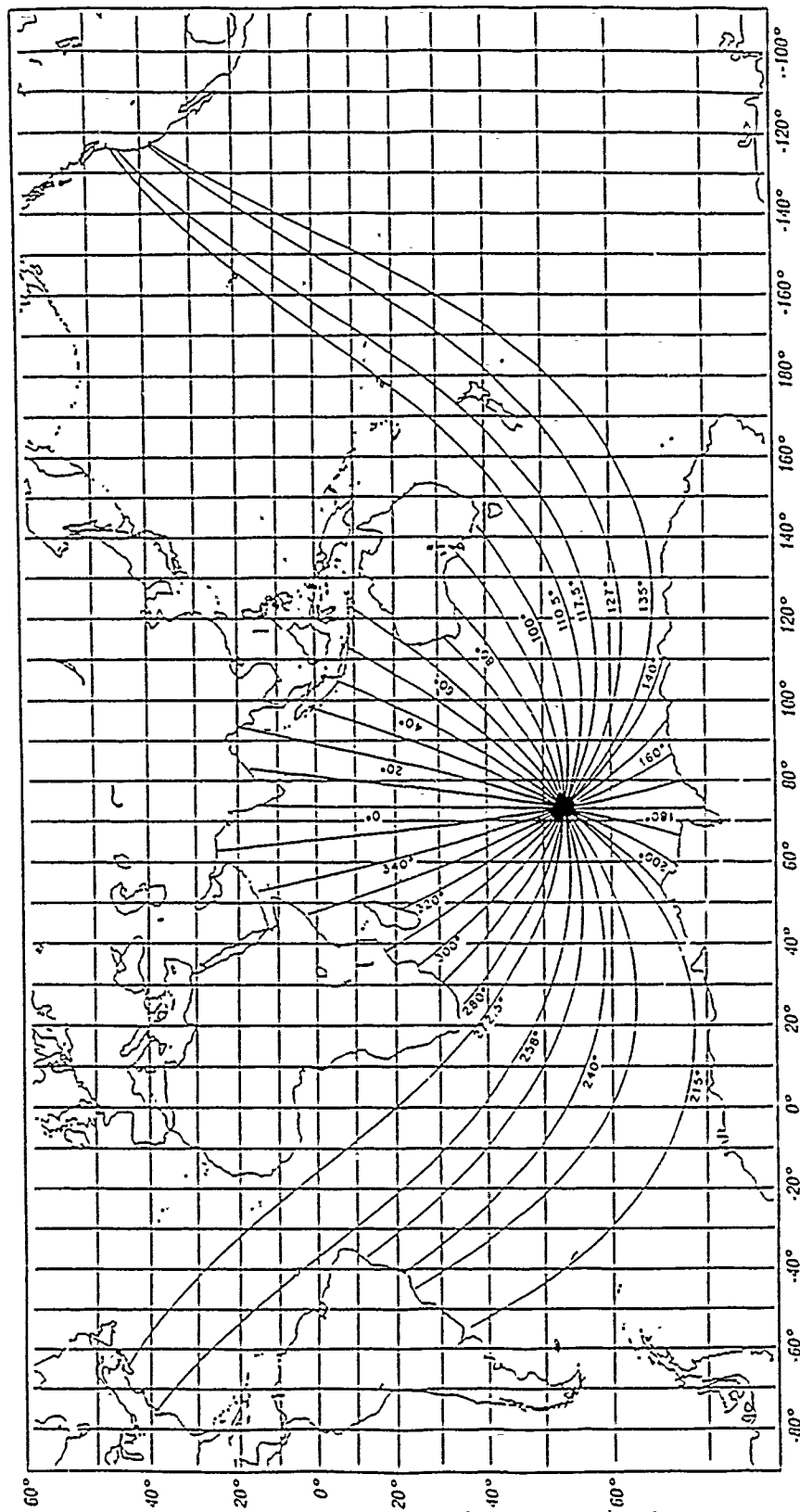


Figure 1.1: Heard Island Raypaths.

II. ACOUSTIC WAVE PROPAGATION

A. THE WAVE EQUATIONS

Ocean acoustic signals frequently travel from source to receiver along multiple paths. These multiple "raypaths" are formed as a direct result of the nonhomogeneous structure of the ocean. However, understanding how waves propagate in the ocean can be facilitated by first treating it as a homogeneous medium and developing equations for wave propagation, and next, by the introduction of Snell's Law and some basic rules for determining the raypaths in a nonhomogeneous medium.

Sound travels as a result of the displacement of particles at some source, which causes a local change in pressure. The resulting pressure change then propagates away from the source in a spherical manner as a pressure wave with velocity c . At distances sufficiently far from the source the wave can be treated as planar over a finite region.

1. Plane Wave Equation

Assuming the propagation direction to be in the x direction with the wave front formed in the y - z plane, the pressure change and particle speed can be related by the equation [Refs. 6,7,8]

$$\frac{\partial p(x,t)}{\partial x} = -\rho \frac{\partial u(x,t)}{\partial t} \quad (2.1)$$

where $p(x,t)$ is pressure, $u(x,t)$ is particle speed, and ρ is the density of water. The rate of change of $u(x,t)$ with respect to x is given by [Ref. 6]

$$\frac{\partial u(x,t)}{\partial x} = -\frac{1}{B} \frac{\partial p(x,t)}{\partial t}, \quad (2.2)$$

where B is the bulk modulus of elasticity. Combining Eq. (2.1) and Eq. (2.2) gives

$$\frac{\partial^2 p(x,t)}{\partial t^2} = \frac{B}{\rho} \frac{\partial^2 p(x,t)}{\partial x^2}. \quad (2.3)$$

The sound speed in water, c , is related to B and ρ and is given by

$$c^2 = \frac{B}{\rho}. \quad (2.4)$$

Substituting into Eq. (2.3) gives the acoustics plane wave equation [Refs. 6,7]

$$\frac{\partial^2 p(x,t)}{\partial t^2} = c^2 \frac{\partial^2 p(x,t)}{\partial x^2}. \quad (2.5)$$

2. Spherical Wave Equation

The next step in the process is to work backward from Eq. (2.5) and find an expression for the traveling wave in spherical coordinates. First, assume a point source with the pressure wave expanding radially. Instead of the single dimension x , Eq. (2.5) is three dimensional. Changing coordinate systems from linear to Cartesian and taking the Laplacian in place of the partial derivatives with respect to position Eq. (2.5) takes the form [Refs. 6,7]

$$\frac{\partial^2 p(x,y,z,t)}{\partial t^2} = c^2 \nabla^2 p(x,y,z,t). \quad (2.6)$$

Assuming no losses, a uniform traveling wave is simply a function of distance r from the source and time, and is independent of angular displacement

from the source. Therefore, it is logical to convert to a spherical coordinate system and use the spherical form of the Laplacian operator in Eq. (2.6). After some simplification [Ref. 6], Eq. (2.6) becomes

$$\frac{\partial^2 (rp(r,t))}{\partial t^2} = c^2 \frac{\partial^2 (rp(r,t))}{\partial r^2}, \quad (2.7)$$

where r is the radial distance from the source. This is the spherical form of the wave equation and is a function of only the distance from the origin, and time. It is the same as Eq. (2.5) with $p(x,t)$ replaced by $rp(r,t)$.

Given a complex sinusoidal pressure function, a solution to Eq. (2.7) is given by

$$p(r,t) = \frac{p_m}{r} e^{j\omega(t-r/c)}, \quad (2.8)$$

where p_m is the peak pressure amplitude at unit distance, and c is the nominal sound speed [Ref. 6].

B. RAYPATH DETERMINATION

There are two fundamental reasons that an acoustic signal from a single source may have multiple raypaths and corresponding arrival times at a single receiver. First, the ocean is a nonuniform medium. It varies in depth (pressure), salinity, and temperature. There are numerous currents, eddies and tidal effects. Second, it is limited vertically by a sea-air interface and a sea-bottom interface. Because of the first condition, sound speed is also nonuniform and is a function of salinity, temperature, and pressure. Fortunately, a great deal is known about the dependencies of sound speed in water and the behavior of sound waves at the sea-air and sea-bottom interfaces. Applying this knowledge, and Snell's Law, raypaths can be accurately determined.

2. Sound Speed

Sound speed formulation as a function of temperature, salinity, and pressure has been determined numerically, and is approximately [Ref. 7]

$$c = 1449.2 + 4.6T - 0.055T^2 + 0.00029T^3 + (1.34 - 0.01T)(S-35) + 1.58 \times 10^{-6}P, \quad (2.9)$$

where c is sound speed, T is temperature, P is gauge pressure of a column of water, and S is salinity. For most applications salinity variations are small and can be neglected. Pressure is a linear function of depth. Temperature gradients and layers are found throughout the world's oceans and are in general a function of depth. Warmer water tends to reside near the surface with colder temperatures at deeper depths. A certain amount of mixing can also occur to form isothermal and isosaline water. Temperature layers can also be formed due to currents and eddies. Sound waves can be thought of as tending to bend toward points of lower sound speed, as will become apparent in the next section on Snell's Law.

2. Snell's Law

A raypath through the ocean medium can be determined by considering the sound speed versus depth to be a continuous function, implying a continuously stratified medium. Changes in propagation path can be computed by treating the entire stratified medium as a set of n layers, and determining the refraction from medium 1 to medium 2, then 2 to 3, and so on to medium n . This is done through the use of Snell's Law, which is derived in several texts [Refs. 6,7]. Stated here, a sound wave (ray) will be refracted from medium to medium according to the relation

$$\frac{\sin(\theta_1)}{c_1} = \frac{\sin(\theta_2)}{c_2} = \dots = \frac{\sin(\theta_n)}{c_n}, \quad (2.10)$$

where θ_1 is the incident angle, measured from the ray to the vertical, and $\theta_2, \dots, \theta_n$ are refraction angles. The constants $c_1, c_2, \dots, c_n$ are the sound speeds in medium 1, medium 2, and medium n , respectively. It is assumed that the energy in the ray is constant through the boundary.

As an example, consider the case where $c_1 > c_2$. In this case, the sound speed in medium 1 is faster than that of medium 2. Rearranging Eq. (2.10) in the following manner

$$\frac{c_2 \sin(\theta_1)}{c_1} = \sin(\theta_2), \quad (2.11)$$

it can be seen that ratio of c_2 to c_1 is less than one. For the equality to hold, θ_2 must be a smaller angle than that of the incident angle, θ_1 . This means the ray is "bending" downward towards the medium with the lower sound speed. Similarly the ray bends upwards for the case when $c_2 > c_1$. In other words, sound waves bend toward the region of slower sound speed.

In reality, when sound impinges on a boundary between two mediums the sound waves are both reflected and transmitted. Application of Snell's Law remains the same for the transmitted wave, and the reflected wave has a reflection angle that is equal to the incident angle. The transmitted angle remains a function of the sound speeds of the mediums on either side of the boundary and is given by Eq. (2.10). However, the amplitude of the transmitted wave does not equal that of the incident wave, because conservation of energy must apply. Therefore, the amplitude of the reflected and transmitted waves must sum to that of the incident wave at the boundary. Assuming the density of medium 1 and 2

to be the same, the ratio of the amplitude of the reflected wave to that of the incident wave is given by [Ref. 6]

$$\frac{R}{I} = \frac{c_2 \sin(\theta_1) - c_1 \sin(\theta_2)}{c_2 \sin(\theta_1) + c_1 \sin(\theta_2)}, \quad (2.12)$$

where R is the reflected amplitude, I is the incident amplitude, θ_1 is the incident angle, θ_2 is the transmitted angle, and c_1 , and c_2 are the sound speeds in medium 1 and 2, respectively. Similarly, the ratio of the transmitted to incident amplitude is given by

$$\frac{T}{I} = \frac{2 c_2 \sin(\theta_1)}{c_2 \sin(\theta_1) + c_1 \sin(\theta_2)}, \quad (2.13)$$

where T is the transmitted amplitude. For the case when the mediums do not have the same density, the sound speeds c_i in Eq. (2.12) and Eq. (2.13) are simply replaced by

$$Z_i = \rho_i c_i, \quad (2.14)$$

where ρ_i , and Z_i are density and impedance of the i th medium, respectively.

For rays incident at the sea surface the sound wave is considered to be totally reflected, and the amount of reflection at the ocean bottom is based on the local bottom consistency. There are two special conditions that can be applied with Snell's Law. In the first case, when $c_1 > c_2$ and the ratio of c_1 to c_2 becomes very large, the transmission angle θ_2 must approach zero. In this case, the wave transmits perpendicular to the boundary. In the second case, when $c_2 > c_1$, there is a condition when the transmitted angle must be negative to satisfy Eq. (2.10). However, θ_2 cannot be negative. This is the condition for total reflection and the reflected angle equals the incident angle, and θ_2 is zero.

3. Travel Time

Once the sound speed as a function of depth has been determined, the ray path can be determined. Travel time can then be evaluated as an integral over the raypath. Travel time, T_i , for a particular ray i from point a to b is [Ref. 9]

$$T_i = \int_a^b \frac{ds}{c(x,y,z)}, \quad (2.15)$$

where sound speed is assumed to be independent of time t . If the assumption is made that

$$c(x,y,z) = c_0(x,y,z) + \delta c(x,y,z), \quad (2.16)$$

where $c_0(x,y,z)$ is an assumed background sound speed field, and $\delta c(x,y,z)$ is an unknown sound speed field, Eq. (2.15) can then be written as

$$T_i = T_{0i} + \delta T_i \cong \int_a^b \frac{ds}{c_0(x,y,z)} - \int_a^b \frac{\delta c(x,y,z) ds}{c_0^2(x,y,z)}, \quad (2.17)$$

for $\delta c \ll c_0$. The travel time T_i has been broken down into a travel time T_{0i} due solely to the assumed sound speed field c_0 , and δT_i , a linear function of the unknown sound speed field perturbation $\delta c(x,y,z)$. Variation in the arrival time T_i is a perturbation of the arrival time due to perturbations in the sound speed along the path. By measuring these variations and using linear inverse mathematical techniques it is possible to determine some desired characteristic of this geophysical problem [Refs. 2,9].

III. M-SEQUENCES AND HADAMARD PROCESSING

A. INTRODUCTION

An important parameter in underwater acoustic tomography is the arrival time of a transmitted signal. A very useful means for measuring arrival time is to apply an impulsive excitation to the system, in this case the ocean environment. An impulse can be generated by explosive or implosive sources cheaply and easily, but suffer from unevenness in the frequency spectrum, and repeatability. Another method involves the use of pseudorandom noise. Pseudorandom noise is deterministic and is formed by a sequence of ones and zeros known as a maximal-length sequence or m-sequence [Refs. 10,11]. M-sequences are typically transmitted by using the ones and zeros of the sequence ($\{1,0\}$ mapped to $[-1,1]$) to phase encode a carrier signal [Refs. 12,13,14].

M-sequences are generated using a simple binary shift register that is formed according to the desired primitive polynomial. Since the signal being transmitted is known, a simple autocorrelation can be formed using a bank of matched filters. The resulting correlation is impulse-like, and has a shorter duration than the originally transmitted signal and has a much larger amplitude, which makes measurement of arrival time an easier task.

The following sections give a brief description of shift register fundamentals, m-sequences, and fast Hadamard processing of m-sequences.

B. SHIFT REGISTER FUNDAMENTALS

A general sequence shift register generator is shown in Figure 3.1. A sequence of ones and zeros can be output from any one of the registers starting at any register state, except zero. In what follows all output sequences are taken from the 0th register. The sequence of bits generated is periodic and will repeat with some period L , based on the structure of the generator, as described by a polynomial such as

$$g(D) = b_n D^n + b_{n-1} D^{n-1} + \dots + b_1 D + b_0, \quad (3.1)$$

where D is the unit delay and b_k are the feedback weighting coefficients. The weighting coefficients take on values of one or zero indicating connection or no connection to the k^{th} register. All arithmetic operations for polynomials are performed using finite-field arithmetic (modulo two) and are described in more detail in Reference 10.

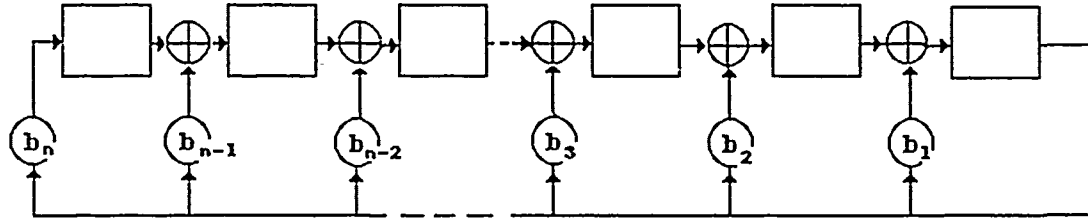


Figure 3.1: General sequence shift register generator for Eq. (3.1).

An example of a third order sequence shift register generator is shown in Figure 3.2, and its corresponding generating polynomial is

$$g(D) = D^3 + D + 1. \quad (3.2)$$

In this case Eq. (3.2) happens to be a primitive polynomial, where a primitive polynomial is any polynomial that will not repeat its register state until after $2^n - 1$ delays, where n is the number of delays in the shift register generator.

Therefore, a shift register that is defined by a primitive polynomial will generate a sequence that is of maximal-length and is known as a maximal-length sequence or m-sequence.

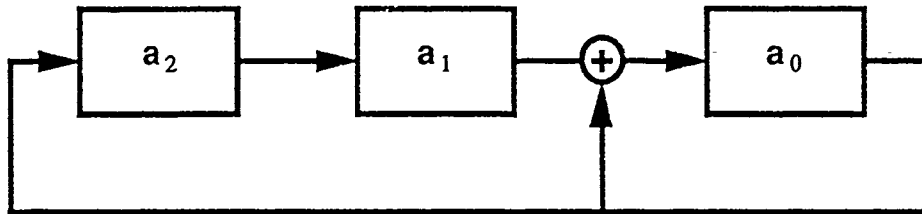


Figure 3.2: Shift register realization of Eq. (3.2).

Continuing with this example, the sequential output from the registers of Figure 3.2 is then given as in Figure 3.3, where the initial register contents is arbitrarily set to $a_2=1$, $a_1=0$, $a_0=0$ [Ref. 9].

The corresponding m-sequence is then obtained by taking any one column from the register states top to bottom (or bottom to top). Note, that the combination of [000] never occurs. This is because an initial value of zero will not allow any transition in state and can be easily verified by inspection.

Cycle	a_2	a_1	a_0
1	1	0	0
2	0	1	0
3	0	0	1
4	1	0	1
5	1	1	1
6	1	1	0
7	0	1	1
8	1	0	0

Figure 3.3: Shift register contents when generating a third order m-sequence. The eighth cycle shows that the register begins to repeat.

The characteristics of the m-sequence is unchanged whether it is transmitted in the forward or reverse direction, but when performing the fast Hadamard Transform, reviewed in the next section, the direction in which the sequence is transmitted is significant. Therefore, the top to bottom sequence will be designated the "forward" code and the bottom to top the "reverse" code,

$$\text{forward } m = 0011101$$

$$\text{reverse } m = 1011100. \quad (3.3)$$

M-sequences have several desirable characteristics. Its correlation function is triangular in shape, see Figure 3.4, and short in duration. It is deterministic, once the polynomial is determined all output is known. It can be implemented easily and is periodic. However, one major drawback is that its autocorrelation requires N multiplies, and since the arrival time is never known, the input signal must be correlated with all N shifted versions of the original sequence which

requires N^2 multiplies. This can be overcome by use of the fast Hadamard transform (FHT), discussed in the next section.

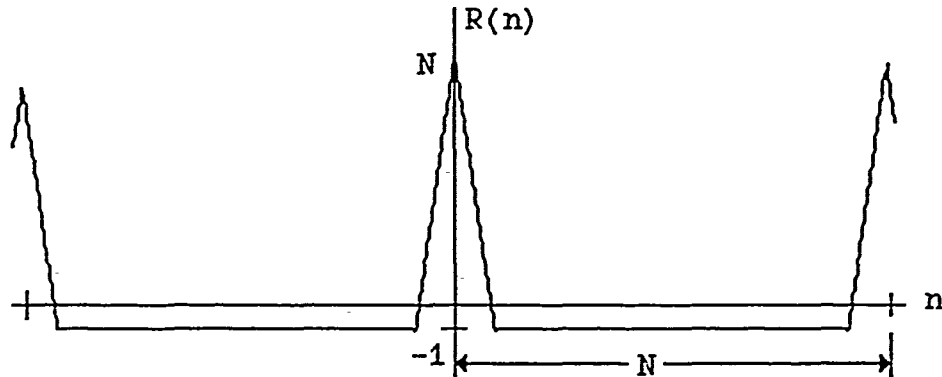


Figure 3.4: Autocorrelation function for a m-sequence of length N .

C. THE FAST HADAMARD TRANSFORM

The autocorrelation of the input data sequence with all possible shifted versions of the original m-sequence can be written as

$$MD = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \end{bmatrix}, \quad (3.4)$$

where M is a matrix whose rows are the shifted versions of the forward code and D is the input data vector. This product requires N^2 multiplications. To reduce this number, a Hadamard transform can be used. The third order Hadamard matrix is given by

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1-1 & 1-1 & 1-1 & 1-1 & 1-1 & 1-1 & 1-1 & 1-1 \\ 1 & 1-1-1 & 1 & 1-1-1 & 1 & 1-1-1 & 1 & 1-1-1 \\ 1-1-1 & 1 & 1-1-1 & 1 & 1-1-1 & 1 & 1-1-1 & 1 \\ 1 & 1 & 1 & 1-1-1-1-1 & 1 & 1-1-1-1-1 & 1 & 1-1-1-1-1 \\ 1-1 & 1-1-1 & 1-1 & 1-1 & 1-1 & 1-1 & 1-1 & 1-1 \\ 1 & 1-1-1-1-1-1 & 1 & 1-1-1-1-1-1 & 1 & 1-1-1-1-1-1 & 1 & 1-1-1-1-1-1 \\ 1-1-1 & 1-1 & 1-1 & 1-1 & 1-1 & 1-1 & 1-1 & 1-1 \end{bmatrix}, \quad (3.5)$$

where H can be formed recursively from

$$H_1 = [1], \quad H_{i+1} = \begin{bmatrix} H_i & H_i \\ H_i & -H_i \end{bmatrix}, \quad (3.6)$$

and by performing a simple mapping of (1,-1) to (1,0). H can be written in a form that is easier for most people to work with [Refs. 15,16,17],

$$H = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}. \quad (3.7)$$

It can also be shown [Refs. 16,17] that the matrix H can be factored into the product of two matrices consisting of a binary count, in this case from 0 to 7,

$$H = AA^T = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}. \quad (3.8)$$

The matrix M, Eq. (3.4), can also be factored into two matrices, L and S, given by [Refs. 16,17]

$$LS = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}. \quad (3.9)$$

It is easily verified that

$$M = LS. \quad (3.10)$$

Note that by adding a leading column of zeros to S , forming S' , and a leading row of zeros to L , forming L' . All possible combinations of ones and zeros are formed in L' , S' , as in A , with the differences between the matrices being the order in which they occur. It is straight forward to find a matrix P such that $S' = A^T P$ and another matrix U such that $L' = UA$. Combining these results maps M' to the Hadamard matrix as [Ref. 16,17]

$$M' = L'S' = UAA^T P = UHP, \quad (3.11)$$

where M' is the M matrix with an appended column and row of zeros in the first row and column. Recall that the correlation was performed by forming the product of the M matrix and the data vector D . By replacing M with M' and forming a new data vector

$$D' = \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{bmatrix}, \quad (3.12)$$

and after substituting Eq. (3.11) for M' the correlation becomes

$$R' = M'D' = UHPD'. \quad (3.13)$$

From this it can be seen that the matrix M is not needed and the Hadamard matrix can be used in its place. By using the Hadamard matrix in the form of Eq. (3.5) and forming a product with D' , the result becomes a sum of the individual terms of D' with + and - weights and has the form

$$H D' = \begin{bmatrix} a+b+c+d+e+f+g+h \\ a-b+c-d+e-f+g-h \\ a+b-c-d+e+f-g-h \\ a-b-c+d+e-g-g+h \\ a+b+c+d-e-f-g-h \\ a-b+c-d-e+f-g+h \\ a+b-c-d-e-f+g+h \\ a-b-c+d-e+f+g-h \end{bmatrix}, \quad (3.14)$$

which when written in the form of a flow graph having the same form as the well known fast Fourier transform shown in Figure 3.5, where the complex multiplies are set to one [Refs. 9,16,17]. This is known as the fast Hadamard transform (FHT).

All that is left is to determine the permutation matrices P and U . By looking at how P and U must be formed, it can be seen that P must have ones at the indices [Ref. 15,16]

ROW	0	1	2	3	4	5	6	7
COLUMN	0	2	1	4	6	7	3	5,

and U must have ones at the indices

ROW	0	4	2	1	6	3	7	5
COLUMN	0	1	2	3	4	5	6	7,

which correspond to the binary values of S' and L' . It turns out that the matrices S' and L' provide all of the necessary information. All that is needed to perform the desired multiplication is to permute the input data vector according to S' , form the FHT, and permute again according to L' . L and S are generated from

the primitive polynomial that defines the m-sequence using the generators shown in Figure 3.6 for the forward code and Figure 3.7 for the reverse code [Ref. 15]. The modification to form S' and L' simply involves adding leading zeros to S and L. Since the correlation is always over 2^n-1 data points, a leading zero is added to the input data vector D forming D' adding no new data. After processing, D' holds the correlation function with the zero position containing the average level, which may be used to remove the bias in the autocorrelation function [Ref. 11].

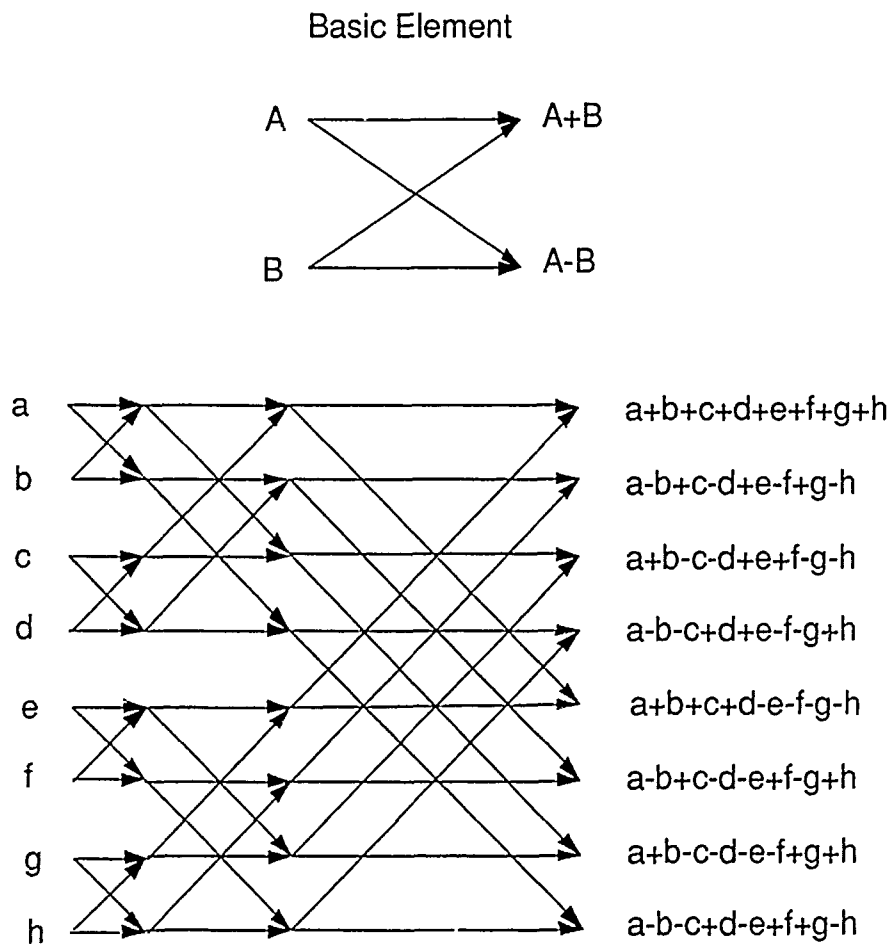
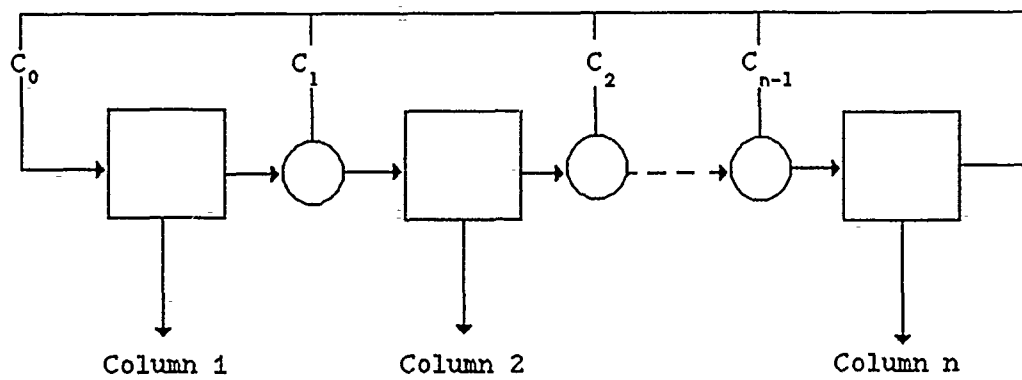
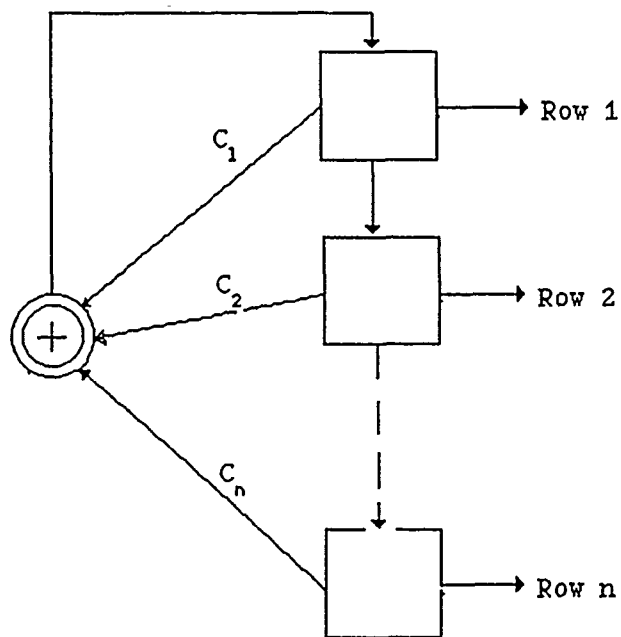


Figure 3.5: Basic fast Hadamard transform element for cascading additions and the full diagram for an eight point FHT.

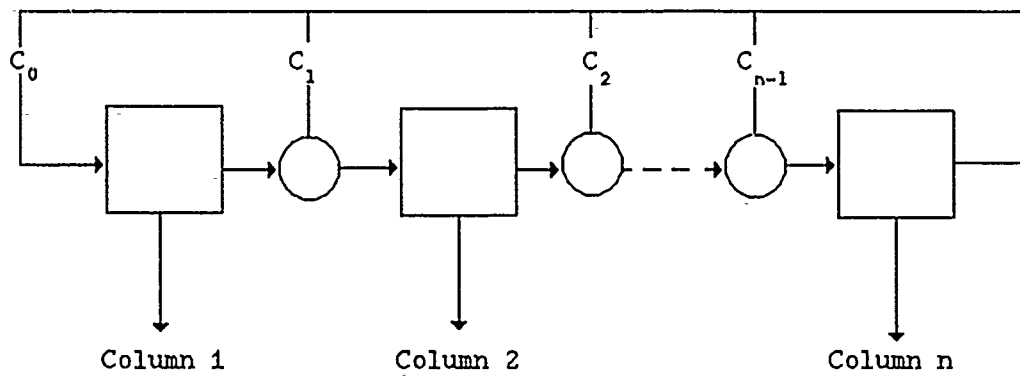


(a)

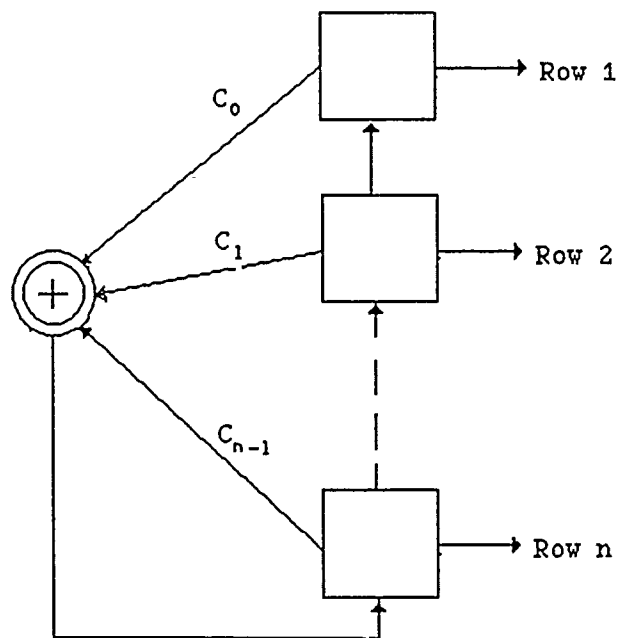


(b)

Figure 3.6: (a) L generator and (b) S generator for the forward code.



(a)



(b)

Figure 3.7: (a) L generator and (b) S generator for the reverse code.

IV. TIME-VARYING DOPPLER PROCESSING

A. PHASE ENCODING OF M-SEQUENCES

A simple method in which m-sequences can be transmitted is to phase encode the digits using a mapping of (0,1) to (1,-1) and appropriately selecting a phase angle which will maximize the signal-to-noise performance. The phase angle that optimizes this is

$$\psi = \tan^{-1}(\sqrt{N}), \quad (4.1)$$

where ψ is the phase modulation angle, and N is the length of the sequence [Ref. 11]. However, for the Heard Island Experiment, $\psi = 45$ degrees is used to simplify SNR estimation. The transmitted signal $s(t)$ takes the form

$$s(t) = A \cos(2\pi f_c t + M(t)\psi), \quad (4.2)$$

where A is the amplitude, f_c is the carrier frequency, and $M(t)$ takes on values of +1 or -1 depending on the m-sequence code. The minimum length of time $M(t)$ remains constant is the digit duration d .

It is normal practice to choose an integer number of cycles, Q , of the carrier frequency per digit.

B. SIGNAL PROCESSING WITH ZERO DOPPLER

At the receiver the signal is normally sampled at an integer multiple of the carrier frequency f_c , i.e. with $f_s = mf_c$, where m is chosen to be an integer greater than two, to ensure that the Nyquist criterion is satisfied, and f_s is the sampling frequency. Sampling in this fashion simplifies processing and interleaving of data, since there is an integer number of data points per digit d . The received signal is $r(t) = s(t - T_1)$, assuming no attenuation, no dispersion, and

a single raypath. The sampled input $r(n)$ is then demodulated to remove the carrier from the signal. The received signal $r(n)$ will arrive at some unknown arrival time T_i , and must be multiplied by both the sine and cosine functions to form the in-phase and quadrature components $p(n)$ and $q(n)$, respectively [Ref. 18]. For $p(n)$ this then gives

$$p(n) = A \cos\left(\frac{2\pi f_c n}{f_s} + M\left(\frac{n}{f_s} - T_i\right)\psi\right) \cos\left(\frac{2\pi f_c n}{f_s}\right)$$

$$p(n) = \frac{A}{2} \left[\cos\left(M\left(\frac{n}{f_s} - T_i\right)\psi\right) + \cos\left(\frac{4\pi f_c n}{f_s} + M\left(\frac{n}{f_s} - T_i\right)\psi\right) \right], \quad (4.3)$$

and similarly for $q(n)$

$$q(n) = \frac{A}{2} \left[\sin\left(M\left(\frac{n}{f_s} - T_i\right)\psi\right) - \sin\left(\frac{4\pi f_c n}{f_s} + M\left(\frac{n}{f_s} - T_i\right)\psi\right) \right]. \quad (4.4)$$

The high frequency components are then removed by lowpass filtering with the cutoff frequency greater than the digit bandwidth. The resulting waveforms are

$$p(n) = \frac{A}{2} \cos\left(M\left(\frac{n}{f_s} - T_i\right)\psi\right)$$

$$q(n) = \frac{A}{2} \sin\left(M\left(\frac{n}{f_s} - T_i\right)\psi\right), \quad (4.5)$$

which are constant over the length of the digit of $M(n)$.

Because of the lower digit bandwidth, decimation in time may be performed without any loss of information [Ref. 19]. This significantly reduces the data rate and storage requirements of the processed data. Further processing follows the scheme of the previous chapter. The input data vector is permuted, the FHT is performed, and permuted again to the correct order. Multiple samples per digit must be accounted for by interleaving when performing the FHT on a given sequence length. Finally, magnitude and phase are computed in the normal fashion.

The autocorrelation function, see Figure 3.4, for m-sequences has a negative DC level for all digit positions except the first. A correction may be made which will adjust this level to zero. Knowing the phase modulation angle, the DC bias can be removed by adding the correction

$$C_{\text{cor}} = f_1 L_{\text{avg}}, \quad (4.6)$$

where C_{cor} is the complex level adjustment, L_{avg} is the average DC level from the FHT, and f_1 is given by the relation [Ref. 11]

$$f_1 = j \frac{(N+1) \tan(\psi) + j(N - \tan^2(\psi))}{N^2 + \tan^2(\psi)}, \quad (4.7)$$

where N is the length of the m-sequence and ψ is the phase modulation angle. Since all parameters of Eq. (4.7) are known, f_1 is computed once at system initialization and C_{cor} requires one complex multiplication for each data length processed.

A block diagram showing the process is given in Figure 4.1. Note that in this case a Butterworth filter of order ten was used in the passband and a Chebychev filter of order five was used for the lowpass filter. Any filter type can logically be substituted.

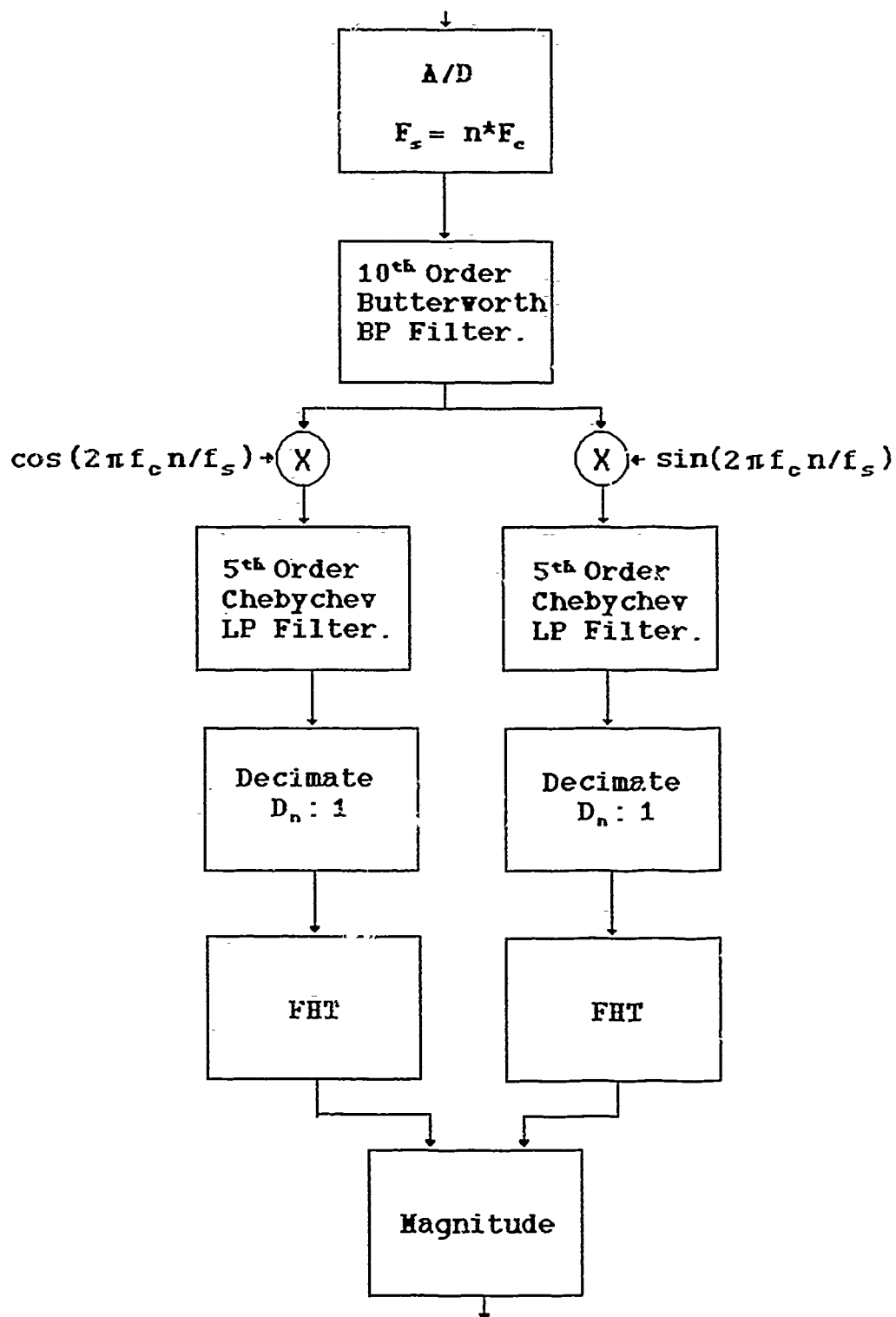


Figure 4.1: Block diagram of signal processing with zero Doppler.

C. SIGNAL PROCESSING WITH DOPPLER

Generally in acoustic tomography the receiving or transmitting ship must maintain way and/or buoys are acted on by the forces of currents, tides, wind etc., all of which cause relative motion between the receiver and transmitter. The Doppler shift that is then imparted on the signal must be corrected for at the receiver. For the relatively slow and uniform speeds encountered in acoustic tomography it is sufficient to compensate for first order Doppler, and neglect acceleration effects [Refs. 20,21].

When Doppler is present a transmitted signal $s(t)$ may be received as

$$r(t) = s[(1 + \frac{v}{c}) t - T_i], \quad (4.8)$$

where v is the velocity in meters/second, c is the nominal sound speed in water in meters/second, and T_i is the delay from transmitter to receiver as in section B, where attenuation, dispersion, and a single raypath are assumed. The magnitude of the spectrum of $r(t)$ can be given as

$$|R(f)| = \left| S\left[\frac{f}{(1 + v/c)}\right] \right|. \quad (4.9)$$

Sampling at an integer multiple of the carrier frequency assuming zero Doppler and replacing the time dependency with the discrete index n gives

$$r(n) = s\left(\frac{n}{f_s} - T_i\right). \quad (4.10)$$

With Doppler, the received signal is then given by

$$r(n) = s\left[\left(\frac{n(1 + v/c)}{f_s}\right) - T_i\right]. \quad (4.11)$$

Adjusting the sampling frequency by the same Doppler shift gives a new sampling frequency of

$$f'_s = f_s(1 + v/c), \quad (4.12)$$

which forms a new received signal

$$r'(n) = s\left[\left(\frac{n(1 + v/c)}{f'_s}\right) - T_i\right] \quad (4.13)$$

Substituting Eq. (4.12) into Eq. (4.13) yields

$$r'(n) = s\left(\frac{n}{f_s} - T_i\right) = r(n). \quad (4.14)$$

At this point the results are identical and processing of the data can proceed as in the zero Doppler case [Ref. 20].

Two methods can be employed to perform the adjustment to the sampling frequency as discussed. The first is to use a bank of sampling devices that sample at a set of sampling frequencies that cover the desired Doppler range such that each device samples at frequencies $f_{s1}, f_{s2}, \dots, f_{sk}$. This method requires a great deal of hardware and is inflexible and expensive to implement. The second method is to frequency shift the received signal and interpolate between samples at the new sampling frequency to predict the value of the signal as if the desired sampling frequency had been used [Refs. 20,21].

Doppler resolution is $1/T_A$ Hz, where T_A is the analysis interval [Ref. 20]. For example, a $N=255$ length m-sequence with $Q=5$ and $f_c=57$ Hz giving a T_A of 22.368 seconds and a resolution of 0.04471 Hz, which from Eq. (4.12) corresponds to a Doppler shift of + or - 2.3 knots. Figure 4.2 is an ambiguity plot derived from this example. The plot shows signal arrival time for the given

m-sequence versus signal Doppler over a + or - 5 knot range. As can be seen in the plot signal strength decreases as the carrier frequency moves away from the zero Doppler case and is not detectable at about 2 knots on either side of center.

Frequency shifting in the frequency domain is equivalent to multiplying the samples or the demodulates in the time domain by $\exp(j2\pi n f_d / f_s)$ [Ref. 20], where f_d is the Doppler shift of the carrier signal in Hertz. The Doppler shift f_d is determined directly from the expected Doppler and is given by

$$f_d = \frac{v}{c} f_c. \quad (4.15)$$

For a sampling frequency that is four times the carrier the multiplication exponential can then be related to the Doppler speed, v (meters/second), and the multiplication of the demodulates in the time domain is by $\exp(j\pi n v / 2c)$. The

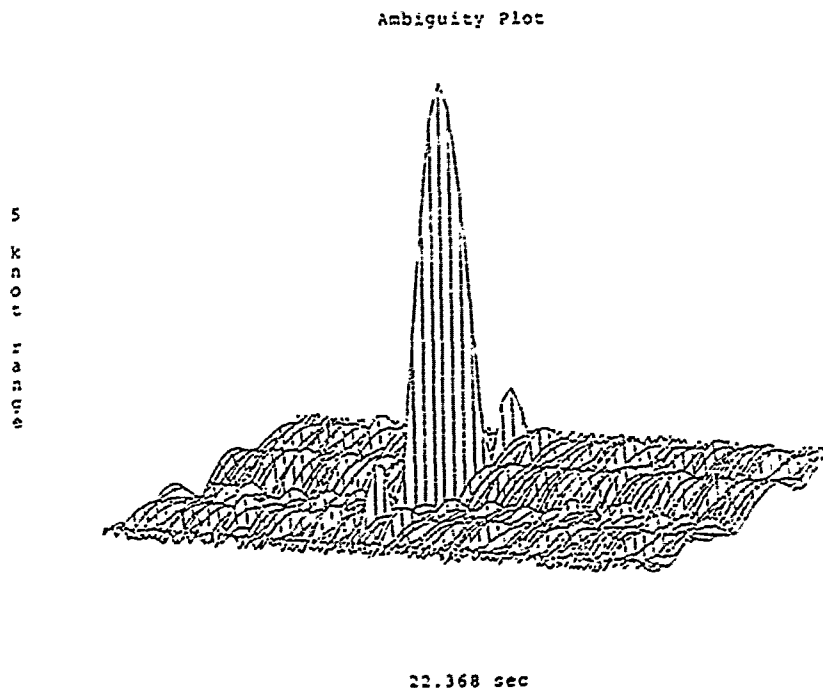


Figure 4.2: Ambiguity plot of Arrival Time vs. Doppler for a theoretical sequence with $N=255$, $Q=5$, and $f_c=57$ Hz.

data is then interpolated at the new sampling frequency corresponding to the original sampling frequency shifted by the same amount of Doppler as given by Eq. (4.12).

The Doppler processing scheme in this thesis is performed as follows. Frequency shifting is performed by combining the shift with the demodulation process. This reduces the number of complex multiplies by combining the modulation angle and the frequency shift via simple addition, see Figure 4.3. The resampled signal is then linearly interpolated [Ref. 21]. Since the Doppler shift is unknown, except within some range, each data set is processed over the entire Doppler range, where the Doppler step size is set in knots. Frequency shifting and then interpolating results in the following expression for the resampled signal [Ref. 21]

$$s(t_a+h) = s(t_a)e^{j2\pi f_d t_a} + \frac{h}{t_b-t_a} [s(t_b)e^{j2\pi f_d t_b} - s(t_a)e^{j2\pi f_d t_a}], \quad (4.16)$$

where $t_a < t_a+h < t_b$ and time replaces the discrete index n for clarity. The times t_a and t_b correspond to sampled data points, and h is the time spacing from t_a to the point to be interpolated. The position h , varies from interpolation point to interpolation point, according to the sampling period $1/f_s'$, as time progresses. Other techniques for interpolating between samples are available but are slower and more complex and will not be discussed here.

At this point processing continues as in the zero Doppler case. Figure 4.3 shows the process in block diagram form with an additional block for performing coherent averaging to increase signal processing gain, if required or desired [Ref. 18].

Figure 4.4 shows the Doppler resolution for a transmitted signal with -1.4 knots of Doppler, using the parameters given in the above example without

noise, and was processed by this scheme. Note that these results are plotted on a different scaling than that of Figure 4.2 but show the same results. The Doppler resolution is again approximately + or - 2 knots as expected.

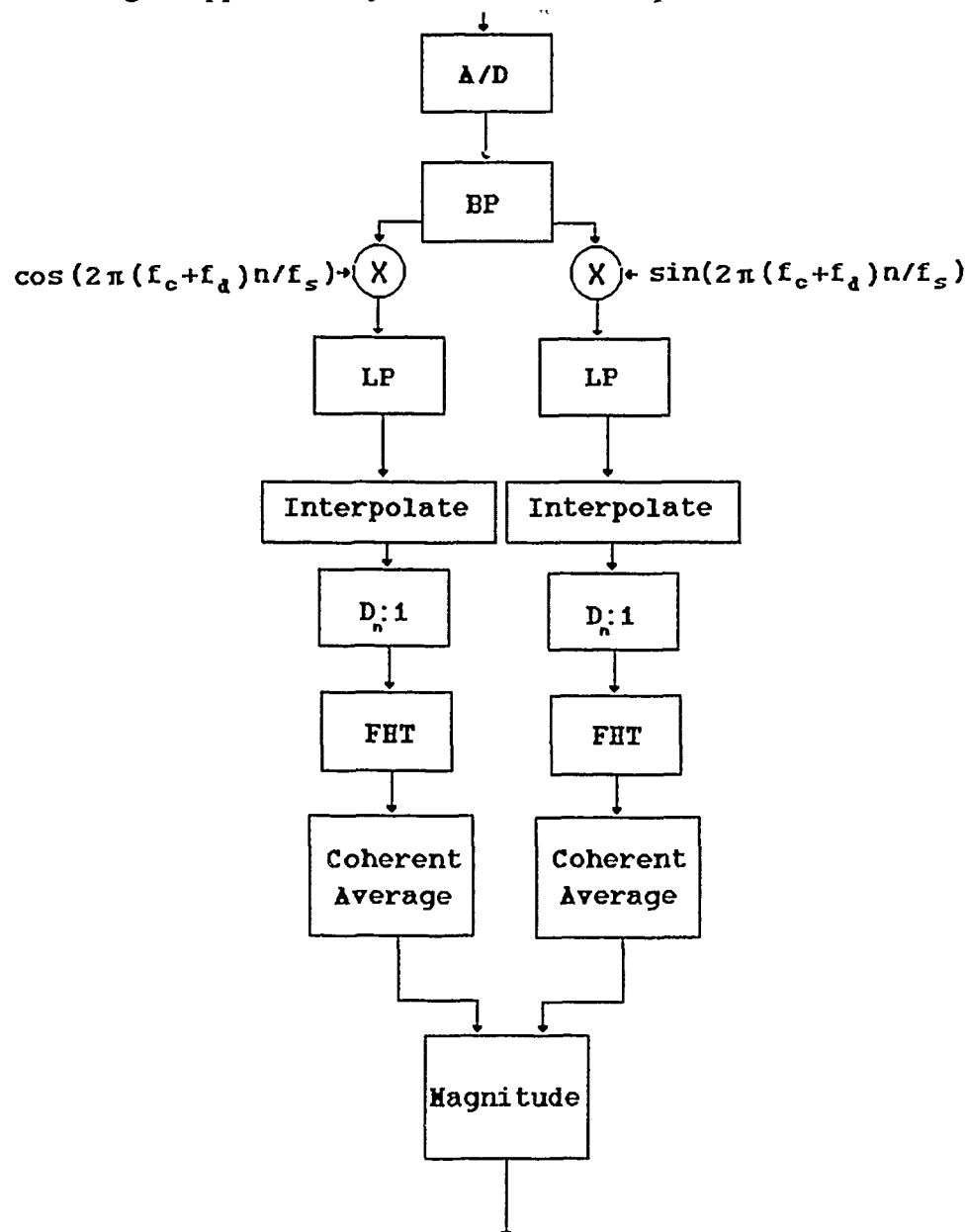


Figure 4.3: Block diagram of signal processing with Doppler and coherent averaging incorporated.

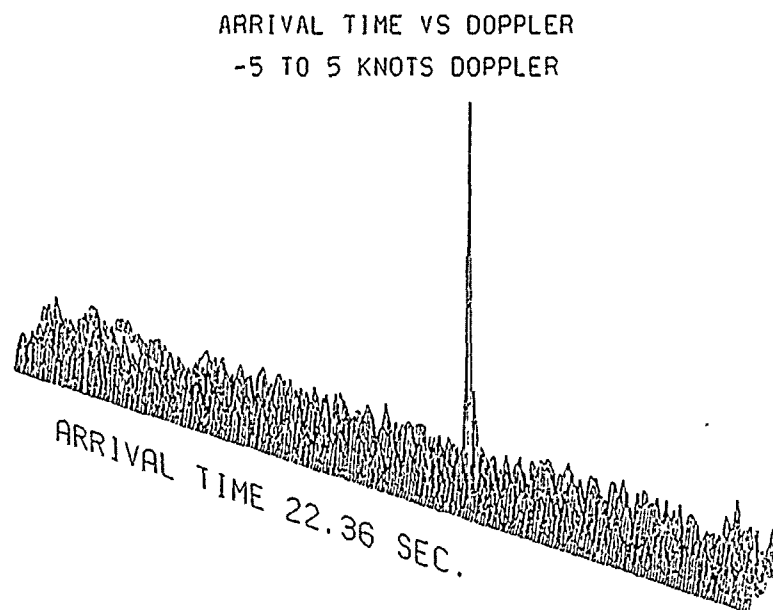


Figure 4.4: Ambiguity plot for Arrival Time vs. Doppler for sequence length $N=255$, $Q=5$, and $f_c=57$ Hz, processed by SEQREM program with no signal noise.

V. RESULTS AND CONCLUSIONS

A. RESULTS

The objective of this thesis was to develop a program in the C programming language that can process any m-sequence at any carrier frequency over any acceptable Doppler range. This objective has been fully met. The package SEQREM has been developed which will scan over any specified Doppler range and specified Doppler step size. In cases where no Doppler is expected the program skips the Doppler scanning procedure, reducing run time. Programming allows for any carrier frequency, any type filter of any order, up to twenty, and any size m-sequence, with dynamic memory space allocation for array manipulation, limited only by the available computer memory. It permits coherent averaging of sequence segments as specified at initialization, improving processing gain, and computes and removes the DC bias in the autocorrelation function. Decimation in time is performed which significantly reduces off-line storage requirements for processed data.

Appendix A contains a complete set of processing results for all three cases, closing, opening, and zero Doppler. In all of the plots a 255 digit sequence is used with $Q = 5$, and a carrier of 57 Hz, corresponding to one of the signals to be transmitted in the Heard Island Experiment. The sampling frequency is four times the carrier frequency or 228 Hz. The simulated data assumes white Gaussian noise. The plots are labeled with the Doppler that was processed and with the sequence period, adjusted for the expansion or compression, according

to the Doppler bin. Decimation in time was performed at a ratio of ten to one. Peaks correspond to the center of detection of the signal arrival within the period. The leading edge of the peak is the actual arrival time and can be extracted by an edge detection program. Sequential frames are aligned one behind the other and represent increasing time along the y direction. All plots are in magnitude. Phase plots are omitted for brevity. Some specific results are contained here.

Figure 5.1 shows the detection of a zero Doppler signal that has been shifted in time and has some arbitrary arrival phase. The signal is shifted approximately three digits with an $\text{SNR} = 0 \text{ dB}$.

The sequence removal process is linear and can detect multiple signal arrivals. Figure 5.2 shows this clearly for two arrivals spaced 27 digits apart (approximately 6 seconds) with $\text{SNR} = -15 \text{ dB}$. Figure 5.3 shows the same signal with one digit separation, and the signal arrival times are still resolvable.

The next figure, Figure 5.4, demonstrates the processing gain from coherent averaging. The data is the same as that in Figure 5.1, but with each segment output averaged over three frames, and a larger shift in arrival time. Note the increased SNR as compared with Figure 5.1.

Figures 5.5 and 5.6 show the detection of two Doppler shifted signals. The first is for an opening situation with -1.4 knots of Doppler and -12 dB SNR. The second is a closing situation with 3.4 knots Doppler and a SNR of -15 dB.

Graphs of all Doppler bins searched are include in Appendix A. Appendix B gives a brief description of the program SEQREM.

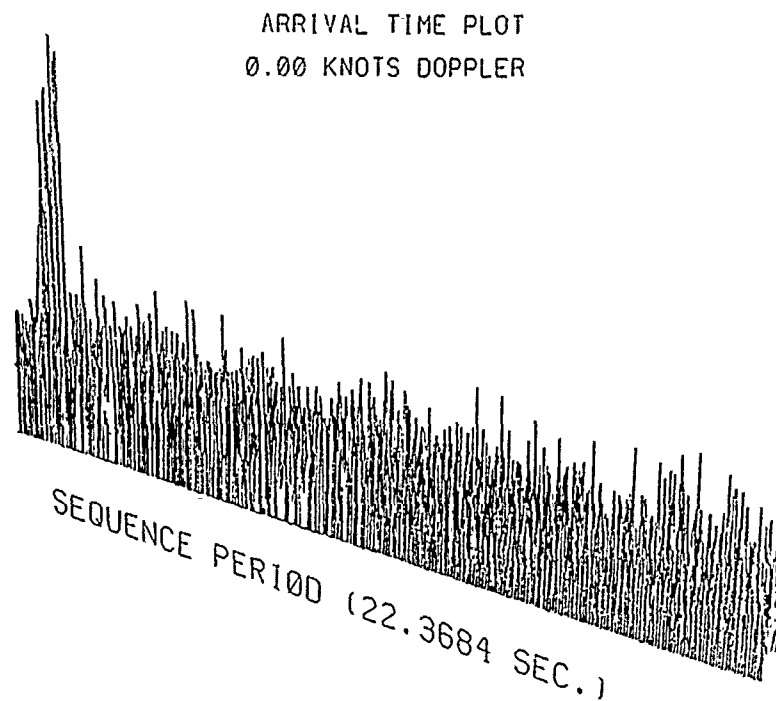


Figure 5.1: Zero Doppler signal. $N=255$, $Q=5$, $f_c=57$ Hz, 0 dB SNR.

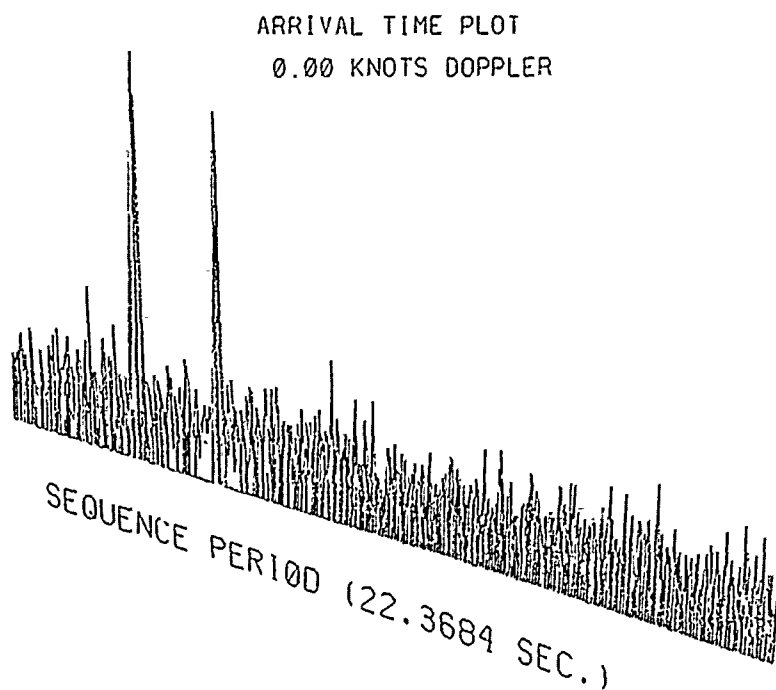


Figure 5.2: Zero Doppler signals, 27 digits apart. $N=255$, $Q=5$, $f_c=57$ Hz, -15 dB SNR.

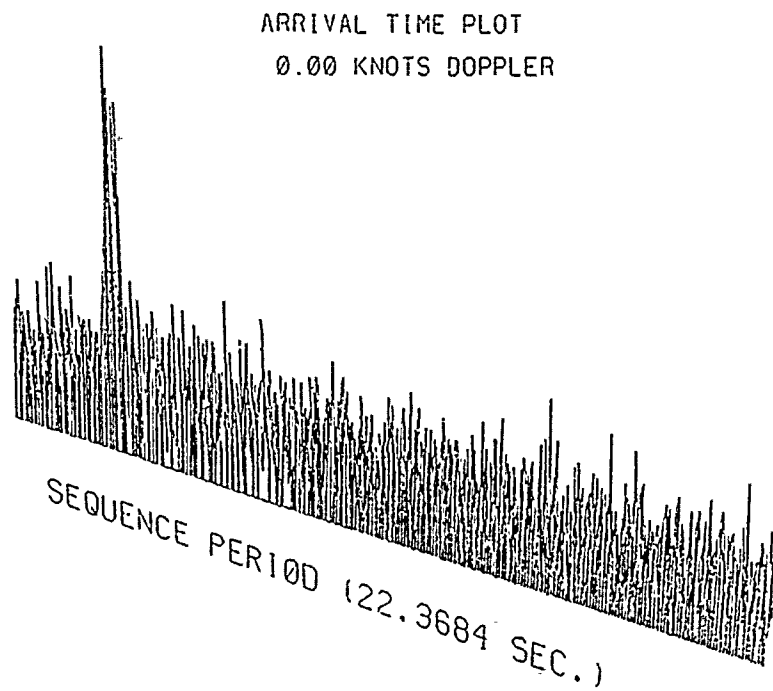


Figure 5.3: Zero Doppler signals, 1 digit apart. $N=255$, $Q=5$, $f_c=57$ Hz, 0 dB SNR.

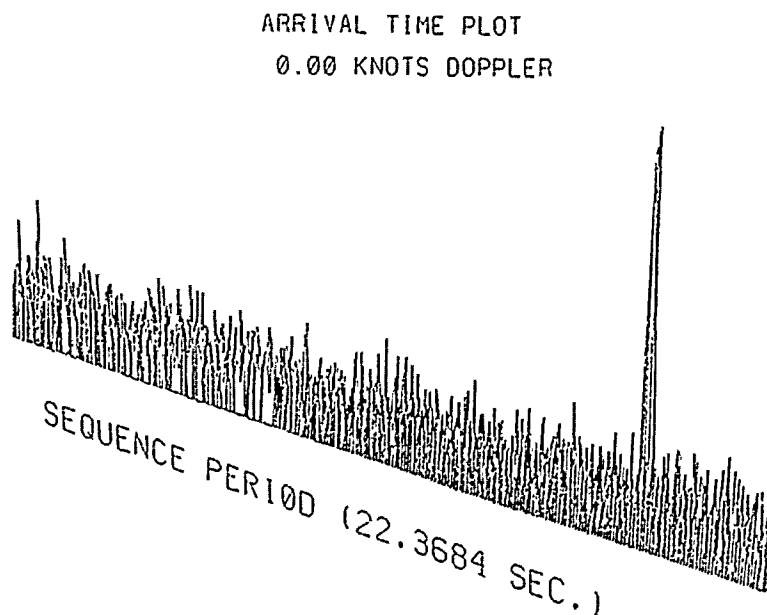


Figure 5.4: Zero Doppler signal. $N=255$, $Q=5$, $f_c=57$ Hz, -15 dB SNR coherently averaged over 3 frames.

ARRIVAL TIME PLOT
-1.50 KNOTS DOPPLER

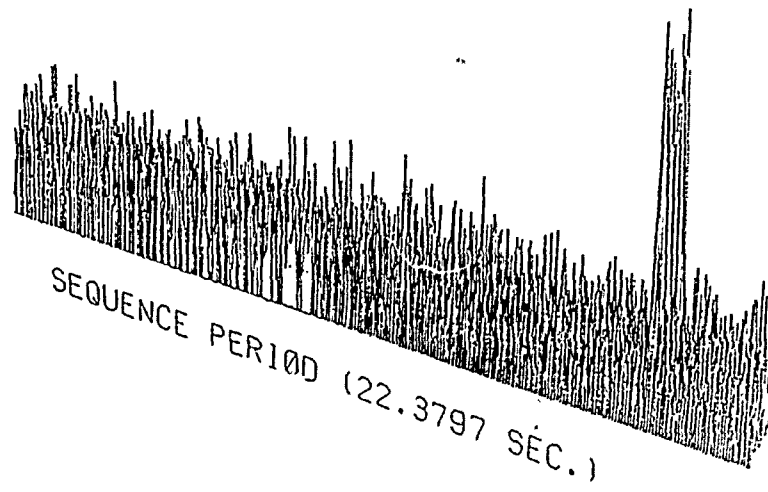


Figure 5.5: -1.4 knot Doppler signal. $N=255$, $Q=5$, $f_c=57$ Hz, -12 dB SNR.

ARRIVAL TIME PLOT
3.50 KNOTS DOPPLER

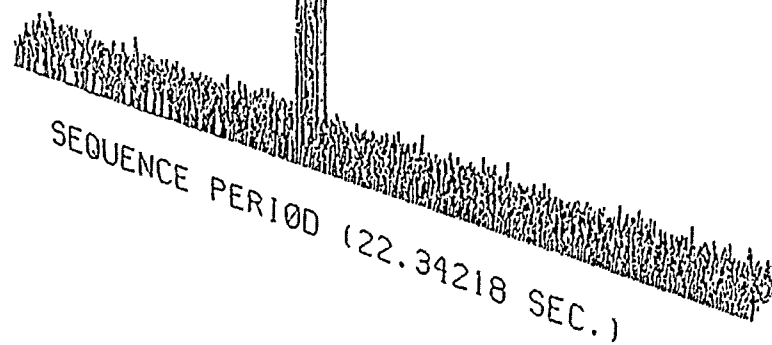


Figure 5.6: 3.4 knot Doppler signal. $N=255$, $Q=5$, $f_c=57$ Hz, -15 dB SNR.

B. CONCLUSIONS

The results of the previous section show that the software developed for this thesis operates as desired. Sequence detection is performed as predicted, via software compensation of sampling frequency, for any reasonable Doppler using linear interpolation techniques. Linear interpolation may not be the optimum interpolation method, but it has been demonstrated that it does work and is simple and faster than more complex techniques. This software will enable the Naval Postgraduate School (NPS) to participate in the Heard Island Experiment and will provide an important processing capability for any future experiment, and most importantly, a Doppler capability that was not previously possessed by the Underwater Acoustics Systems Laboratory of the Electrical and Computer Engineering Department at NPS.

C. ADDITIONAL WORK

Despite all of the advantages of this software, further work is necessary. The program as it stands, operates close to real time on individual segments, given the inherently low data rates used in acoustic tomography, but resides on a ULTRIX-32 V2.0 operating system with a VAX 11/785 processor. This is a time sharing system and is not designed for real time processing. Because of this, long data sets end up with a reduced priority as time progresses, causing slower execution from the user's perspective.

This problem will be remedied with the delivery of a Concurrent Computer Corporation VME Native-Mode Data Acquisition System in November of 1990. The intent is to transfer this software from the ULTRIX-32 system to this machine. The VME supports multiple processors and can process 14 MIPS at 33

MHz. It uses a memory cached system of up to 128 MB. It can also support up to 7 separate graphics heads. This system is capable of real time operation with a UNIX type operating system and will be ideal for use with this programming package. Given this system, the work that needs to be done includes:

1. Transfer of the existing program and related functions to the VME system. This includes the SEQREM program, initialization functions, FHT function, and magnitude computation routines.
2. Modification of the programs to incorporate real time processing in a parallel vice serial structure. Data input to the system is first read, processed, stored to file, and then the next segment is read and so on. A parallel scheme is needed to allow reading of the next data set while computation is progressing on the current while the results of the last set are being displayed.
3. Design of an I/O system to handle data input and output channel(s). Without the VME available, it was necessary to assume input data is maintained in text file. Output is also to a file with sequential Doppler searches appended to the same data file. The software also assumes single channel input. It would be desirable to handle multiple channels.
4. Development of a display software/system to present the data in an convenient form, such as a waterfall type display. Currently, plotting of data is performed off-line, after all computations are completed. There is little flexibility for plotting various size data sets and real time display is not possible.
5. Altering the Doppler bin search from a sequential to a parallel operation. Currently in the data processing section, Doppler processing is performed in increments via a standard "for" loop.

APPENDIX A

The results of Doppler processing for the three possible cases are given in the following figures. Figure A.1 shows a zero Doppler signal. No scan is made, since no motion is expected. Figure A.2 is a closing Doppler situation. Doppler search is over a + or - 3.5 knot range, in 0.5 knot steps. Figure A.3 is the opening case with a search over + or - 3.0 knots, also in 0.5 knot increments. In each figure the following parameters apply: primitive polynomial = 537₈, N = 255, carrier = 57 Hz, $\psi = 45$ degrees, initial state = 1₈, Q = 5. White Gaussian noise is assumed. Plots are displayed as arrival time versus time. This data simulates a signal to be transmitted in the Heard Island Experiment.

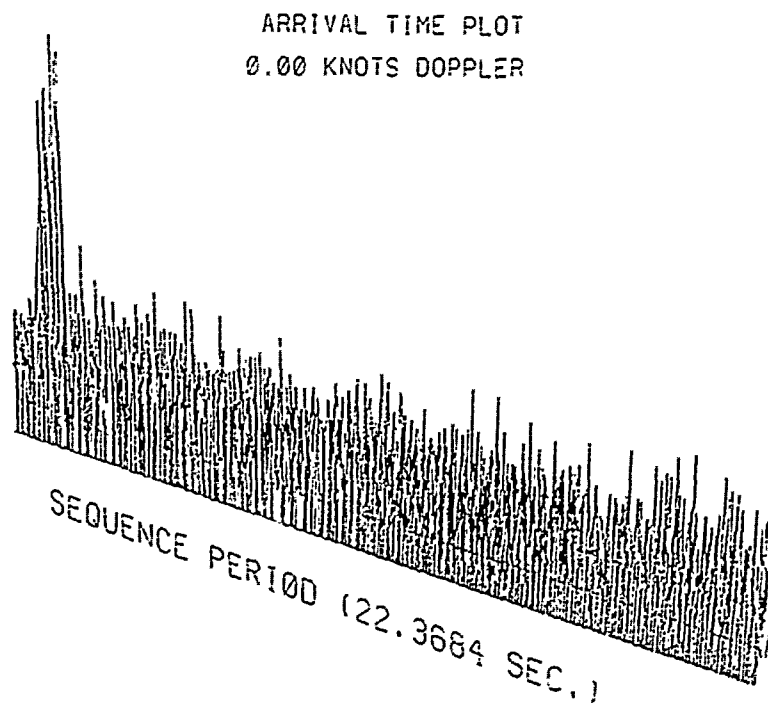
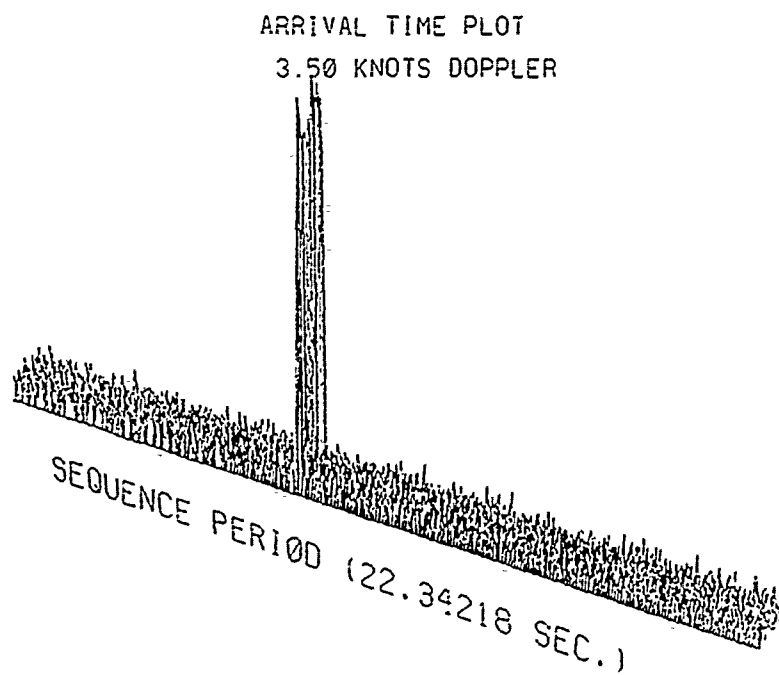
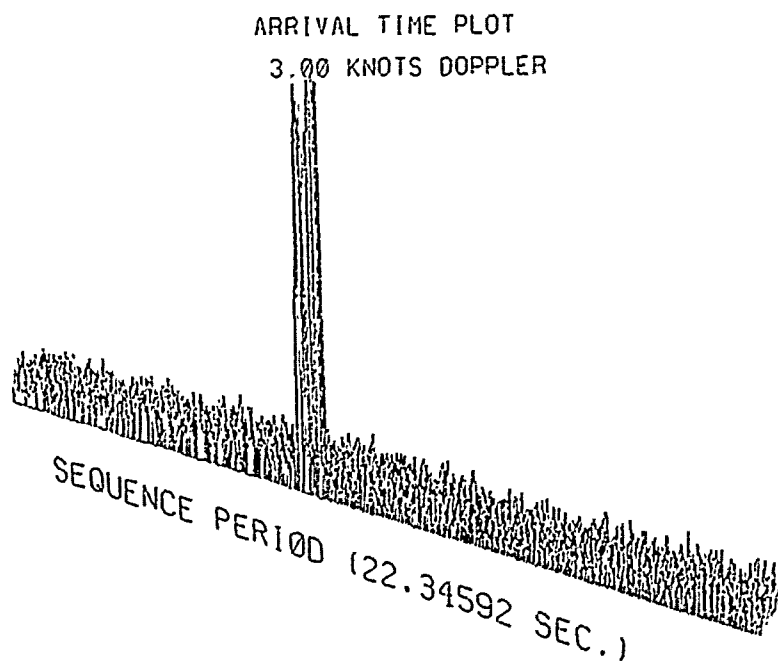


Figure A.1: Zero Doppler signal. SNR=0 dB.

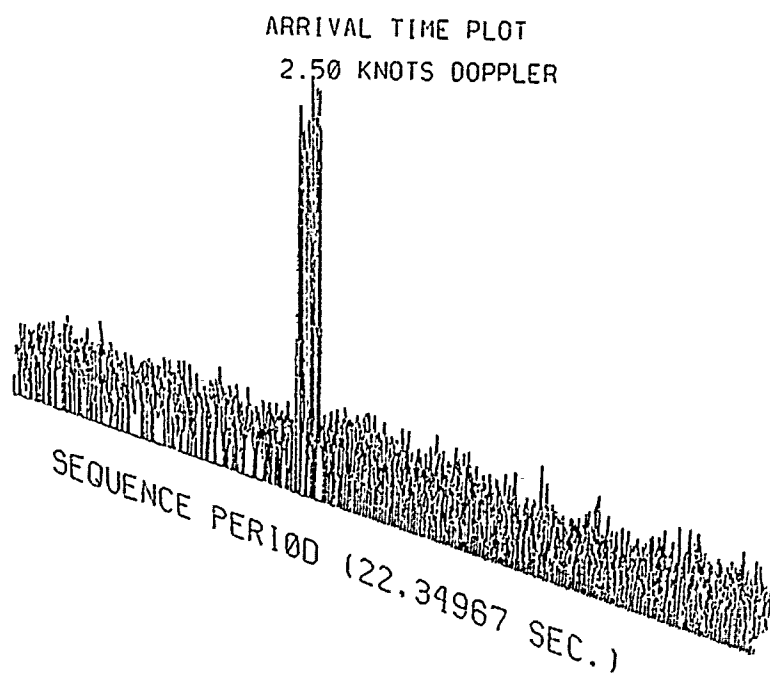


(a)

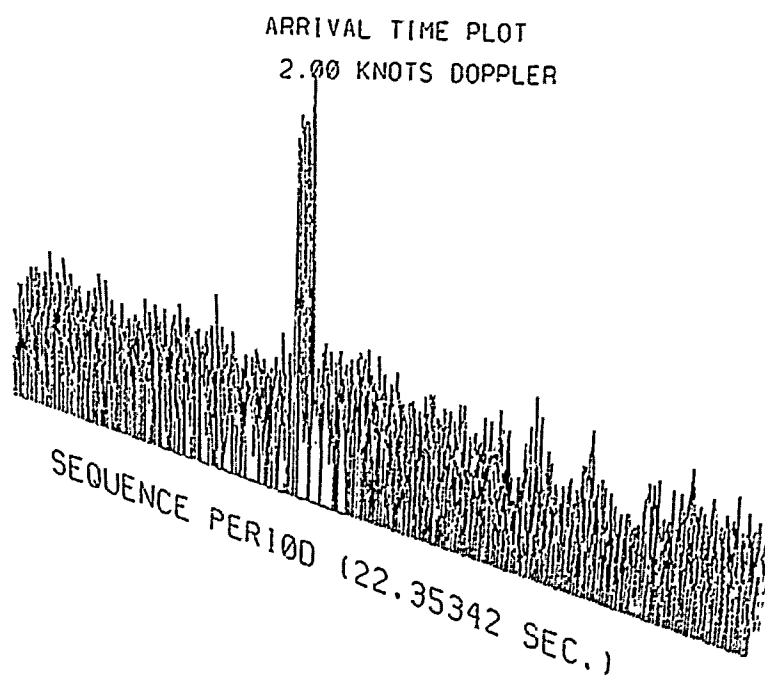


(b)

Figure A.2 : 3.4 knot Doppler signal, 0.5 knot steps. SNR= -15 dB.



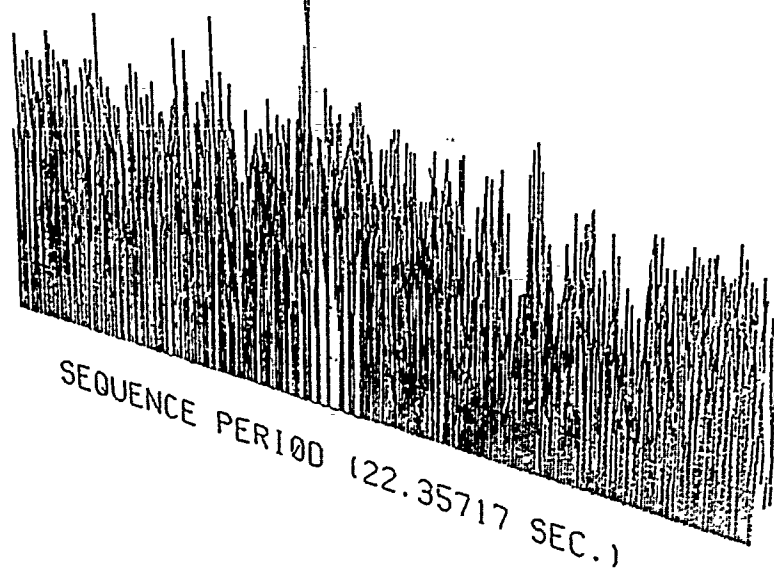
(c)



(d)

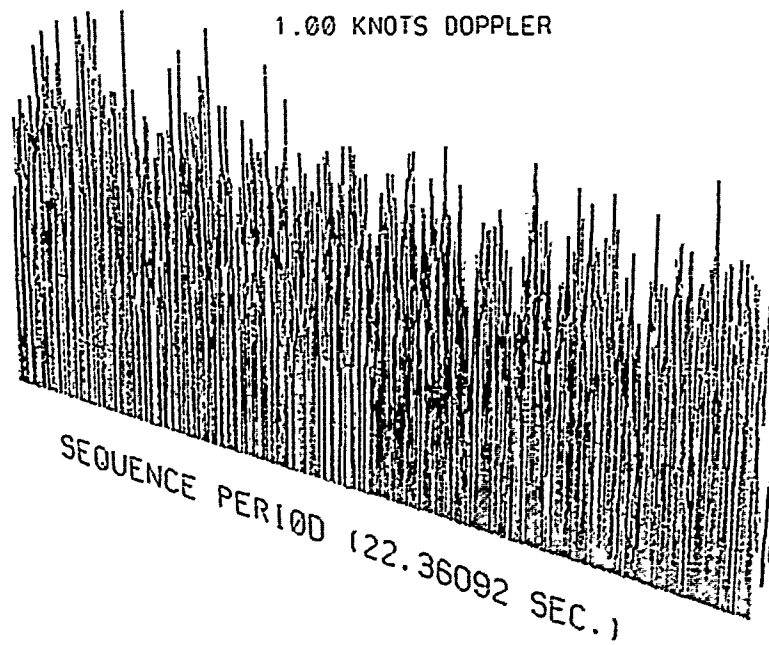
Figure A.2 : (cont.)

ARRIVAL TIME PLOT
1.50 KNOTS DOPPLER



(e)

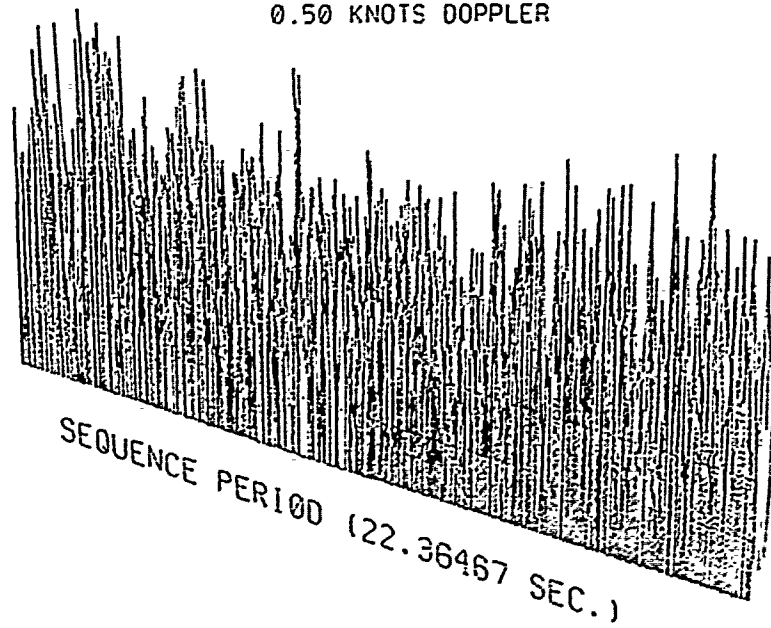
ARRIVAL TIME PLOT
1.00 KNOTS DOPPLER



(f)

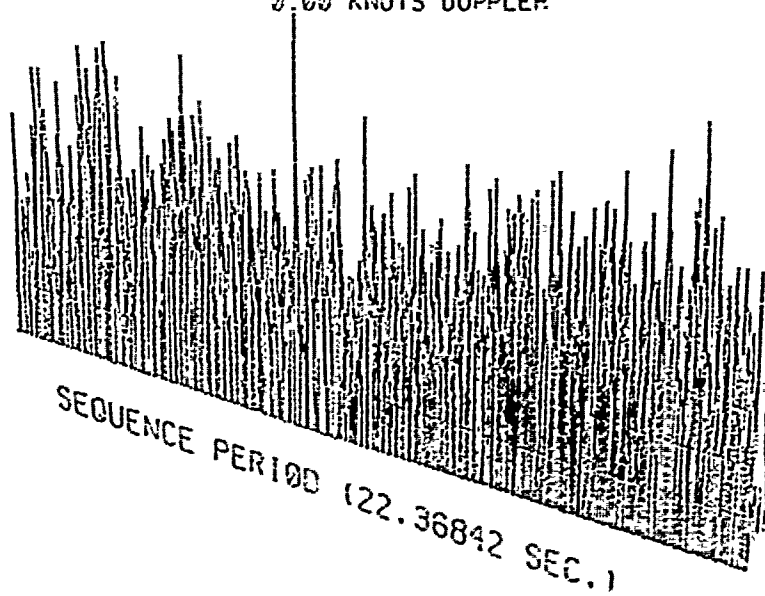
Figure A.2 : (cont.)

ARRIVAL TIME PLOT
0.50 KNOTS DOPPLER



(g)

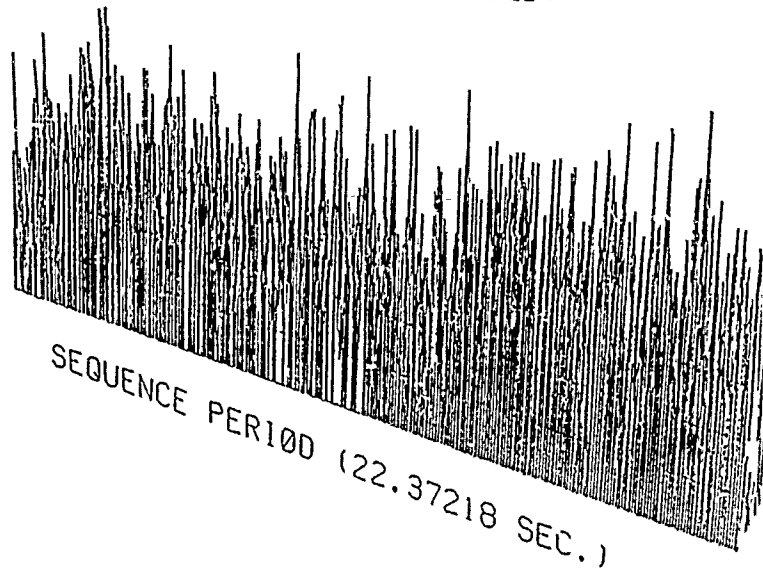
ARRIVAL TIME PLOT
0.60 KNOTS DOPPLER



(h)

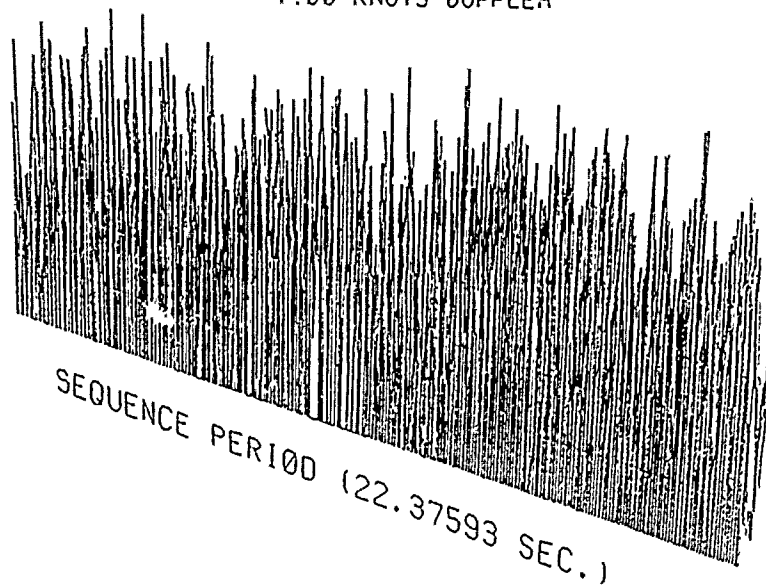
Figure A.2 : (cont.)

ARRIVAL TIME PLOT
-0.50 KNOTS DOPPLER



(i)

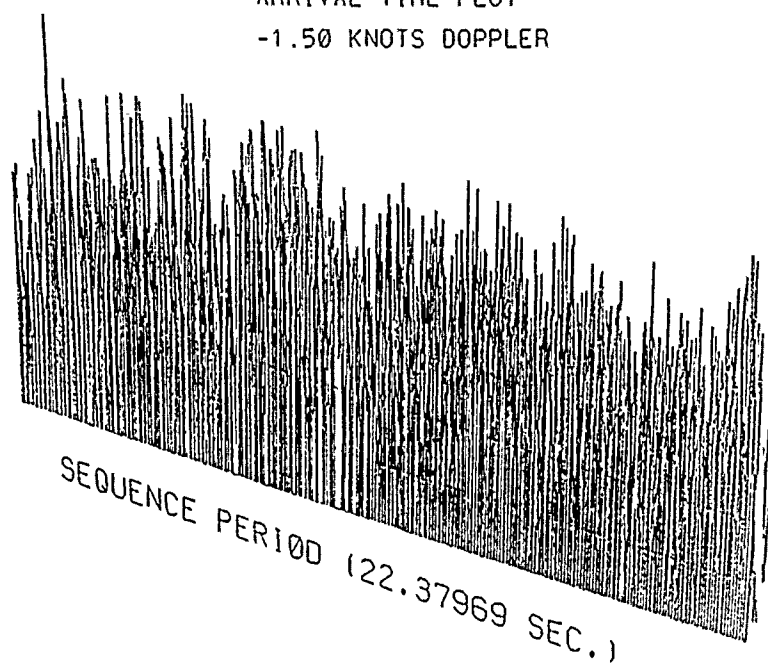
ARRIVAL TIME PLOT
-1.00 KNOTS DOPPLER



(j)

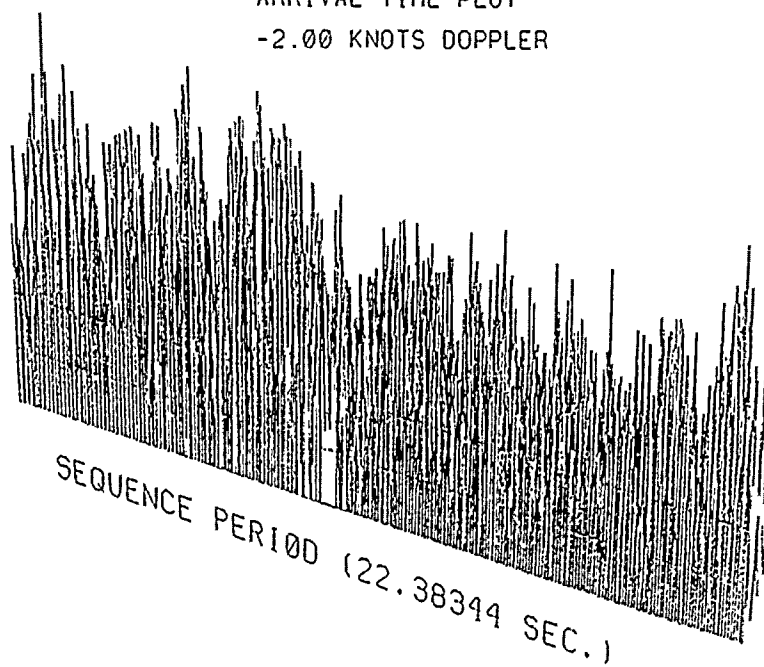
Figure A.2 : (cont.)

ARRIVAL TIME PLOT
-1.50 KNOTS DOPPLER



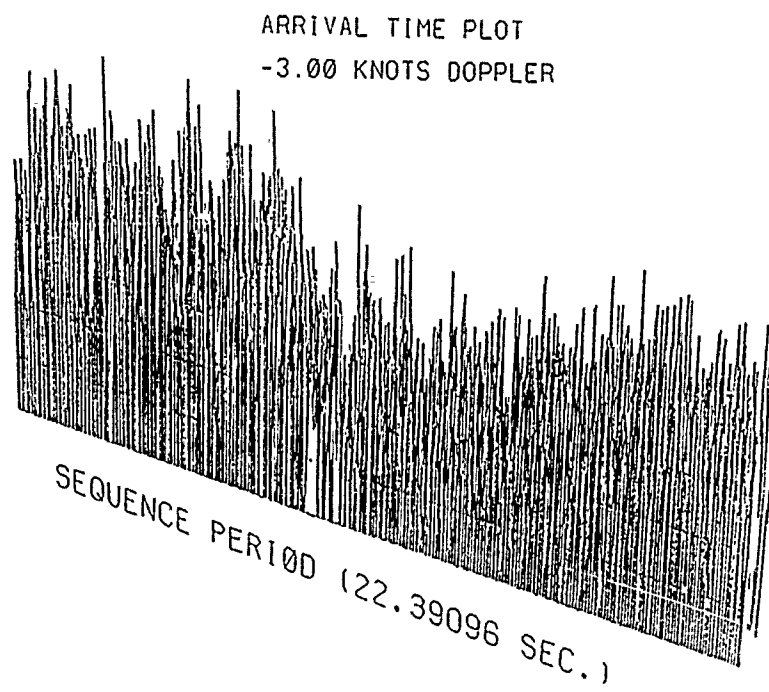
(k)

ARRIVAL TIME PLOT
-2.00 KNOTS DOPPLER

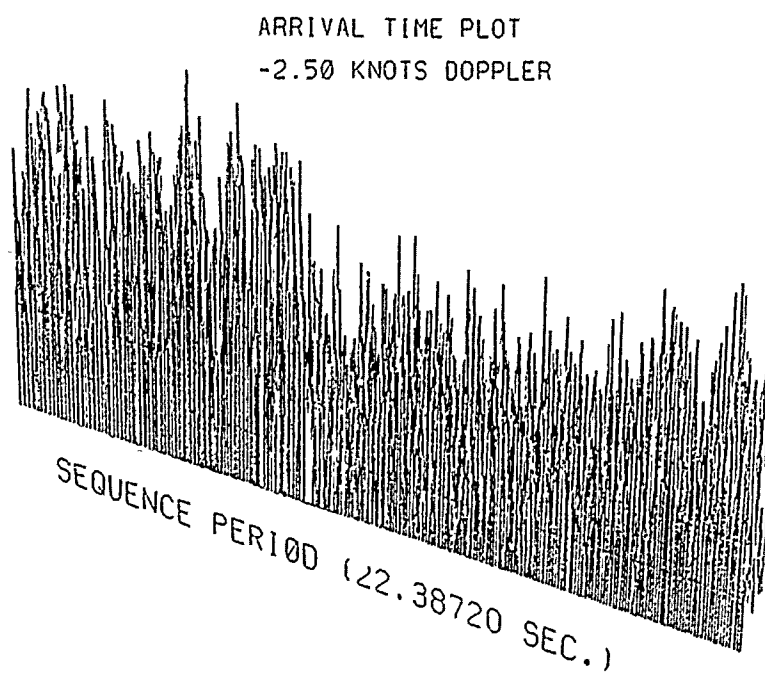


(l)

Figure A.2 : (cont.)



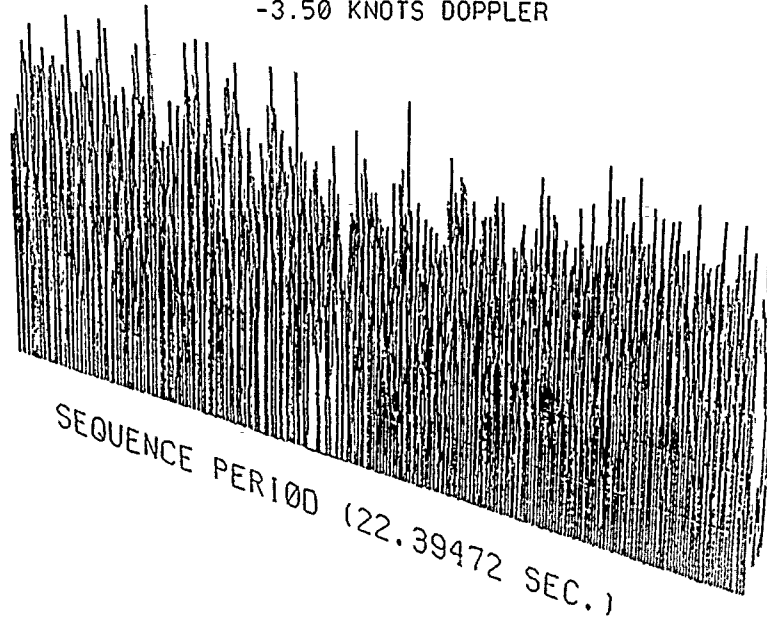
(m)



(n)

Figure A.2 : (cont.)

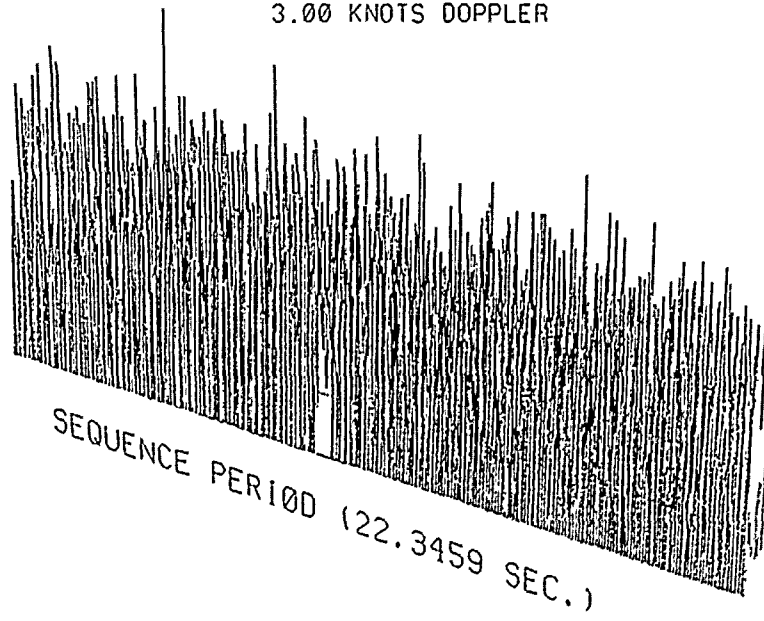
ARRIVAL TIME PLOT
-3.50 KNOTS DOPPLER



(o)

Figure A.2 : (cont.)

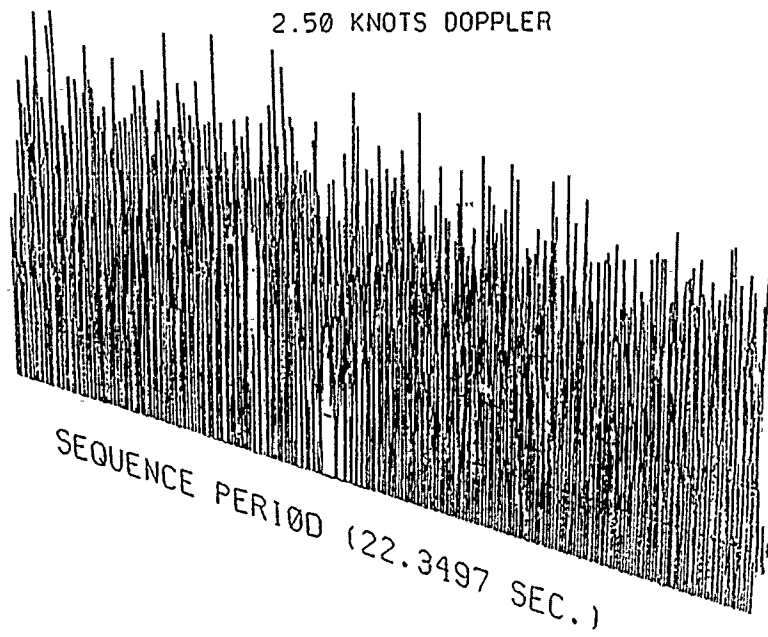
ARRIVAL TIME PLOT
3.00 KNOTS DOPPLER



(a)

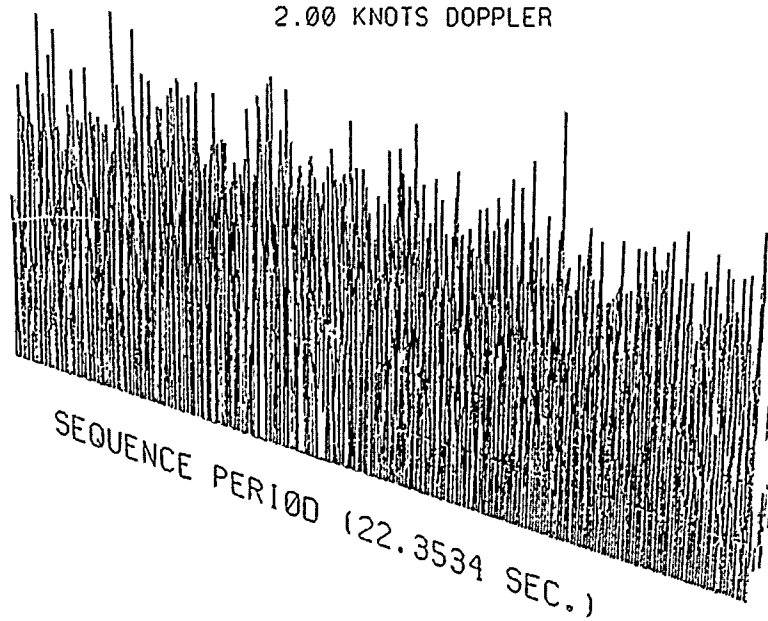
Figure A.3 : -1.4 knots Doppler, 0.5 knot steps. SNR= -12 dB.

ARRIVAL TIME PLOT
2.50 KNOTS DOPPLER



(b)

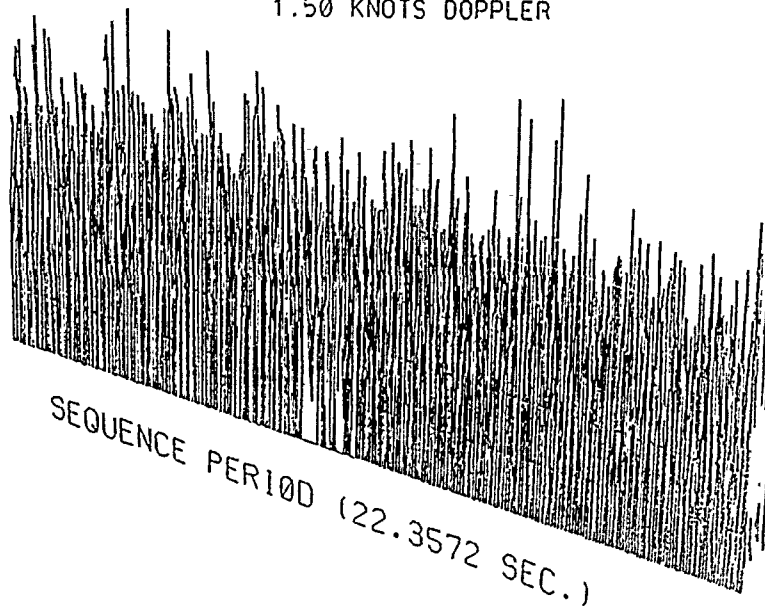
ARRIVAL TIME PLOT
2.00 KNOTS DOPPLER



(c)

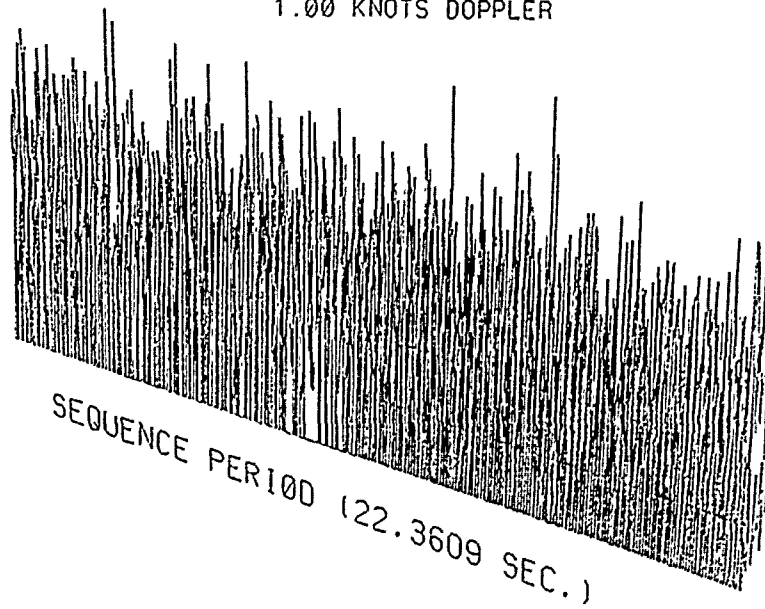
Figure A.3 : (cont.)

ARRIVAL TIME PLOT
1.50 KNOTS DOPPLER



(d)

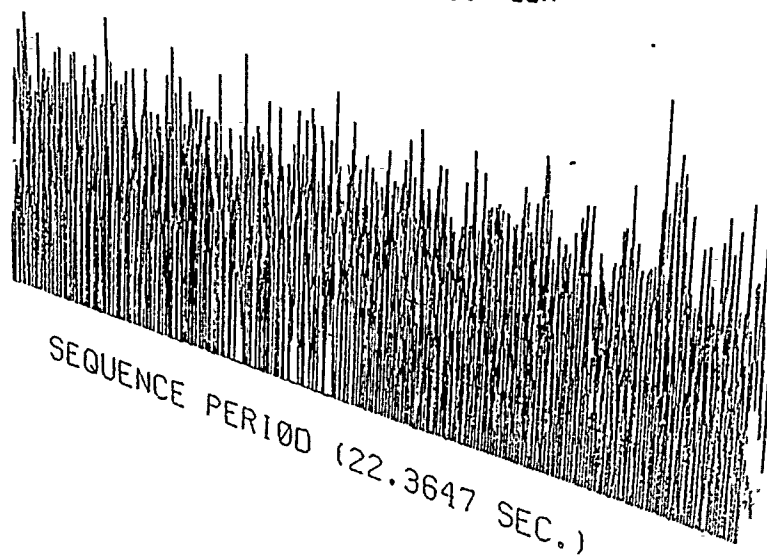
ARRIVAL TIME PLOT
1.00 KNOTS DOPPLER



(e)

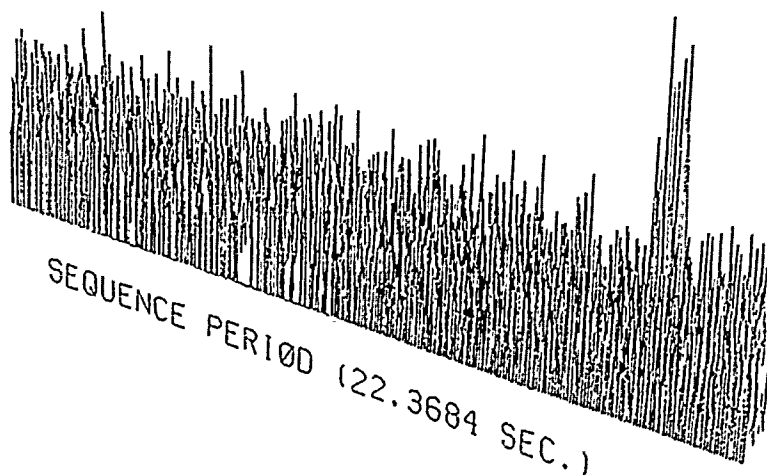
Figure A.3 : (cont.)

ARRIVAL TIME PLOT
0.50 KNOTS DOPPLER



(f)

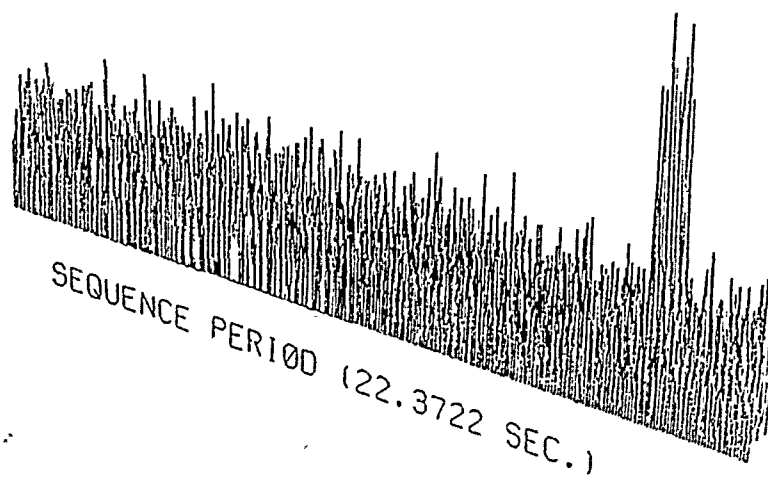
ARRIVAL TIME PLOT
0.00 KNOTS DOPPLER



(g)

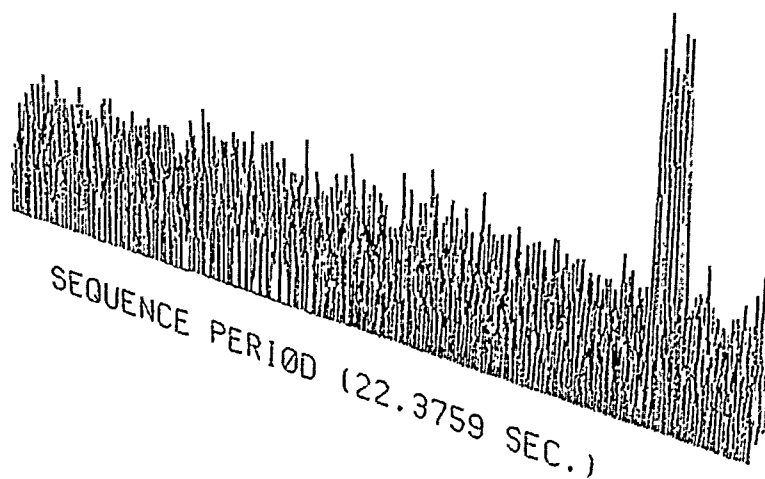
Figure A.3 : (cont.)

ARRIVAL TIME PLOT
-0.50 KNOTS DOPPLER



(h)

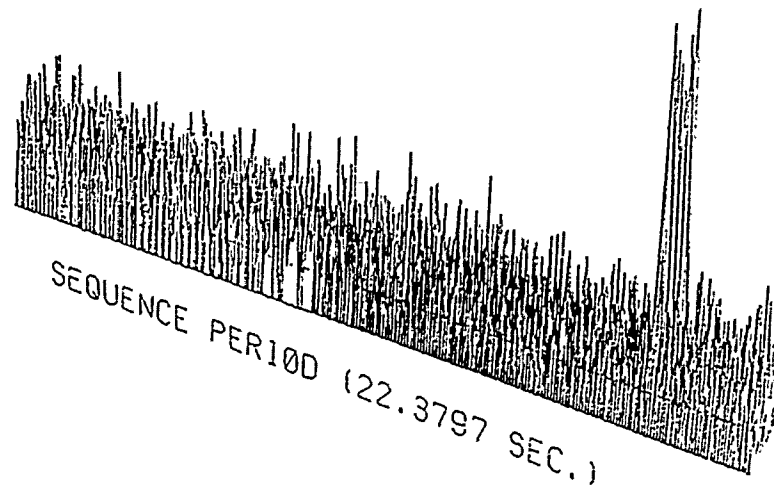
ARRIVAL TIME PLOT
-1.00 KNOTS DOPPLER



(i)

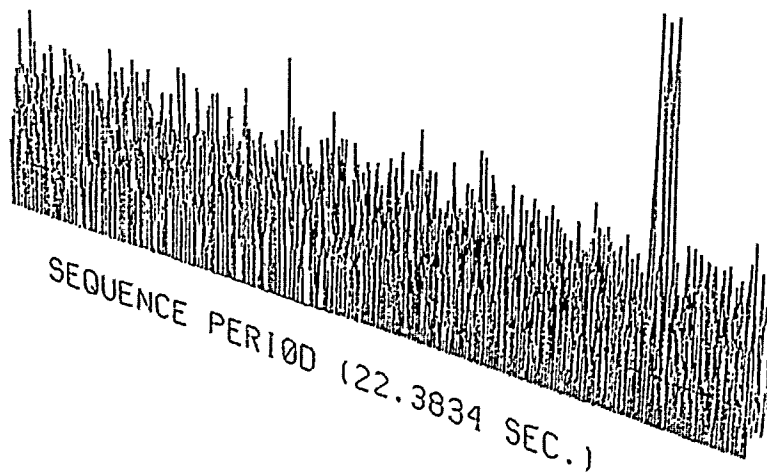
Figure A.3 : (cont.)

ARRIVAL TIME PLOT
-1.50 KNOTS DOPPLER



(j)

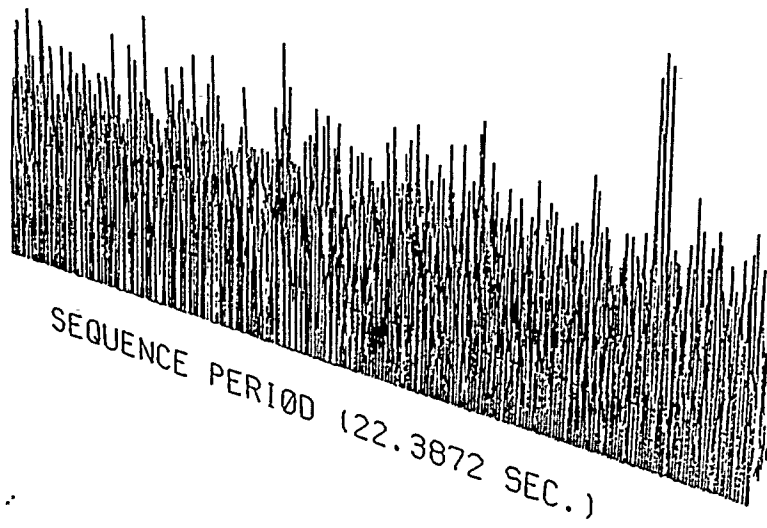
ARRIVAL TIME PLOT
-2.00 KNOTS DOPPLER



(k)

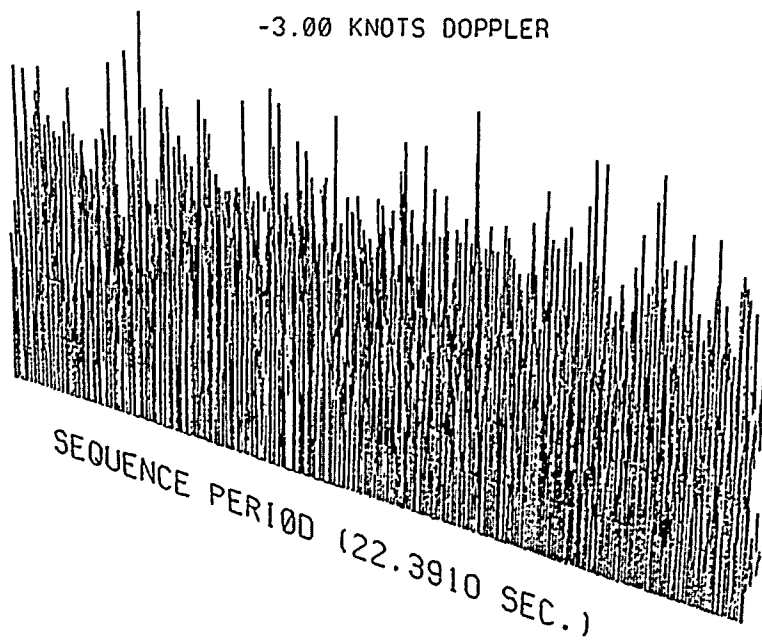
Figure A.3 : (cont.)

ARRIVAL TIME PLOT
-2.50 KNOTS DOPPLER



(l)

ARRIVAL TIME PLOT
-3.00 KNOTS DOPPLER



(m)

Figure A.3 : (cont.)

APPENDIX B

The following programs make up the SEQREM program or are associated utility programs. Since sufficient documentation is included with each program, only a very brief description is provided for each. This software may be obtained on floppy disk by contacting Professor James H. Miller at (408) 646-2384 or by writing to his address in the initial distribution list on page 98.

A. MAIN PROGRAM

1. MACROFILE

This is a short file that holds definition macros and global variables.

```
/* MACROFILE includes the necessary libraries and declares global constants
and global variables for general use.

Variables:

    scram - pointer to indices to scramble input data vector prior to FHT.

    unscram - pointer to restore data vector after FHT performed.

    law - polynomial law used to generate M-sequence.

    degree - the degree of law.

    initial - initial register load for generation of the sequence. */

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

#define BTRMAX 11
#define PI 3.1415926536
#define FALSE (unsigned)0
#define TRUE (unsigned)!FALSE
unsigned *scram, *unscram, law, degree, initial;
char *malloc();
```

2. SEQREM

This program is the root of all the processing functions for sequence removal. It uses standard file I/O and may be used with any standard input numerical format.

The program call is SEQREM, and the user is prompted for the following information before processing may begin (Note: some parameters are requested from within initialization functions, however, this is transparent to the user):

1. Data file containing formatted filter coefficients.
2. Primitive polynomial defining the m-sequence.
3. Initial register load.
4. Transmission direction, i.e. forward or reverse code.
5. Carrier frequency.
6. Sampling frequency (checks for four times carrier).
7. Phase modulation angle.
8. Cycles per digit, Q.
9. Desired time decimation, if any.
10. Coherent averaging desired, if any.
11. Expected Doppler range (knots).
12. Doppler search step size (knots).
13. Input data file.
14. Output data file.

Assumptions:

1. Sampling at four times carrier.
2. One period of m-sequence processed at a time.
3. M-sequence is taken from the least significant register.
4. Decimation ratio is a rational.

/* SEQ_REM performs sequence removal of a phase encoded M-Sequence. It is designed to work for any maximal length sequence as defined by a primitive polynomial. The polynomial, carrier frequency, sampling frequency, etc. are input by the user. All filtering and demodulation is performed, but filter coefficients must be stored in an external file in the format described by the function GET_FLT_COEF(). For ease of use, these may be generated off line by the MATLAB program SET_FILTER.M.

M-Sequences may be transmitted in the forward or reverse direction.

? - is used as a wild card for brevity and may indicate 1,2,a,b etc. in the following variable and file descriptions.

Variables:

c,j,i,k,n,count - integer counters used in various loops.

dir,yesno - variables used for user inquiries.

num_pts - number of points processed per digit, before and after decimation.

num_coh_avg - number of frames to be coherently averaged.

vect_len - number of points in a sequence before decimation in time.

vect_len_inter - number of points need to be read for interpolation of one segment length at the new sampling freq.

step - step size determined for decimation in time.

cycles - number of carrier cycles per digit, must be integer number.

first_run - flag to determine if a segment is the first one to be processed in the current doppler bin.

end - flag to determine when end of file has been reached.

seq_len - number of digits in a M-sequence.

highpass_?,lowpass?_? - pointer to storage for filter coefficients.

indata1,indata2 - array for data storage and processing storage before decimation in time. Indata1 is initially input and then is imaginary part and indata2 is real part (after demodulation).

h_data_im,h_data_re - arrays for real and imaginary vectors for storing scrambled data for FHT processing.

re_data,im_data - arrays for real and imaginary data storage after data unscrambling.

avg_re, avg_im - arrays for real and imaginary data storage when coherent averaging is performed.

mag, phase - array for magnitude and phase of processed data.

fc,fs - pointers for carrier frequency and sampling frequency.

dclvl_im,dclvl_re,pdstal_re,pdstal_im - used to remove correlation dc bias.

ifact,rfact,ph_ang - factors and phase angle used to compute correction to dc bias based on phase modulation angle.

den - temporary storage used in above calculations of dc bias.

snd_vel - input average sound velocity to be used.
 Ts,Ta,Ti - sampling period, sequence period, and interpolated sampling period in doppler interpolation.
 time_s,time_i - current relative sample time and interpolation time, which are used to compute h.
 h - distance from time a to sample to be interpolated where time a < h time b.
 ptr1,ptr2 - used to save position in array space when performing doppler search.
 doppler - current doppler shift being processed (m/s).
 doppler_start,doppler_end - beginning and ending of doppler search interval (m/s).
 doppler_step,doppler_bin - doppler step size used in scan, and doppler converted to radians for demodulation.

Functions:

init_had() - gets relevant parameters for computing scrambling and unscrambling indices.
 fwd_had(),rev_had() - compute indices for foward and reverse transmitted sequences. Both return polynomial degree from input.
 lowpass?(),highpass() - identical functions for performing filtering operations.
 get_fit_coef() - retrieves filter coefficient data from user provided file.
 demodulate() - performs demodulation for input signal to dc. Has an argument doppler_bin for doppler frequency shifts.
 hadamard() - performs the FHT on scrambled data. Identical to FFT with multiplication factors omitted.
 mag_phase() - computes the magnitude and phase of the processed data.*/

#include "macrofile.h"

```

main()
{
  int c,j,k,n,i,dir,cycles,num_pts,step,vect_len,vect_len_inter,count,num_coh_avg;
  unsigned init_had(), rev_had(),seq_len, first_run, end;
  void get_fit_coef(),hipass(),lowpass1(),demodulate(),hadamard(),mag_phase();
  void lowpass2(),lowpass3();
  float *hipass_a,*hipass_b,*lowpass_a,*lowpass_b,*lowpass1_a,*lowpass1_b;
  float *indata2, *h_data_re, *h_data_im, *re_data, *im_data, *fs, *fc;
  float *mag, *phase,*avg_re,*avg_im,dclvl_re,dclvl_im,pdstal_re,pdstal_im;
  float *data_i_re, *data_i_im, rfact, ifact, den, ph_ang, doppler,snd_vel;
  float doppler_start,time_s, time_i, h, Ts, Ta, Ti, *temp_ptr1, *temp_ptr2;
  float doppler_end,doppler_bin,doppler_step,*indata1,*new();
  int yesno;

```

```

char infile[21], outfile[21];
FILE *fp1,*fp2;

/* Initialize filter coefficients for a low and high pass filter. */
/* These values are computed and stored in a file off line. */

count = 0;

/* Allocate memory for fc, fs, and filter coefficients to be used. 10th
order BUTTERWORTH bandpass, and 5th order CHEBYCHEV lowpass filters
must be used. Filter coefficients are retrieved and filters are
initialized. Memory allocated to filter coefficients is freed after
initialization. */

fc = new(1);
fs = new(1);
hipass_a = new(BTRMAX);
hipass_b = new(BTRMAX);
lowpass_a = new(BTRMAX);
lowpass_b = new(BTRMAX);
lowpass1_a = new(BTRMAX);
lowpass1_b = new(BTRMAX);
getflt_coef(hipass_a,hipass_b,lowpass_a,lowpass_b,lowpass1_a,lowpass1_b);
hipass(hipass_b,hipass_a,BTRMAX);
lowpass1(lowpass_b,lowpass_a,BTRMAX-5);
lowpass2(lowpass_b,lowpass_a,BTRMAX-5);
lowpass3(lowpass1_b,lowpass1_a,BTRMAX);
free(hipass_a);
free(hipass_b);
free(lowpass_a);
free(lowpass_b);
free(lowpass1_a);
free(lowpass1_b);

/* Set sound velocity to desired value, for given conditions. */

printf("\nEnter the Average Sound Velocity (meters/sec.).");
printf("\n\n          Sound Velocity: ");
scanf("%f",&snd_vel);

/* Initialize parameters for M-Sequence, and set transmit direction.
Set scrambling and unscrambling indices accordingly. */

initial = init_had();
printf("\nIs the M-Sequence transmitted in the forward or reverse\n");
printf("direction? \n");
printf("\n          (forward=0/reverse=1): ");
scanf("%d",&dir);
if (dir == 0)
    degree = fwd_had(law,initial,scram,unscram);
else
    degree = rev_had(law,initial,scram,unscram);
seq_len = (1<<degree)-1;

/* Compute offset to remove DC bias from correlation. */

printf("\nEnter the Phase Modulation Angle used to encode the signal.\n ");
printf("\n          Phase Angle: ");
scanf("%f",&ph_ang);

```

```

ph_ang = ph_ang*PI/180;
den = seq_len*seq_len + tan(ph_ang)*tan(ph_ang);
ifact = (seq_len+1)*tan(ph_ang)/den;
rfact = -(seq_len-tan(ph_ang)*tan(ph_ang))/den;

/* Input of transmission parameters and demodulator is initialized. */

printf("\nEnter the Carrier Frequency used.\n ");
printf("\n          Fc: ");
scanf("%f",&fc[0]);
printf("\nEnter the Sampling Frequency. Sampling Frequency must be an");
printf("\nfour times the Carrier Frequency.\n");
printf("\n          Fs: ");
scanf("%f",&fs[0]);
if ( *fc/*fs) != 0.25)
{
    printf("\nSampling Frequency will not work with this program. Exiting!\n");
    exit(1);
}
printf("\nEnter the number of carrier cycles per digit. An ");
printf("\ninteger number of cycles must be used.\n");
printf("\n          Q: ");
scanf("%d",&cycles);

/* Compute sequence length in seconds (Ta) and sampling period. The number
   of points generated per digit is computed and provided as an aid in
   determining the decimation in time to be used, if desired. The total
   number of points per sequence period is computed (vect_len).*/

num_pts = (*fs/*fc)*cycles;
vect_len = num_pts*seq_len;
Ta = (float)(seq_len * cycles / *fc);/* computes the sequence length in sec.*/
Ts = 1/*fs); /* compute sampling period. */
printf("\nThere are %d pts per digit. For Processing Savings ",num_pts);
printf("\ndecimation in time is performed. The default used is one point");
printf("\nfor each cycle in the digit. In this case %d points are",cycles);
printf("\nprocessed per digit. (i.e. every %d th point.)", (num_pts/cycles));
printf("\n\n          Use default?(yes=1/no=0): ");
scanf("%d",&yesno);
if (yesno == 1 )
    step = num_pts/cycles;
else
{
    printf("\nEnter the desired Decimation. (e.g. for every point enter 1,\n");
    printf("for every other point enter 2, for every third point enter 3, etc");
    printf("\nDecimation must be evenly divisible into the pts. per digit.");
    printf("\n(NOTE: Decimation must be less than %d to ensure \n",num_pts);
    printf("          at least one point per digit is used!)\n");
    printf("\n          Decimation: ");
    scanf("%d",&step);
    if ( num_pts%step != 0)
    {
        printf("\nInvalid decimation. Try again.");
        printf("\n          Decimation: ");
        scanf("%d",&step);
        if ( num_pts%step != 0)
            printf("\nError. Bye!\n");
    }
    if (step > num_pts)
    {
        printf("\nDecimation chosen is too large, use a smaller number.\n");
    }
}

```

```

        printf("\n          Decimation: ");
        scanf("%d",&step);
        if (step > num_pts)
            printf("\nSorry! Still won't work, aborting\n");
    }

/* Determine if coherent averaging of sequences is desired for improving
   processing gain. */

printf("\nEnter the number of Sequence Frames to be coherently averaged.");
printf("\nNumber Frames = 1 implies no coherent averaging desired.\n\n");
printf("          Number Frames: ");
scanf("%d",&num_coh_avg);
if (num_coh_avg <= 0)
{
    printf("\nInvalid Number of Frames. Must Use Positive Integer.");
    printf("\n\n          Number Frames: ");
    scanf("%d",&num_coh_avg);
    if (num_coh_avg <= 0)
    {
        printf("\nInvalid Number of Frames. Aborting!\n");
        exit(1);
    }
}

/* Query for Doppler range to be searched. An input of zero will cause
   the resampling for doppler steps to be skipped. Doppler is converted
   from knots to meters/sec for computation. Sequence frequency resolution
   is computed to assist in determining doppler bins to be searched. */

printf("\nEnter the doppler range to be searched (knots).");
printf("\n\n          Doppler (+/-): ");
scanf("%f",&doppler);
printf("\nOne knot corresponds to a %1.5f Hz shift",*fc-*fc*(1.0-0.5/snd_vel));
printf(" in frequency.\nEnter the Doppler increment to be searched.");
printf("\n\n          Step (knots): ");
scanf("%f",&doppler_step);
if (doppler_step == 0.0)
    doppler_step = 1.0;
doppler /= 2.0;          /* convert knots to meters/second */
doppler_step /= 2.0;
doppler_start = -doppler;
doppler_end = doppler;

/* Query for input and output files. */

printf("\nEnter Input File Name (20 Characters Maximum).\n");
printf("\n          File Name: ");
scanf("%s",infile);
printf("\nEnter Output File Name (20 Characters Maximum).\n");
printf("\n          File Name: ");
scanf("%s",outfile);
if ((fp1 = fopen(infile,"r"))==NULL)
{
    printf("\nUnable to Open Input File. Aborting!\n");
    exit(1);
}
if ((fp2 = fopen(outfile,"w"))==NULL)
{
    printf("\nUnable to Open Output File. Aborting!\n");

```

```

    exit(1);
}

/* After Decimation there are less data pts => comput new num_pts. */

num_pts /= step;

/* Set data length for input to maximum needed for highest doppler,
   which corresponds to the maximum number of points used in interpolation.*/

Ti = Ts/(1 + doppler_end/snd_vel);
vect_len_inter = Ti*vect_len/Ts + 1;

/* Allocate memory as required. */

indata1 = new(vect_len_inter);
indata2 = new(vect_len_inter);
im_data = new(num_pts*seq_len);
re_data = new(num_pts*seq_len);
mag = new(num_pts*seq_len);
phase = new(num_pts*seq_len);
avg_re = new(num_pts*seq_len);
avg_im = new(num_pts*seq_len);
h_data_im = new(seq_len+1);
h_data_re = new(seq_len+1);
data_i_re = new(vect_len);
data_i_im = new(vect_len);

/* DATA PROCESSING SECTION.
   Doppler processing is performed automatically according to the step
   size specified by the user. Data is processed in lengths corresponding
   one segment at a time. Sufficient points for complete interpolation
   of one doppler shifted sequence are picked off the input file.
   During doppler shift interpolation the input end point is saved for
   processing in the next segment to insure segment continuity. */

for (doppler = doppler_start; doppler <= doppler_end; doppler+=doppler_step)
{
    /* Initialize demodulation routine. */

    demodulate(fc,fs,doppler,vect_len);

    /* Determine the sampling period for the new sampling frequency used in
       interpolation, as a function of doppler. */
    Ti = Ts/(1 + doppler/snd_vel);

    /* If zero doppler case is being performed endpoint bookkeeping is not
       used and data length is the same as input data (vect_len). */

    if (doppler != 0 )
    {
        fscanf(fp1,"%f\n",&indata1[0]);
        vect_len_inter = Ti*vect_len/Ts;
        indata1++;
        *indata2++ = 0;
    }
    else
        vect_len_inter = vect_len;

    /* Initialize flags and interpolation parameters. first_run indicates

```

```

first segment being processed, and end indicates EOF reached. Time_s
indicates current relative sampling time and time_i indicates current
relative interpolation time. */

time_s = 0.0;
time_i = Ti;
end = FALSE;
first_run = TRUE;
doppler_bin = PI * doppler / (2 * snd_vel);

/* Ta is sequence length in seconds. Header is printed to separate
doppler scans. */

Ta = cycles * seq_len / (fc * (1 + doppler/snd_vel));
fprintf(fp2, "START\n");
fprintf(fp2, "%6.2f KNOTS DOPPLER\n", 2 * doppler);
fprintf(fp2, "SEQUENCE PERIOD (%7.4f SEC.)\n", Ta);
fprintf(fp2, "%5d %10.7f\n", seq_len * num_pts, Ta / (seq_len * num_pts - 1));

/* Process entire input file for current doppler search. */

while (!end)
{
    for (j=0; j<vect_len_inter; j++)
        if ((c=fgetc(fp1)) == EOF)
        {
            rewind(fp1);
            demodulate(fc, fs, doppler, 0); /* clears demodulator for next run.*/
            end = TRUE;
            break;
        }
        else
        {
            ungetc(c, fp1);
            fscanf(fp1, "%f\n", &indata1[j]);
        }

/* Test for EOF reached before complete segment read. Prevents processing
partial segments. */

        if (!end)
        {

/* Test for first segment. If it is the first data point has not been
filtered, adjusts bookkeeping of endpoint. */

            if ((doppler != 0.0) && first_run)
            {
                --indata1;
                --indata2;
                vect_len_inter += 1;
            }

/* Lowpassing and then highpassing reduces to a BP equivalent and avoids
complex numbers at this point. Caution: the filter phase must be linear
in the pass region. */

            hipass(indata1, indata1, vect_len_inter);
            lowpass3(indata1, indata1, vect_len_inter);
            demodulate(indata1, indata2, doppler_bin, vect_len_inter);
            lowpass1(indata1, indata1, vect_len_inter);
            lowpass2(indata2, indata2, vect_len_inter);

```

```

/* This loop performs a linear interpolation of the demodulates. Maintains
bookkeeping of endpoint, and first segment. Interpolates vect_len points
for hadamard processing. Skips this if zero doppler case. */

```

```

    if (doppler != 0.0)
    {
        i=0;
        if (first_run)
        {
            vect_len_inter -= 1;
            first_run = FALSE;
        }
        else
        {
            --indata1;
            --indata2;
        }
        for (j=0; j<vect_len; j++)
        {

/* First case is down shift of doppler. */

            if (time_i > time_s+Ts)
            {
                time_s += Ts;
                i++;
                h = time_i - time_s;
                data_i_im[j] = indata1[i] + (h/Ts)*(indata1[i+1]-indata1[i]);
                data_i_re[j] = indata2[i] + (h/Ts)*(indata2[i+1]-indata2[i]);

/* Reset sampling time and interpolation time. The relative time is important
only. */

                time_s = 0.0;
                time_i = h + Ti;
                if (time_i > 2*Ts)
                {
                    time_s += Ts;
                    i++;
                }
            }

/* Second case is upshift in doppler. */

            else
            {
                h = time_i - time_s;
                data_i_im[j] = indata1[i] + (h/Ts)*(indata1[i+1]-indata1[i]);
                data_i_re[j] = indata2[i] + (h/Ts)*(indata2[i+1]-indata2[i]);
                time_i += Ti;
                if (time_i > time_s + Ts)
                {
                    time_s += Ts;
                    i++;
                    time_i -= time_s;
                    time_s = 0.0;
                }
            }
        }
    }

```

```

/* Save indatal and indata2 working space. */

    temp_ptr1 = indatal;
    temp_ptr2 = indata2;
    *temp_ptr1++ = indatal[i+1];
    *temp_ptr2++ = indata2[i+1];

/* Set pointers to interpolated values */

    indatal = data_i_im;
    indata2 = data_i_re;
}

/* Decimate in time according to step. And compute sequence dc level. */

    n=0;
    dclvl_re = 0.0;
    dclvl_im = 0.0;
    for (j=0; j<vect_len; j+=step)
    {
        im_data[n] = indatal[j];
        re_data[n++] = indata2[j];
    }

/* Restore indatal and indata2 working space, when processing for doppler. */

    if (doppler != 0.0)
    {
        indatal = temp_ptr1;
        indata2 = temp_ptr2;
    }

/* Scramble data vector and process data according to interleave via FHT
   (HADAMARD) and unscramble. Real and imaginary parts are processed
   separately. */

    for (j=0; j<num_pts; j++)
    {
        h_data_im[0] = 0;
        h_data_re[0] = 0;
        n=0;
        for (k=0; k<(seq_len*num_pts); k+=num_pts)
        {
            h_data_im[scram[n]] = im_data[k+j];
            h_data_re[scram[n++]] = re_data[k+j];
        }
        hadamard(degree, h_data_im);
        hadamard(degree, h_data_re);
        n=0;
        dclvl_re+=h_data_re[0]; /* Save pedestal information. */
        dclvl_im+=h_data_im[0];
        for (k=0; k<(seq_len*num_pts); k+=num_pts)
        {
            im_data[k+j] = h_data_im[unscram[n]];
            re_data[k+j] = h_data_re[unscram[n++]];
        }
    }

/* Compute pedestal corrections for real and imaginary parts. */

    pdstal_re = (dclvl_re/num_pts)*rfact - (dclvl_im/num_pts)*ifact;

```

```

        pdstal_im = (dclvl_re/num_pts)*ifact + (dclvl_im/num_pts)*rfact;

/* Case 1 is if Coherent averaging is performed. Else, no coherent averaging
is performed. Each segment is output. */

        if (num_coh_avg > 1)
        {

/* Make DC level correction. */

                for(k=0;k<seq_len*num_pts;k++)
                {
                        avg_re[k] += (re_data[k]-pdstal_re);
                        avg_im[k] += (im_data[k]-pdstal_im);
                }
                count++;
                if (count==num_coh_avg)
                {
                        count = 0;
                        for (k=0;k<seq_len*num_pts;k++)
                        {
                                avg_re[k] /= num_coh_avg;
                                avg_im[k] /= num_coh_avg;
                        }

/* Compute magnitude and phase and print results. */

                                mag_phase(avg_re,avg_im,mag,phase,(num_pts*seq_len));
                                for (j=0;j<num_pts*seq_len;j++)
                                {
                                        fprintf(fp2,"%8.1f %8.1f\n",mag[j],phase[j]);
                                        avg_re[j] = 0.0;
                                        avg_im[j] = 0.0;
                                }
                        }
                }

/* Case 2. No coherent averaging. Compute magnitude and phase and
print results. */

        else
        {
                for (k=0;k<seq_len*num_pts;k++)
                {
                        re_data[k] -= pdstal_re;
                        im_data[k] -= pdstal_im;
                }
                mag_phase(re_data,im_data,mag,phase,(num_pts*seq_len));
                for (j=0;j<num_pts*seq_len;j++)
                        fprintf(fp2,"%8.1f %8.1f\n",mag[j],phase[j]);
        }
}

/* Finished close out variables and files. */

fclose(fp1);
fclose(fp2);
free(fc);
free(fs);

```

```

free(indata1);
free(indata2);
free(im_data);
free(re_data);
free(mag);
free(phase);
free(avg_re);
free(avg_im);
free(h_data_im);
free(h_data_re);
free(data_i_re);
free(data_i_im);

}

/* NEW is a short function to allocate memory for floating point vectors. */

float *new(size)
int size;
{
float *newdata;

if (( newdata = (float *)malloc(size*sizeof(float)))==NULL)
{
printf("Cannot Allocate Storage!\n");
exit(1);
}
return(newdata);
}

```

B. INITIALIZATION PROGRAMS

1. INIT_HAD

This program prompts for the primitive polynomial initial register load and allocates memory space to the variables scram and unscram. Returns initial register load, initial.

/* INIT_HAD is a program to initialize memory allocation for the scrambling and unscrambling arrays to be used with Fast Hadamard Transform. It also returns the initial register load used with the Shift Register Generator.

Variables:

scram - external array to hold scrambling indices.

unscram - external array to hold unscrambling indices.

initial - initial register load for SRG.

law - polynomial law that defines the SRG structure.

length - length of sequence generated by the input law. Assumes

```

        maximal length polynomial.

        temp - temporary holding spot for determining sequence length.*/

#include "macrofile.h"
unsigned init_had()
{
    extern unsigned law;
    unsigned initial, temp, length;

    printf("\nEnter polynomial law for the desired M-Sequence. Use");
    printf("\noctal integer representation only.");
    printf("\n(e.g. D3 + D + 1 = 1011 binary => 13 Octal.)\n");
    printf("\n      LAW: ");
    scanf("%o",&law);
    temp = law;
    length = 1;
    while (temp>>=1)
        length<<=1;
    length--;
    if ((scram = (unsigned *)malloc(length*sizeof(unsigned))) == NULL)
    {
        printf("\nCannot Allocate Scram Array!!!!\n");
        exit(1);
    }
    if ((unscram = (unsigned *)malloc(length*sizeof(unsigned))) == NULL)
    {
        printf("\nCannot Allocate Unscram Array!!!!\n");
        exit(1);
    }
    printf("\nEnter the initial register load in Decimal, do not use zero.");
    printf("\n\n      Initial Load: ");
    scanf("%u",&initial);
    if (initial == 0)
    {
        printf("\nError! Initial load cannot be 0!!\n\n");
        printf("\nEnter the initial register load in Decimal, do not use zero.");
        printf("\n\n      Initial Load: ");
        scanf("%u",&initial);
        if (initial == 0)
        {
            printf("\nError! Aborting.\n\n");
            exit(1);
        }
    }
    return(initial);
}

```

2. FWD_HAD

Computes the permutation indices for use with the FHT. Data is transmitted in the forward direction. Returns the degree of the primitive polynomial.

```

/* FWD_HAD() computes the permutation indices for scrambling and
   unscrambling a data vector generated using Maximal Length Sequences.
   These indices are used with the Fast Hadamard Transform and is performed
   in place. Sequences are assumed to be transmitted in the forward

```

direction.

Variables:

law - Shift Register Generator law to be used.

initial - initial register law.

scram - array of indices for scrambling data.

unscram - restoration indices for use after FHT.

degree - gives the degree of the polynomial law.

temp - temporary holder for computing degree and seq_len.

S_law - law formed from law to implement the S_gen structure.

rev_law - law in the reverse order to implement the L_gen structure.

end_bit - maintains end bit for around end feed in S_gen.

L_gen - variable that acts as L generator delay registers.

S_gen - variable that acts as S generator delay register.

Reference: The Feedback generators used for forming the forward scrambling and unscrambling indices are adapted from:

M. Cohn and A Lempel, 'On Fast M-Sequence Transforms',
IEEE Transactions on Information Theory, Jan. 1977. */

```
unsigned fwd_had(law,initial,scram,unscram)
unsigned law, initial, *scram, *unscram;
{
    unsigned degree,temp,temp2,seq_len,S_law,rev_law;
    unsigned end_bit,L_gen,S_gen;
    int i,j,count;

    /* Initialize variables. */
    temp = law;
    degree = 0;
    S_gen = 0;
    rev_law = 0;
    seq_len = 1;

    /* Computes the length of the M-Sequence, and Polynomial Degree. */
    while(temp>>=1)
    {
        seq_len <=<= 1;
        degree++;
    }

    /* Reverses law for use with L_gen. */
    temp = law;
    for (i=0; i<= degree;i++)
    {
        rev_law = (temp&1)|(rev_law<<1);
        temp>>=1;
    }
```

```

/* Set end_bit for register end feed for S_gen logic. Set initial load
   for S_gen. Set law for use in S_gen logic.*/
end_bit = seq_len--;
S_law = (law>>1);
end_bit>>=1;
temp = initial;
for (i=0; i<seq_len-1; i++)
{
    if (temp&1)
        temp = (temp^law)>>1;
    else
        temp >>= 1;
    if(i >= seq_len-degree)
    {
        S_gen = S_gen|(temp&1);
        S_gen <<=1;
    }
}
S_gen |= (initial&1);
/* Load L_gen to generate unscrambling indices. */
L_gen = (1<<(degree-1));

/* Compute permutations scram and unscram. */
for (i=0; i<seq_len; i++)
{
    unscram[i] = L_gen;
    temp2 = 0;
    temp = S_gen;
    for (j=0; j<degree; j++)
    {
        temp2 <<=1;
        temp2 |= (temp&1);
        temp >>=1;
    }
    scram[i] = temp2;
    if (L_gen&1)
        L_gen = (L_gen^rev_law)>>1;
    else
        L_gen>>=1;
    temp = (S_gen & S_law);
}

/* Count the number of 1's for modulus two sum for end feedback to S_gen. */
count=0;
for (j=0; j<degree; j++)
{
    if (temp&1)
    {
        count++;
        temp>>=1;
    }
    else
        temp>>=1;
}
if ( (count%2) == 0)
    S_gen <<= 1;
else
    S_gen = (S_gen<<1)|1;
S_gen &= seq_len;
}
return(degree);

```

)

3. REV_HAD

Computes the permutation indices for use with the FHT. Data is transmitted in the reverse direction. Returns the degree of the primitive polynomial [Ref. 15].

```
/* REV_HAD() computes the scrambling and unscrambling indices for a
Maximal Length Sequence that is to be processed using a Fast Hadamard
Transform and is performed in place. The sequence is assumed to be
transmitted in the reverse direction.

law - The polynomial law defining the Maximal Length Sequence that is
used.

initial - The initial register load for the shift register generator.

scram - Pointer to the array to contain the scrambling indices,
to be used with the transformed array. The transform
is not done in place.

unscram - Pointer to the array to contain the unscrambling indices.
The transform is not done in place.

degree - The degree of the law is returned.

Reference: This procedure was adapted from the article "On Fast
M-Sequence Transforms", by Martin Cohn and Abraham Lempel,
IEEE Transactions on Information Theory, Jan. 1977.

The program was modified from code provided by Kirk
Metzger with permission. */

#include "macrofile.h"

unsigned rev_had(law,initial,scram,unscram)
unsigned int law, initial, *scram, *unscram;
{
/* temp Scratch variable for use in finding degree etc.
end_bit Bit is set corresponding to highest bit in law. (i.e. the
end register n).
rev_initial Initial register load for reverse generation.
seq_len Length of sequence based on law.
index Loop counter.
ss_contents Shift register contents for scram array.
ms_contents M sequence register contents for unscram array.*/

unsigned int degree, temp, end_bit, rev_initial, seq_len, index;
register unsigned ss_contents, ms_contents;

/* Initialize variables. */
temp = law;
seq_len = 1;
rev_initial = 0;
degree = 0;

/* Find sequence length and degree of polynomial. */
while (temp>>=1)
{
```

```

    seq_len<<=1;
    degree++;
}

/* set end_bit */
end_bit = seq_len-->>1;

/* find reverse generator initial load */
for(index = 0; index < degree; index++)
{
    rev_initial = (rev_initial<<1)|(initial&1);
    initial>>=1;
}

/* generate scram and unscram values using law, end_bit and values of
ss_contents and ms_contents. */
ms_contents = 1;
ss_contents = rev_initial;
for (index =0; index<seq_len; index++)
{
    *scram++ = ss_contents;
    *unscram++ = ms_contents;
    temp = ss_contents&law;
    ss_contents>>=1;
    do { if (temp&1)
        ss_contents^=end_bit;}
    while (temp>>=1);
    if (ms_contents&1)
        ms_contents = (ms_contents^law)>>1;
    else
        ms_contents>>=1;
}
return (degree);
}

```

4. GET_FLT_CO.

Retrieves the filter coefficients to be used in initializing the filtering programs. Two tenth order and one fifth order filters are used in this program.

```

/* GET_FLT_COEF retrieves the filter coefficients from the user specified
file using a columnar format. The first column is the b coefficients
and the second is the a coefficients. The first 11 rows are the highpass
coefficients used for the upper end of the passband filter and the next
eleven are the lowpass coefficients for the lower end of the passband.
the final six rows are for the low pass filter used after demodulation.

```

```

The coefficients file may be created by any program desired by the user
but must be in the appropriate format. However, a MATLAB program,
set_filter.m, does this automatically. Set_filter.m uses a BUTTERWORTH
filter (10th order) for the passband and a CHEBYCHEV filter for the
lowpass filter (5th order). */

```

```

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

```

```

void get_fit_coef(hi_a,hi_b,low_a,low_b,lowl_a,lowl_b)
float *hi_a,*hi_b,*low_a,*low_b,*lowl_a,*lowl_b;
{
int i;
FILE *fp;
char filterfile[21];

printf("\nEnter the filename with the desired filter coefficients.");
printf("(20 Characters Max)\n");
printf("\n      Filename: ");
scanf("%s",filterfile);
if((fp = fopen(filterfile,"r"))== NULL)
{
printf("\nUnable to open file.  Aborting!\n");
exit(1);
}
for (i=0;i<=10;i++)
fscanf(fp,"%f %f\n",&hi_b[i],&hi_a[i]);
for (i=0;i<=10;i++)
fscanf(fp,"%f %f\n",&lowl_b[i],&lowl_a[i]);
for (i=0; i<=5; i++)
fscanf(fp,"%f %f\n",&low_b[i],&low_a[i]);
fclose(fp);
}

```

C. DEMODULATION AND FILTERING

1. DEMODULATE

This function performs demodulation of the input carrier signal as well as the necessary frequency shift when compensating for Doppler. No assumptions are made about the sampling frequency within this program. It may be set and cleared as desired. The time index *n* is remembered so the repetitive calls may be made.

/* DEMODULATE performs complex demodulation of a sinusoidal carrier for digital signal processing. Given an input signal, sampling frequency, and carrier frequency two outputs are produced, one for the real part and one for the imaginary part. The user determines the number of points to be processed in each call to the program.

sin_out - Input signal and output as imaginary part of the demodulated signal. Input data is overwritten and lost forever.
On initialization *sin_out* is set to the desired carrier frequency. (floating type pointer.)

cos_out - Output signal as real part of the demodulated signal.
On initialization *cos_out* is set to the desired sample frequency. (floating type pointer.)

dopp_bin - Denotes the doppler frequency shift in the sampling frequency to be used when doppler processing is used, and is determined externally. When no doppler processing is to be used it must

be set to 0.

n - counter that maintains time increment for digit processing (static).

theta - digital frequency (static).

flag - Set when system has been initialized (static). Cleared when N has been set to zero.

N - the number of data points to be processed. Whenever set to zero the system must be reset.

Reset: call with all arguments, N=0.

Set :First call with sin_out=fc, cos_out=fs, N is don't care.

Operation : Call with sin_out as input data, returns sin_out as imaginary part and cos_out as real part. N= number of data points to process and is the same length as the input data. */

```
#include <math.h>
#define PI 3.1415926536

void demodulate(sin_out,cos_out,dopp_bin,N)
float *sin_out, *cos_out, dopp_bin;
int N;
{
    int i;
    static int flag, n;
    static double theta;

    if (flag == 0)
    {
        flag = 1;
        theta = 2 * PI * (*sin_out/(*cos_out));
        n = 0;
        return;
    }
    else if( N == 0)
    {
        flag = 0;
        theta = 0.0;
        n = 0;
        return;
    }
    else
    {
        for(i=0; i<N; i++)
        {
            *cos_out++ = *sin_out * cos(n*(theta + dopp_bin));
            *sin_out++ = *sin_out * sin(n*(theta + dopp_bin));
            n++;
        }
        return;
    }
}
```

2. HIGHPASS

This program performs a highpass filtering of input data. It is assumed to be used in conjunction with a lowpass filter to form a bandpass filter. Processing in this manner simplifies the processing by avoiding complex operations. It maintains its state on subsequent calls and can be initialized with any type filter up to order 20. It may also be cleared and reset. (Note: All filtering programs are generic and may be adapted for any filtering application, lowpass, highpass, bandpass and bandstop. The names were chosen to distinguish them from each other within SEQREM.)

/* HIGHPASS is a filtering program that operates using difference equations. It is intended that the filter be IIR but this is not required. (i.e. $H(z) = B(z)/A(z)$ is a polynomial where the B_n 's and A_n 's specify the coefficients of the difference equation, which must of the same length. If they are not of the same order zeros must be appended.)

Initialization: The first call sets the desired filter coefficients to be used throughout. No filtering is performed at this stage. In this case `in_num` specifies numerator and `out_den` specifies the denominator coefficients. `M` is the number of coefficients, ordered highest to lowest. (NOTE: `out_den[0]` is assumed to be one and is not actually set during the initialization.)

Example 1: $H(z) = 1/z$; $\Rightarrow M = 2$, `in_num[0] = 0`, `in_num[1] = 1`,
`out_den[0] = 1`, `out_den[1] = 0`

Example 2: $H(z) = (0.5z^2 + 1)/(z^2 + 0.5z + 0.6)$ $\Rightarrow M = 3$,
`in_num[0] = 0.5`, `in_num[1] = 0`, `in_num[2] = 1`,
`out_den[0] = 1`, `out_den[1] = 0.5`, `out_den[2] = 0.6`

Filtering: Subsequent calls perform the desired filtering. In this case `in_num` is the input data, `out_den` is the output data from the filter, `M` specifies the number of data points being processed, and may be of any length as long as sufficient space has been allocated to the input pointer and the output pointer.

Clearing: To clear and reinitialize the filter after initializing once, simply set `M` to 0 and call again as explained above.

Data Types: Inputs are pointers to array's of float, `M` is integer.

Filter Order: Maximum order is 25.

Variables:

`in_num` - pointer to input numerator coefficients, or to input data segment.

in_den - pointer to input denominator coefficients, or to output data segment. (NOTE: input and output segments may be the same, but input values will be overwritten.)

A - Denominator filter coefficients (static).

B - Numerator filter coefficients (static).

Y - Output storage buffer used in recursion (static).

X - Input storage buffer used in recursion (static).

N - Remembers filter order for computing output (static).

M - Filter order during initialization, and number of points to be processed on subsequent calls.

flag - maintains initialization status (static).

Other Filters: LOWPASS1(), LOWPASS2(), and LOWPASS3() are coded identically to this program and work in the same manner. */

```
#include <math.h>
#define MAXORD 25
#define MAX 24

void hipass(in_num,out_den,M)
float *in_num, *out_den;
int M;
{
    static float A[MAXORD], B[MAXORD], Y[MAXORD], X[MAXORD];
    static int flag, N;
    int i,j;

    /* Loop clears output and coefficient values for reuse in new filter.*/
    if (M==0)
    {
        for (i=0;i<MAXORD;i++)
        {
            A[i] = 0;
            B[i] = 0;
            Y[i] = 0;
            X[i] = 0;
        }
        flag = 0;
        return;
    }
    /* Loop sets coefficients of the desired filter on first entry if flag=0 */
    /* Note that A[0] is assumed 1, since it corresponds to the desired output*/
    else if (flag == 0)
    {
        B[0] = in_num[0];
        for (i=1; i<M;i++)
        {
            B[i] = in_num[i];
            A[i] = out_den[i];
        }
        flag = 1;
        N = M;
    }
}
```

```

    return;
}
/* Otherwise perform filtering operations. Output stored in out_den[]. */
else
{
    for(j=0; j<M; j++)
    {
        X[MAX] = in_num[j];
        /* Start filtering operation. */
        Y[MAX] = B[0] * X[MAX];
        for (i=1; i<N; i++)
            Y[MAX] = Y[MAX] + B[i]*X[MAX-i] - A[i]*Y[MAX-i];
        out_den[j] = Y[MAX];
        /* Perform time shift of data points stored in filter. */
        for (i=0; i<MAX; i++)
        {
            Y[i] = Y[i+1];
            X[i] = X[i+1];
        }
    }
    return;
}
}

```

3. LOWPASS1

This program performs a lowpass filtering of input data. It operates exactly the same as HIGHPASS.

/* LOWPASS1 is a filtering program that operates using difference equations. It is intended that the filter be IIR but this is not required. (i.e. $H(z) = B(z)/A(z)$ is a polynomial where the B_n 's and A_n 's specify the coefficients of the difference equation, which must of the same length. If they are not of the same order zeros must be appended.)

Initialization: The first call sets the desired filter coefficients to be used throughout. No filtering is performed at this stage. In this case in_num specifies numerator and out_den specifies the denominator coefficients. M is the number of coefficients, ordered highest to lowest. (NOTE: out_den[0] is assumed to be one and is not actually set during the initialization.)

Example 1: $H(z) = 1/z$; $\Rightarrow M = 2$, in_num[0] = 0, in_num[1] = 1,
out_den[0] = 1, out_den[1] = 0

Example 2: $H(z) = (0.5z^2 + 1)/(z^2 + 0.5z + 0.6)$ $\Rightarrow M = 3$,
in_num[0] = 0.5, in_num[1] = 0, in_num[2] = 1,
out_den[0] = 1, out_den[1] = 0.5, out_den[2] = 0.6

Filtering: Subsequent calls perform the desired filtering. In this case in_num is the input data, out_den is the output data from the filter, M specifies the number of data points being processed, and may be of any length as long as sufficient space has been allocated to the input pointer and the output pointer.

Clearing: To clear and reinitialize the filter after initializing once, simply set M to 0 and call again as explained above.

Data Types: Inputs are pointers to array's of float, M is integer.

Filter Order: Maximum order is 25.

Variables:

in_num - pointer to input numerator coefficients, or to input data segment.

in_den - pointer to input denominator coefficients, or to output data segment. (NOTE: input and output segments may be the same, but input values will be overwritten.)

A - Denominator filter coefficients (static).

B - Numerator filter coefficients (static).

Y - Output storage buffer used in recursion (static).

X - Input storage buffer used in recursion (static).

N - Remembers filter order for computing output (static).

M - Filter order during initialization, and number of points to be processed on subsequent calls.

flag - maintains initialization status (static).

Other Filters: HIGHPASS(), LOWPASS2(), and LOWPASS3() are coded identically to this program and work in the same manner. */

```
#include <math.h>
#define MAXORD 25
#define MAX 24

void lowpass1(in_num,out_den,M)
float *in_num, *out_den;
int M;
{
    static float A[MAXORD], B[MAXORD], Y[MAXORD], X[MAXORD];
    static int flag, N;
    int i,j;

    /* Loop clears output and coefficient values for reuse in new filter.*/
    if (M==0)
    {
        for (i=0;i<MAXORD;i++)
        {
            A[i] = 0;
            B[i] = 0;
            Y[i] = 0;
            X[i] = 0;
        }
        flag = 0;
        return;
    }
    /* Loop sets coefficients of the desired filter on first entry if flag=0 */
```

```

/* Note that A[0] is assumed 1, since it corresponds to the desired output*/
else if (flag == 0)
{
    B[0] = in_num[0];
    for (i=1; i<M; i++)
    {
        B[i] = in_num[i];
        A[i] = out_den[i];
    }
    flag = 1;
    N = M;
    return;
}
/* Otherwise perform filtering operations. Output stored in out_den[]. */
else
{
    for(j=0; j<M; j++)
    {
        X[MAX] = in_num[j];
        /* Start filtering operation. */
        Y[MAX] = B[0] * X[MAX];
        for (i=1; i<N; i++)
            Y[MAX] = Y[MAX] + B[i]*X[MAX-i] - A[i]*Y[MAX-i];
        out_den[j] = Y[MAX];
        /* Perform time shift of data points stored in filter. */
        for (i=0; i<MAX; i++)
        {
            Y[i] = Y[i+1];
            X[i] = X[i+1];
        }
    }
    return;
}
}

```

4. LOWPASS2

This program performs a lowpass filtering of input data. It operates exactly the same as HIGHPASS.

/* LOWPASS2 is a filtering program that operates using difference equations. It is intended that the filter be IIR but this is not required. (i.e. $H(z) = B(z)/A(z)$ is a polynomial where the B_n 's and A_n 's specify the coefficients of the difference equation, which must of the same length. If they are not of the same order zeros must be appended.)

Initialization: The first call sets the desired filter coefficients to be used throughout. No filtering is performed at this stage. In this case in_num specifies numerator and out_den specifies the denominator coefficients. M is the number of coefficients, ordered highest to lowest. (NOTE: out_den[0] is assumed to be one and is not actually set during the initialization.)

Example 1: $H(z) = 1/z$; $\Rightarrow M = 2$, in_num[0] = 0, in_num[1] = 1,
out_den[0] = 1, out_den[1] = 0

Example 2: $H(z) = (0.5z^2 + 1)/(z^2 + 0.5z + 0.6) \Rightarrow M = 3$,
 $in_num[0] = 0.5, in_num[1] = 0, in_num[2] = 1$,
 $out_den[0] = 1, out_den[1] = 0.5, out_den[2] = 0.6$

Filtering: Subsequent calls perform the desired filtering. In this case in_num is the input data, out_den is the output data from the filter, M specifies the number of data points being processed, and may be of any length as long as sufficient space has been allocated to the input pointer and the output pointer.

Clearing: To clear and reinitialize the filter after initializing once, simply set M to 0 and call again as explained above.

Data Types: Inputs are pointers to array's of float, M is integer.

Filter Order: Maximum order is 25.

Variables:

in_num - pointer to input numerator coefficients, or to input data segment.

in_den - pointer to input denominator coefficients, or to output data segment. (NOTE: input and output segments may be the same, but input values will be overwritten.)

A - Denominator filter coefficients (static).

B - Numerator filter coefficients (static).

Y - Output storage buffer used in recursion (static).

X - Input storage buffer used in recursion (static).

N - Remembers filter order for computing output (static).

M - Filter order during initialization, and number of points to be processed on subsequent calls.

flag - maintains initialization status (static).

Other Filters: $HIGHPASS()$, $LOWPASS1()$, and $LOWPASS3()$ are coded identically to this program and work in the same manner. */

```
#include <math.h>
#define MAXORD 25
#define MAX 24

void lowpass2(in_num,out_den,M)
float *in_num, *out_den;
int M;
{
    static float A[MAXORD], B[MAXORD], Y[MAXORD], X[MAXORD];
    static int flag, N;
    int i,j;

    /* Loop clears output and coefficient values for reuse in new filter.*/
    if (M==0)
    {
```

```

    for (i=0; i<MAXORD; i++)
    {
        A[i] = 0;
        B[i] = 0;
        Y[i] = 0;
        X[i] = 0;
    }
    flag = 0;
    return;
}
/* Loop sets coefficients of the desired filter on first entry if flag=0 */
/* Note that A[0] is assumed 1, since it corresponds to the desired output*/
else if (flag == 0)
{
    B[0] = in_num[0];
    for (i=1; i<M; i++)
    {
        B[i] = in_num[i];
        A[i] = out_den[i];
    }
    flag = 1;
    N = M;
    return;
}
/* Otherwise perform filtering operations. Output stored in out_den[] */
else
{
    for (j=0; j<M; j++)
    {
        X[MAX] = in_num[j];
        /* Start filtering operation. */
        Y[MAX] = B[0] * X[MAX];
        for (i=1; i<N; i++)
            Y[MAX] = Y[MAX] + B[i]*X[MAX-i] - A[i]*Y[MAX-i];
        out_den[j] = Y[MAX];
        /* Perform time shift of data points stored in filter. */
        for (i=0; i<MAX; i++)
        {
            Y[i] = Y[i+1];
            X[i] = X[i+1];
        }
    }
    return;
}
}

```

5. LOWPASS3

This program performs a lowpass filtering of input data. It operates exactly the same as HIGHPASS.

```

/* LOWPASS3 is a filtering program that operates using difference
equations. It is intended that the filter be IIR but this is not
required. (i.e.  $H(z) = B(z)/A(z)$  is a polynomial where the Bn's and
An's specify the coefficients of the difference equation, which must
of the same length. If they are not of the same order zeros must be
appended.)

```

Initialization: The first call sets the desired filter coefficients to be used throughout. No filtering is performed at this stage. In this case in_num specifies numerator and out_den specifies the denominator coefficients. M is the number of coefficients, ordered highest to lowest. (NOTE: out_den[0] is assumed to be one and is not actually set during the initialization.)

Example 1: $H(z) = 1/z$; $\Rightarrow M = 2$, in_num[0] = 0, in_num[1] = 1, out_den[0] = 1, out_den[1] = 0

Example 2: $H(z) = (0.5z^2 + 1)/(z^2 + 0.5z + 0.6)$ $\Rightarrow M = 3$, in_num[0] = 0.5, in_num[1] = 0, in_num[2] = 1, out_den[0] = 1, out_den[1] = 0.5, out_den[2] = 0.6

Filtering: Subsequent calls perform the desired filtering. In this case in_num is the input data, out_den is the output data from the filter, M specifies the number of data points being processed, and may be of any length as long as sufficient space has been allocated to the input pointer and the output pointer.

Clearing: To clear and reinitialize the filter after initializing once, simply set M to 0 and call again as explained above.

Data Types: Inputs are pointers to array's of float, M is integer.

Filter Order: Maximum order is 25.

Variables:

in_num - pointer to input numerator coefficients, or to input data segment.

in_den - pointer to input denominator coefficients, or to output data segment. (NOTE: input and output segments may be the same, but input values will be overwritten.)

A - Denominator filter coefficients (static).

B - Numerator filter coefficients (static).

Y - Output storage buffer used in recursion (static).

X - Input storage buffer used in recursion (static).

N - Remembers filter order for computing output (static).

M - Filter order during initialization, and number of points to be processed on subsequent calls.

flag - maintains initialization status (static).

Other Filters: HIGHPASS(), LOWPASS1(), and LOWPASS2() are coded identically to this program and work in the same manner. */

```
#include <math.h>
#define MAXORD 25
#define MAX 24
```

```

void lowpass3(in_num,out_den,M)
float *in_num, *out_den;
int M;
{
static float A[MAXORD], B[MAXORD], Y[MAXORD], X[MAXORD];
static int flag, N;
int i,j;

/* Loop clears output and coefficient values for reuse in new filter.*/
if (M==0)
{
for (i=0;i<MAXORD;i++)
{
A[i] = 0;
B[i] = 0;
Y[i] = 0;
X[i] = 0;
}
flag = 0;
return;
}
/* Loop sets coefficients of the desired filter on first entry if flag=0 */
/* Note that A[0] is assumed 1, since it corresponds to the desired output*/
else if (flag == 0)
{
B[0] = in_num[0];
for (i=1; i<M;i++)
{
B[i] = in_num[i];
A[i] = out_den[i];
}
flag = 1;
N = M;
return;
}
/* Otherwise perform filtering operations. Output stored in out_den[] */
else
{
for(j=0; j<M; j++)
{
X[MAX] = in_num[j];
/* Start filtering operation. */
Y[MAX] = B[0] * X[MAX];
for (i=1;i<N;i++)
Y[MAX] = Y[MAX] + B[i]*X[MAX-i] - A[i]*Y[MAX-i];
out_den[j] = Y[MAX];
/* Perform time shift of data points stored in filter. */
for (i=0;i<MAX;i++)
{
Y[i] = Y[i+1];
X[i] = X[i+1];
}
}
return;
}
}

```

D. FAST HADAMARD TRANSFORM (FHT)

1. HADAMARD

This function performs the FHT of an input data vector in place. A power of two length is required. Computation is identical to the FFT butterfly process with the complex multiplies set to one.

```
/* HADAMARD performs the Fast Hadamard Transform (FHT) on the input data.
   The transform is performed in place and assumes no interleaving.
   Interleaving must be performed externally.
```

```

   The FHT performs the same operations as the FFT with the complex
   multiplies set to + or - 1. The input and output data vector
   must be first ordered and then reordered after exit.
```

```
   Variables:
```

```
   degree - Provides the degree of the polynomial being worked with
             which gives the length of the M-Sequence.
```

```
   din - Pointer to the input data to be transformed. On exit it holds
          the transformed data.
```

```
   m,n - counters.
```

```
   itop, ispace, iwidth, ibot - common place holders used performing
                                the butterfly operations of the FFT. */
```

```
#include <math.h>
```

```
void hadamard(degree,din)
float *din;
int degree;
{
    int m,n,itop, ispace,iwidth,ibot;
    float temp;
    unsigned int one, size;

    one = 1;
    size = (one<<degree);
    for ( m=1; m<=degree;m++)
    {
        ispace = (one<<m);
        iwidth = (one<<(m-1));
        for (n=0;n<iwidth;n++)
            for( itop=n;itop<=(size-2);itop+=ispace)
            {
                ibot = itop + iwidth;
                temp = din[ibot];
                din[ibot] = din[itop] - temp;
                din[itop] = din[itop] + temp;
            }
    }
}
```

E. MAGNITUDE AND PHASE

1. MAG_PHASE

This function simply computes the magnitude and phase of the demodulates after either coherent averaging or directly after the FHT and the appropriate permutations.

```
/* MAG_PHASE computes the magnitude and phase of the input data, and
   returns the results in the pointers mag and phase.

   realpart - Pointer for the real part of the input data of length N.

   impart - Pointer for the imaginary part of the input data of length N.

   mag - Pointer to data segment to hold the magnitude of the data and
        is also of length N.

   phase - Pointer for data segment to hold the phase of the data and is
          also of length N. (Degrees)

   N - Integer indicating the size of the input data segment.   */

#include <math.h>
#define PI 3.1415927

void mag_phase(realpart,impart,mag,phase,N)
float *realpart, *impart, *mag, *phase;
int N;
{
    int i;

    for (i=0;i<N;i++)
    {
        mag[i] = sqrt(realpart[i] * realpart[i] + impart[i] * impart[i]);
        if (realpart[i] == 0.0)
        {
            phase[i] = 0.0;
        }
        else
            phase[i] = atan(impart[i] / realpart[i]) * (180.0/PI);
    }
}
```

F. UTILITY PROGRAMS

1. SET FILTER

This is a handy MATLAB program for finding the coefficients for a tenth order Butterworth bandpass filter and a fifth order Chebychev lowpass filter, which stores them in a file formatted for use with GET_FLT_COEF. Sampling frequency and cutoff frequencies are prompted from the user.

```
% function set_filter      Set_filter computes the filter coefficients for
%                          the bandpass filter (high and lowpass filters)
%                          and for the lowpass filter used in the sequence
%                          removal program. The bandpass region uses a
%                          10th order BUTTERWORTH filter and the lowpass region
%                          uses a 5th order CHEBYCHEV filter with the results
%                          stored in the user specified file in the format
%                          required by the GET_FLT_COEF function. The user is
%                          queried for cutoff frequencies and sampling
%                          frequency.

function set_filter

fprintf('\n                          WARNING!!!!!!  \n');
fprintf('\n File to hold results should be non-existent prior to running\n');
fprintf(' this routine because the results are appended to any pre-existing\n');
fprintf(' data.\n');
filename = input('Enter filename to store results: ','s');
fs = input('Enter the Sampling frequency to be used: ');
fc = input('Enter the Cut Off Frequency for the Low Pass Filter: ');
fprintf('\n Enter the Cut Off Frequencies for the BP filter.\n');
fl = input('Lower: ');
fh = input('Upper: ');
fs = fs/2;
if (fl > fs) || (fc > fs) || (fh > fs)
    error('Invalid Frequencies. Cutoff Frequencies Must be <= Fs/2');
end
% Finds the Coefficients for the High Pass Filter (BP low end)
wn = fl/fs;
[b1,a1] = butter(10,wn,'high');
% Finds the Coefficients for the Low Pass Filter (BP hi end)
wn = fh/fs;
[b2,a2] = butter(10,wn);
% Finds the Coefficients for the Low Pass Filter.
wn = fc/fs;
[b3,a3] = cheby1(5,0.5,wn);
for i=1:11
    fprintf(filename,'%e  %e\n',b1(i),a1(i));
end
for i=1:11
```

```

        fprintf(filename, '%e    %e\n', b2(i), a2(i));
end
for i=1:6
    fprintf(filename, '%e    %e\n', b3(i), a3(i));
end

```

2. APLOT

This program was used to generate the plots in Appendix A with NCAR graphics. It is data specific because of the nature of NCAR graphics when used for surface plotting routines.

C This is a program to plot the results of the program SEQ_REM
C in successive plots for each doppler searched.

```

        INTEGER X,Y
        PARAMETER (X=510, Y=30)

        CHARACTER*32 IFILE,TEMP,SEQLEN
        CHARACTER*20 DOPPLER
        INTEGER I,J,LENGTH,ISZ
        REAL MAG(X,Y),PH(X,Y),XPOS,YPOS,ZPOS
        REAL WORK(2*X*Y+X+Y)
        REAL TS,ANGH,ANGV,ZMAX
        COMMON /SRFIP1/ IFR,ISTP,IROTS,IDRX,IDRY,IDRZ,IUPPER,ISKIRT,
1          NCLA,THETA,HSKIRT,CHI,CLO,CINC,ISPVAL

35  FORMAT(A)
40  FORMAT(A20)

        IFR = 0
        IDRZ = 1
        WRITE(*,*) 'THIS PLOTS THE RESULTS FROM SEQREM PROGRAM'
        WRITE(*,*) 'ENTER THE INPUT FILE: '
        READ(*,40) IFILE
        OPEN(33,FILE=IFILE,STATUS='OLD')
        ANGH = -60.
        ANGV = 30.
        ZMAX = 0.
        CALL GOPKS(6,ISZ)
        CALL GOPWK(1,2,1)
        CALL GACWK(1)
        READ(33,35) TEMP

50  READ(UNIT=33,FMT=*) DOPPLER
        READ(UNIT=33,FMT=*) SEQLEN
        READ(UNIT=33,FMT=*) LENGTH,TS
        J = 1

60  J = J+1
        DO 80 I=1,LENGTH
        READ(UNIT=33,FMT=*,ERR=100,END=200) MAG(I,J),PH(I,J)
        IF (MAG(I,J) .GT. ZMAX) THEN
            ZMAX = MAG(I,J)
        ENDIF
80  CONTINUE

```

```

GO TO 60

100  J = J-1
      CALL GSELNT(0)
      CALL GSTXAL(2,3)
      CALL GSCHH(.02)
      CALL GTX(0.5,0.9,'ARRIVAL TIME PLOT ')
      CALL GSTXAL(2,3)
      CALL GTX(0.5,0.85,DOPPLER)
      CALL EZSRFC(MAG,LENGTH,J,ANGH,ANGV,WORK)
      XPOS = 0.
      YPOS = -0.25*ZMAX
      ZPOS = 0.0
      CALL PWRZS(XPOS,YPOS,ZPOS,SEQLN,31,25,1,3,0)
      CALL FRAME
      ZMAX = 0.
      GO TO 50

200  J = J-1
      CALL GSELNT(0)
      CALL GSTXAL(2,3)
      CALL GSCHH(.02)
      CALL GTX(0.5,0.9,'ARRIVAL TIME PLOT ')
      CALL GSTXAL(2,3)
      CALL GTX(0.5,0.85,DOPPLER)
      CALL EZSRFC(MAG,LENGTH,J,ANGH,ANGV,WORK)
      XPOS = 0.
      YPOS = -0.25*ZMAX
      ZPOS = 0.0
      CALL PWRZS(XPOS,YPOS,ZPOS,SEQLN,31,25,1,3,0)
      CALL FRAME
      CLOSE(33)
      CALL GDAWK(1)
      CALL GCLWK(1)
      CALL GCLKS
      WRITE(*,*) 'FINISHED'
      STOP
      END

```

3. DOPPLOT

This program was used to generate the arrival time plot, Figure 3.1, with NCAR graphics. It is data specific because of the nature of NCAR graphics when used for surface plotting routines.

```
C      This is a program to plot the results of the program SEQ_REM
C      making a doppler versus time plot for each doppler search.

      INTEGER X,Y
      PARAMETER (X=255, Y=23)

      CHARACTER*32 IFILE,TEMP,SEQLEN
      CHARACTER*20 DOPPLER
      INTEGER I,J,LENGTH,ISZ
      REAL  MAG(X,Y),PH(X,Y),XPOS,YPOS,ZPOS,TEMP1(X,Y),TEMP2(X,Y)
      REAL  WORK(2*X*Y+X+Y)
      REAL  TS,ANGH,ANGV,ZMAX
      COMMON /SRFIP1/ IFR,ISTP,IROTS,IDRX,IDRY,IDRZ,IUPPER,ISKIRT,
1      NCLA,THETA,HSKIRT,CHI,CLO,CINC,ISPVAL

35      FORMAT(A)
40      FORMAT(A20)

      IFR = 0
      IDRZ = 1
      WRITE(*,*) 'THIS PLOTS THE RESULTS FROM SEQREM PROGRAM IN DOPPLER'
      WRITE(*,*) 'VS.TIME FORMAT'
      WRITE(*,*) 'ENTER THE INPUT FILE: '
      READ(*,40) IFILE
      OPEN(33,FILE=IFILE,STATUS='OLD')
      ANGH = -60.
      ANGV = 30.
      ZMAX = 0.
      CALL GOPKS(6,ISZ)
      CALL GOPWK(1,2,1)
      CALL GACWK(1)
      READ(33,35) TEMP
      J = 1

50      READ(UNIT=33,FMT=*) DOPPLER
      READ(UNIT=33,FMT=*) SEQLEN
      READ(UNIT=33,FMT=*) LENGTH,TS
      K = 1
      DO 53 I=1,LENGTH
      READ(33,*) TEMP1(I,K),TEMP2(I,K)
53      CONTINUE
60      J = J+1
      DO 80 I=1,LENGTH
      READ(UNIT=33,FMT=*,ERR=50,END=200) MAG(I,J),PH(I,J)
      IF (MAG(I,J) .GT. ZMAX) THEN
          ZMAX = MAG(I,J)
      ENDIF
80      CONTINUE
82      CONTINUE
```

```

      DO 85 I=1,LENGTH
      READ(UNIT=33,FMT=*,ERR=50,END=200) TEMP1(I,K),TEMP2(I,K)
85    CONTINUE
      GO TO 82

200   J = J-1
      CALL GSELNT(0)
      CALL GSTXAL(2,3)
      CALL GSCHH(.02)
      CALL GTX(0.5,0.9,'ARRIVAL TIME VS DOPPLER')
      CALL GSTXAL(2,3)
      CALL GTX(0.5,0.85,'-5 TO 5 KNOTS DOPPLER')
      CALL EZSRFC(MAG,LENGTH,J,ANGH,ANGV,WORK)
      XPOS = 0.
      YPOS = -0.25*ZMAX
      ZPOS = 0.0
      CALL PWRZS(XPOS,YPOS,ZPOS,'ARRIVAL TIME 22.36 SEC.',31,25,1,3,0)
      CALL FRAME
      CLOSE(33)
      CALL GDAWK(1)
      CALL GCLWK(1)
      CALL GCLKS
      WRITE(*,*) 'FINISHED'
      STOP
      END

```

4. MGEN

MGEN generates a file with a m-sequence in the forward direction in column form, given a primitive polynomial. The sequence is in (1,-1) form. A second file is also created, which holds one period of the register states. This can be useful for debugging and as an aid for understanding m-sequences.

```

/* This is a generator routine to generate the output from the least
significant register in a M-Sequence shifter register generator.
Output is in the form of -1 and 1 where the mapping is {-1,1} ->
{1,0}. Register states are also stored to a specified file in decimal
form.

```

```

Input:  law = the polynomial law specifying the M-Sequence to be
          generated.
        initial = the initial load placed in the Shift Registers.

```

```

Output: output is a column format of -1 and 1's in floating format
        form for ease of use. output 2 is a column of numbers in decimal
        representing the register states over one period.

```

```

Initiation: use 'MGEN'

```

```

*/
#include <malloc.h>
#include <stdio.h>
#include <math.h>

```

```

main()
{
    float *mout;
    unsigned int temp, initial, law, seq_len;
    int i;
    char filename[20];
    FILE *fp1,*fp2;

    printf("\nEnter the Polynomial Law in Octal Form: ");
    scanf("%o",&law);
    printf("\nEnter the Initial Register Load in Octal(Not Zero!): ");
    scanf("%o",&initial);
    if (initial==0)
    {
        printf("\n0 is Invalid Register Load. Aborting!\n");
        exit(1);
    }
    temp = law;
    seq_len = 1;
    while(temp>>=1)
        seq_len<<=1;
    seq_len--;
    if ((mout = (float *)malloc(seq_len*sizeof(mout)))==NULL)
    {
        printf("\nCannot Allocate Memory to Output Variable\n");
        exit(1);
    }
    printf("\nEnter file to store M-sequence. (20 Characters max.);");
    scanf("%s",filename);
    fp1= fopen(filename,"w");
    printf("\nEnter file to store register states. (20 Characters max.);");
    scanf("%s",filename);
    fp2= fopen(filename,"w");
    for(i=0;i<seq_len;i++)
    {
        fprintf(fp2,"%u\n",initial);
        if(initial&1)
        {
            mout[i] = -1.0;
            initial = (initial^law)>>1;
        }
        else
        {
            mout[i] = 1.0;
            initial>>=1;
        }
    }
    fclose(fp2);
    for(i=0;i<seq_len;i++)
        fprintf(fp1,"%f\n",*mout++);
    fclose(fp1);
}

```

5. MAKEFILE

This is a file which makes compiling and linking of C programs easier on UNIX based systems. Files in the list are compiled, if not already, and are linked automatically with the MAKE command. The executable code is stored in the file following the -o flag, and -lm includes the necessary library routines.

```
#makefile for thesis work.

OBJECTS = seqrem.o rev_had.o init_had.o fwd_had.o get_flt_coef.o low_filter1.o low_filter2.o
low_filter3.o hi_filter.o demod.o hadamard.o mag_phase.o

SOURCES = seqrem.c set_had.c init_had.c fwd_had.c get_flt_coef.c low_filter1.c low_filter2.c
low_filter3.c hi_filter.c demod.c hadamard.c mag_phase.c

LIBS = -lm

CFLAGS = -O

seqrem: $(OBJECTS)

    cc $(CFLAGS) $(OBJECTS) -o seqrem $(LIBS)

$(OBJECTS): macrofile.h
```

6. EXAMPLE FILTER COEFFICIENT FILE

This shows the structure of the format used to store the filter coefficients used in the various filtering programs. The numerator coefficients constitute the first column and the denominator the second. The first eleven rows are for the first tenth order filter, the next the second, and finally the last six make up the fifth order lowpass filter.

6.127443e-03	1.000000e+00
-6.127443e-02	-9.960099e-01
2.757350e-01	1.759559e+00
-7.352932e-01	-1.112098e+00
1.286763e+00	8.747430e-01
-1.544116e+00	-3.474470e-01
1.286763e+00	1.441634e-01
-7.352932e-01	-3.307116e-02
2.757350e-01	6.691890e-03
-6.127443e-02	-6.798897e-04
6.127443e-03	3.914710e-05
6.127443e-03	1.000000e+00
6.127443e-02	9.960099e-01
2.757350e-01	1.759559e+00
7.352932e-01	1.112098e+00
1.286763e+00	8.747430e-01
1.544116e+00	3.474470e-01
1.286763e+00	1.441634e-01
7.352932e-01	3.307116e-02
2.757350e-01	6.691890e-03
6.127443e-02	6.798897e-04
6.127443e-03	3.914710e-05
1.438164e-05	1.000000e+00
7.190822e-05	-4.515326e+00
1.438164e-04	8.278174e+00
1.438164e-04	-7.695624e+00
7.190822e-05	3.625292e+00
1.438164e-05	-6.920557e-01

References

1. Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall, 1978.
2. Munk, W. and Wunsch, C., "Ocean Acoustic Tomography: A Scheme for Large Scale Monitoring," *Deep-Sea Research*, v. 26A, pp. 123 - 161, 1979.
3. Clare, F., Kennison, D., and Lackman, B., *NCAR Graphics Users Guide*, National Center for Atmospheric Research, 1987.
4. Chiu, C.S. and Ehert, L., personal communication, August 1990.
5. Birdsall, Theodore G., letter dated 23 April 1990, "The Heard Island Experiment."
6. Burdic, William S., *Underwater Acoustic System Analysis*, pp. 17 - 154, Prentice-Hall, 1984.
7. Clay, Clarence S. and Medwin, Herman, *Acoustical Oceanography*, pp. 29 - 136, John Wiley & Sons, 1977.
8. Kinsler, L.E., and others, *Fundamentals of Acoustics*, 3rd ed., pp. 117 - 120, John Wiley & Sons, 1982.
9. Dees, Robert C., *Signal Processing and Preliminary Results in the 1988 Monterey Bay Tomography Experiment*, MS Thesis, Naval Postgraduate School, Monterey, CA, June 1989.
10. Ziemer, Rodger E. and Peterson, Rodger L., *Digital Communications and Spread Spectrum Systems*, pp. 365-415, Macmillan Publishing Company, 1985.

11. Birdsall, T.G. and Metzger, K., "M-Sequence Signal Tutorial", *Naval Oceanographic Office Presentation*, EECS Dept., University of Michigan, April, 1988.
12. Birdsall, Ted, letter dated 27 May 1987, " My Introduction to Hadamard Processes for M-Sequences."
13. Birdsall, T.G., Heitmeyer, R.M., Metzger, K., "Modulation By Linear-Maximal Shift Register Sequences: Amplitude, Biphase and Complement-Phase Modulation", *Cooley Electronics Laboratory*, EECS Dept., University of Michigan, December 1987.
14. Spindel, Robert C., "Signal Processing in Ocean Tomography," *Adaptive Methods in Underwater Acoustics*, ed. H.G. Urban, pp. 687-710, D. Reidel Publishing Company, 1985.
15. Metzger, K., "Memo to M-Sequence Users on Existence of a Fast Algorithm for Sequence Removal," June 8, 1987.
16. Cohn, Martin and Lempel, Abraham, "On Fast M-Sequence Transforms," *IEEE Transactions on Information Theory*, pp. 135 - 137, January 1976.
17. Borish, Jeffrey and Angell, James, "An Efficient Algorithm for Measuring the Impulse Response Using Pseudorandom Noise, " *J. Audio Eng. Soc.*, Vol. 31, No. 7, pp. 478 - 488, July/August 1983.
18. Whalen, Anthony D., *Detection, of Signals in Noise*, pp. 196-225, Academic Press, 1971.
19. Oppenheim, Alan V. and Willsky, Alan S. with Young, Ian T., *Signals and Systems*, pp. 543-555, Prentice-Hall, 1983.
20. Birdsall, T.G., Metzger, K. and Spindel, Robert C., "Signal Processing for Ocean Tomography with Moving Ships", *Asilomar Conf. on Sig., Systems and Comp.*, Monterey, CA, November 1, 1988.